The Engineering of an Object-Oriented Software Development Methodology

Raman Ramsin

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY *of York*

YORK
UK

Department of Computer Science

April 2006

To Shari, My Dear Wife

For being my friend, my companion, my partner, my life

To My Dear Parents

For their unreserved love and support

To Mary and Ramina

For their prayers, and their love

Abstract

Software Engineering as a discipline has provided us with methodologies for developing software systems, yet it can also be used for developing methodologies themselves; after all, as observed and aptly stated by prominent researchers, software development methodologies are software too.

In a bid to address the problems plaguing object-oriented software development processes, this thesis presents a software engineering approach to methodology development: that is, through the generic software engineering phases of Analysis, Design, Implementation, and Test, applied in a risk-driven and iterative-incremental lifecycle. This abstract methodology engineering process has been used for developing an object-oriented methodology, and has thereby gradually evolved into a concrete lifecycle and *meta*-methodology for developing object-oriented methodologies.

As a further contribution of this thesis, the methodology that has been developed through application of the above meta-methodology addresses several key problems currently afflicting object-oriented software development processes. Targeting information systems, the methodology is Feature-Driven and iterative-incremental, and is based on smooth and seamless transition from real-world problem domain models to system models, and ultimately to design models. The system is initially designed as a homogeneous extension to the problem domain, using the same types of elements seen in the problem domain, thus smoothing the transition process. Seamless transition is achieved via the use of reengineering patterns, design patterns and refactoring patterns for iteratively transforming the system models into software design models through redistributing functionalities and refining the structure. Typical anomalies resulting from real-world problem domain modeling are thereby eradicated, while keeping the models tangible to both users and developers.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Dr. Richard Paige, for his invaluable help and guidance, and for always being there. I would also like to thank Dr. Fiona Polack, whose remarkable attention to detail has greatly improved the quality of this thesis.

Declaration

Process-centred descriptions of 24 methodologies and process patterns/metamodels were produced in the analysis phase of the methodology engineering process (reported in Chapter 3 of this thesis). General descriptions of seven of these methodologies (belonging to the *seminal* category) have been previously presented in an introductory chapter of the author's MSc thesis [Ramsin 1995].

The Estate Agency System, which is used as one of the two case studies in Chapter 6, is loosely based on a user-story presented by Xiaocheng Ge, later developed into the eXGrid case study [Ge et al. 2006].

Some of the material presented within this thesis has already been published or reviewed for publication, as listed below:

- RAMSIN, R., AND PAIGE, R.F. 2004. Process-centred review of object-oriented software development methodologies. Technical Report YCS-2004-381. University of York, York, UK.
- RAMSIN, R., AND PAIGE, R.F. 2004. Process-centred review of object-oriented software development methodologies. Submitted to ACM Computing Surveys (revision currently under review).

Chapter 1

Introduction

Software engineering has been evolving over the past thirty years, but it has never completely solved the software crisis [Pressman 2004]. As an integral part of the discipline of software engineering, software development methodologies have also evolved, from shallow and informal in-house methodologies of the late 1960s to the object-oriented methodologies of the 1990s and the new millennium. In the face of fierce resistance and inertia, paradigm shifts have been long and painful, yet Object-Oriented Software Development Methodologies (OOSDMs) have managed to survive, and indeed, prosper. The status quo, however, is far from desirable; object-oriented methodologies have been around for two decades, yet many of the problems associated with these methodologies two decades ago still remain unresolved today.

Aimed at addressing the problems plaguing OOSDMs, this thesis presents a software engineering approach to methodology development, proposing a methodology for requirements-based development of OOSDMs and applying it to produce a methodology. The resulting methodology is described in detail, as is the *meta*-methodology used for producing it, with methodology requirements used for validating the implemented methodology.

This chapter presents the motivations behind this thesis, its objectives and scope, and the research methodology used. A summary of the results has been included, enumerating the main contributions of the thesis. The structure of the remaining chapters of the thesis has also been delineated.

1.1 Motivations

This thesis has been motivated by problems afflicting object-oriented software development methodologies. This section presents the basis for the thesis, illuminating the problem areas and emphasizing the need for a comprehensive stocktaking of what has been achieved, and what remains to be done. Different approaches to methodology development are discussed, and arguments put forward as to why a systematic *engineering* approach is required. As further instances of motivations behind this thesis, the final subsection lists a number of relevant research roadmaps identified by the software engineering community.

1.1.1 The Need for a Retrospective Appraisal

The applicability of the object-oriented approach to systems analysis and design was realized in the mid 1980s, and as a result, the software industry witnessed the advent of a plethora of object-oriented software development methodologies. These methodologies were widely acclaimed as promising means for tackling the software crisis, yet their sheer number and diversity became detrimental to their widespread adoption into the software engineering community. The ensuing "Methodology War" led to efforts aimed at unification and standardization in the mid 1990s, resulting in the development of the UML and integrated (third generation) methodologies [Graham 2001]. This promised an end to the methodology war, but the present situation is far from what was initially expected. Attempts at integration, unification and standardization have actually aggravated the problems of complexity and inconsistency, giving rise to a new family of lightweight, *agile* methodologies, some of which eccentrically defy the long-established values of modeling and process-based development [Boehm and Turner 2004]. The integrated, heavyweight methodologies are very complex, and some of their competitors are little more than controlled code-&-fix methods based on good programming practices. While integrated methodologies are encumbered with unwieldy processes, agile methodologies have tried their best to have as little explicit process as possible.

The evolution process seems to have gone astray, and as a result, we are witnessing the return of some of the older methodologies (such as RDD [Wirfs-Brock et al.

1990, Wirfs-Brock and McKean 2002]). At the same time, some of the methodologies or variants introduced today (such as EUP [Ambler and Constantine 2000a], OPM [Dori 2002a], and FOOM [Shoval and Kabeli 2001]) do not even adhere to UML modeling conventions. On the other hand, the OMG's Model-Driven Architecture (MDA) [OMG 2001], the general development approach based on transforming logical models of the system (called Platform-Independent Models – PIMs) into physical implementation models (called Platform-Specific Models – PSMs) [Siegel and OMG 2001], is still in its early stages of development. It is by no means mature enough to spawn serious methodologies, explaining the lack of rigour in the few such methodologies so far introduced (such as [Gervais 2002]). Even though MDA has been hailed by its proponents as a panacea, many prominent figures in software engineering have expressed serious doubts as to the very feasibility of the MDA approach [Thomas 2004, Fowler 2004].

The course of events suggests that any effort aimed at enhancing object-oriented methodologies should also consider the abundant capabilities of older methodologies, neglected during the integration euphoria. In addition, special attention should be given to the fact that today's integrated methodologies and their agile counterparts have no other choice but to converge. In fact, there are signs of convergence [Boehm and Turner 2004] proving that the imbalance caused by the eccentric leanings of the two camps, disRUPtive overindulgence on one side and eXtreme negligence on the other, is prompting the call to moderation. Recent advances in the fields of process metamodeling and process patterns have also opened new possibilities for ameliorating the status quo.

A closer look at the present state of affairs in the field of object-oriented software development methodologies shows numerous deficiencies, including:

1. Requirements engineering is still the weak link, and requirements traceability is rarely supported; requirements are either not adequately captured or partially lost or corrupted during the development process [Nuseibeh and Easterbrook 2000].
2. Model inconsistency is a dire problem. UML has exacerbated the situation instead of improving it [Paige and Ostroff 2002, Dori 2002a,b].

3. Integrated methodologies are too complex to be effectively mastered, configured, and enacted [Highsmith 2000b, Boehm and Turner 2004]. Although most of them are designed in such a way as to accommodate customization and tailoring down, in practice they tend to rapidly build up and get out of hand [Boehm and Turner 2004].
4. Despite remarkable achievements, agile methodologies are still not mature enough [Abrahamsson et al. 2003, Boehm and Turner 2004, Boehm and Turner 2005, Coram and Bohner 2005, Nerur et al. 2005, Turk et al. 2005, Boehm 2006]; the following are some of the more commonly cited problems:
    a. Unrealistic assumptions (e.g. Scrum, as elaborated in Section 3.4.2.3)
    b. Lack of scalability
    c. Lack of a specific, unambiguous process (e.g. XP and Crystal, as elaborated in Section 3.4.2.3)
5. Seamless development, pioneered by seminal methodologies, is not adequately appreciated and supported in modern-day methodologies [Paige and Ostroff 2002].

Even though object-oriented software development methodologies suffer from various kinds of problems, they are still considered state of the art, and research aimed at improving them is an ongoing evolutionary process [Capretz 2003, Boehm and Turner 2004]. The status quo of the field clearly shows potential for improvement through addressing the abovementioned issues. There is motivation for developing methodologies that use the lessons learnt from UML and the long history of object-oriented methodologies in setting up a framework for software development that addresses the problem issues. The following have been observed by the author (based on personal experience) as general characteristics of such methodologies, highlighting the core areas where further work on OOSDMS is needed:

1. *Compactness*: an extensible core is preferable to a customisable monstrosity or a generic framework with complex parameters and/or prohibitively numerous parameter options.
2. *Extensibility*: with extension mechanisms and guidelines clearly defined.

3. *Traceability to requirements*: all the artefacts should be one way or another traceable to the requirements.
4. *Consistency*: artefacts produced should not be allowed to contradict each other; alternatively, there should be mechanisms for detecting inconsistencies.
5. *Testability of the artefacts from the start*: this will allow tools to be developed to verify and validate the artefacts.
6. *Tangibility of the artefacts*: artefacts should be concrete enough to be related to and understood by the parties involved in the development process.
7. *Visible rationality*: there should be evident rationality behind every task and the order in which the tasks are performed, and undeniable use for every artefact produced. The developers should be able to see this logic, truly sensing that any digression will put their objectives at risk.

1.1.2 The Software Engineering Approach to Methodology Development

Realizing the need and potentiality for further improvement in the field, it is important to point out that the relatively long history of methodology development is a rich source of lessons to be learned. In every methodology, there are features to exploit and pitfalls to avoid, many of which are direct or indirect consequences of the method used in developing the methodology or the circumstances surrounding the development. Choosing the right methodology to develop the desired methodology is therefore of utmost importance. Object-oriented methodologies can be categorized according to the circumstances leading to their development, including the approach and method applied (if any):

- *Revolutionary*: A large number of OOSDMs have been developed by experienced practitioners or academics trying novel ideas and approaches in their day to day engineering practices, ultimately resulting in a methodology offering a whole new approach, marking a watershed step in the history of software development methodologies. Such methodologies act as seeds, starting their own threads of evolution. Methodologies belonging to the first generation of OOSDMs are all revolutionary, as are the first few agile

methodologies (e.g. XP [Beck 1999] and FDD [Coad et al. 1999]). The advent of a revolutionary methodology in this sense does not necessarily indicate the occurrence of a Kuhnian revolution: pre-existing methodologies might co-exist with the new ones, in which case a new trend of evolution aiming at convergence is usually commenced.

- *Evolutionary*: methodologies in this category are based on existing ones. New ideas are always present in these methodologies, yet their dependence on ideas borrowed from existing methodologies is such that precludes their classification as revolutionary. This category has two subcategories, each of which spans a large number of OOSDMs:
  - *Extensions* are methodologies adding new features to an existing methodology. Later versions and complements of revolutionary and evolutionary methodologies belong to this category.
  - *Integrations* are essentially the result of consolidating ideas from two or more methodologies. Methodologists often throw in a few novel ideas, but the bulk of these methodologies consists of bits and pieces borrowed from existing methodologies. The important issues of compatibility and complementarity are of utmost importance: methodologists should ensure compatibility of the constituent parts, and that they actually complement each other in a meaningful way. Integrations are of three types:
    - *Merger*: creators of methodologies come together and agree on a merger of their methodologies. The integration is typically done through a design-by-committee procedure, and always results in complex and unwieldy monstrosities. Mergers are typically the result of corporate ambitions, specifically aimed at bringing together the user communities of the individual methodologies in a bid to impose the integrated methodology as a widely acclaimed standard. RUP [Jacobson et al. 1999, Kruchten 2003] and OPEN [Henderson-Sellers and Graham 1996, Graham et al. 1997] are examples of mergers.
    - *Ad hoc*: the methodologist uses ideas, typically from prominent OOSDMs, in order to assemble his methodology. The selection of methodology components is not based on pre-

planned, objective analysis of the features in existing methodologies; rather, features are scavenged from favourite methodologies in order to fill the needs of the methodologist. Fusion [Coleman et al. 1994] and Catalysis [D'Souza and Wills 1995] are good examples.

- *Engineered*: an objective, comprehensive analysis is performed in order to identify useful features in existing methodologies, as well as the requirements of the target methodology. Based on the analysis results, a methodology is developed and tested. The closest existing OOSDM to this category is the Hodge/Mock methodology [Hodge and Mock 1992]. The developers were not aiming for a general-purpose methodology, but rather one that would be especially suitable for use in a simulation and prototyping laboratory, and therefore have been rather too particular in their choice of methodologies analyzed. Furthermore, there is little trace of disciplined and clear-cut design, implementation and test activities performed in developing the methodology [Mock and Hodge 1992].

While emergence of yet other *revolutionary* OOSDMs is not out of the question, they are inherently unpredictable and unplanned in occurrence, and planning a research aimed at delivering revolutionary features is immensely risky. Evolutionary methodologies, on the other hand, show great potential for improvement, especially with the abundant merits of seminal methodologies mostly neglected during the integration era, not to mention the instability caused by the eccentric leanings of integrated methodologies and agile methods as the main contenders, which has in turn led to convergence attempts. Planned research aimed at ameliorating the status quo by attempting to develop an evolutionary methodology seems to be of acceptable risk. The question comes down to which type of evolutionary approach to methodology development is the most appropriate.

While not without merit, developing *extensions* to existing methodologies is too constraining, since any extensions made to a methodology have to be compatible

with the methodology itself. The methodologist therefore does not have a free hand in applying changes and modifications. Extensions made to agile methods are good examples: extensions are not to in any way hamper agility, which is certainly a task easier said than done.

Considering the motivations and the special circumstances surrounding methodology *mergers*, planning such a development is for the creators only, and even if it weren't, the prospect of developing yet another heavyweight methodology is not appealing.

Contaminated with favouritism and subjectivity, *ad hoc* integration is hardly appropriate as a scientific undertaking. Some previous instances have been quite successful, but limiting the scope of the components used to those favoured by the methodologists, because of previous personal experience or widespread acclaim, is far from objective, and almost certain to miss precious opportunities.

Engineering a methodology through integration is obviously the most appealing to software engineers, and the least prone to subjectivity. However, the methodology engineering approach intended in this context is different from that seen in *Method Engineering*: Method Engineering, originally defined as "The engineering discipline to design, construct, and adapt methods, techniques and tools for the development of information systems" [Brinkkemper 1996], has over the years become mainly restricted to *Situational Method Engineering* [Harmsen 1997], in which methodologies are constructed to fit the project situation at hand. Contrary to the methodology engineering approach intended here, Method Engineering does not address the requirements-based development of a *general* methodology, let alone one based on analyzing existing methodologies and aimed at alleviating their shortcomings and making utmost use of their strengths; typologies of Method Engineering approaches and techniques, listed in [Ralyté et al. 2003, Ralyté et al. 2004], are testimonies to this fact. Nevertheless, Method Engineering has inspired metamodel-based process composition in some object-oriented methodologies; the OPEN methodology [Henderson-Sellers and Graham 1996] is a prominent example.

Although it might seem that the direction of this discussion has been such as to justify the engineering approach via elimination of alternatives, yet the actual

intention has been to show the contrast between the engineering approach and other approaches previously tried. It is evident that a methodology is, after all, essentially a kind of software [Osterweil 1987, Osterweil 1997], and a software engineering approach to its development is therefore preferable. The applicability of the approach is even more evident when the huge amount of experience gained through the rather long history of OOSDMs is considered. The field is even more in need of objective analysis and disciplined engineering than before, since any other approach is bound to overlook the precious potentialities, not to mention the lurking hazards, in a field as overgrown and unkempt as object-oriented software development has become.

1.1.3 Research Roadmaps

Apart from the above-mentioned issues, there are several other key research pointers directly or indirectly related to software development processes and this thesis. Presented at the Conference on the Future of Software Engineering in 2000, these research pointers reflect the problems with the status quo of software engineering and its subfields, and set roadmaps of research for the coming years. Of the various research pointers proposed for the covered areas, the following are relevant to the present research, listed under their respective areas:

- **Software Process** (quoted from [Fuggetta 2000]):

    1. The scope of software [process] improvement methods and models should be widened in order to consider all the different factors affecting software development activities. We should reuse the experiences gained in other business domains and in organizational behaviour research.
    2. Statistics is not the only source of knowledge. We should also appreciate the value of qualitative observations.

- **Requirements Engineering** (quoted from [Nuseibeh and Easterbrook 2000]):

    1. Better modeling and analysis of problem domains, as opposed to the behaviour of software.

- **Object-oriented Modeling** (quoted from [Engels and Groenewegen 2000]):

    1. Development of means to compose and to refine complex structured models
    2. Identification of guidelines for an incremental, round-trip software development process

- **Software Engineering (general)** (quoted from [Finkelstein and Kramer 2000]):

    1. We need to devise and support new structuring schemes and methods for separating concerns in software systems development.
    2. We need to adapt conventional software engineering methods and techniques to work in evolutionary, rapid, extreme and other non-classical styles of software development.

1.2 Objectives and Scope

Motivated by the issues outlined above, the central proposition of this thesis can be summarized as follows:

> An object-oriented software development methodology can be developed (*engineered*) via a software engineering process – that is, through the generic development phases of *analysis*, *design*, *implementation* and *test* – based on analyzing existing methodologies and techniques, identifying their strengths and weaknesses, and producing a set of requirements defining the characteristics of the target methodology. The methodology can then be developed through making utmost use of existing techniques in such a way as to satisfy the requirements.

A further point to clarify is that object-oriented modeling is already saturated with diverse and versatile modeling methods and notations, especially with the advent of the UML and its widespread adoption as the de facto standard modeling language.

Therefore, developing a new modeling language is no longer an important concern in methodology development, and the focus of the development effort undertaken herein is on the process component of the methodology.

1.3 Research Methodology

As the research methodology used is essentially a software engineering process, the methodology used for developing the target object-oriented software development methodology broadly consists of the four generic phases of software engineering: *Analysis*, *Design*, *Implementation*, and *Test*. Due to the risk factor involved, commitment to a more concretely specific lifecycle and methodology could not be made; a *meta*-methodology for developing OOSDMs gradually took shape in the course of the effort and is indeed one of its main contributions. However, due to the need for appropriate measures for risk mitigation, a general iterative-incremental lifecycle was adopted, allowing for ventures into later phases – especially during design and implementation – in order to assess and mitigate development risks. As seen in Figure 1, the methodology is produced through iterations of the Design-Implementation-Test cycle based on the results of the Analysis phase. In addition to the verification and validation performed during the Test activity, requirements-based reviews of the produced methodology are also performed at the end of Design and Implementation activities in each of the iterations, ensuring an acceptable level of quality and maintaining the focus of the effort.

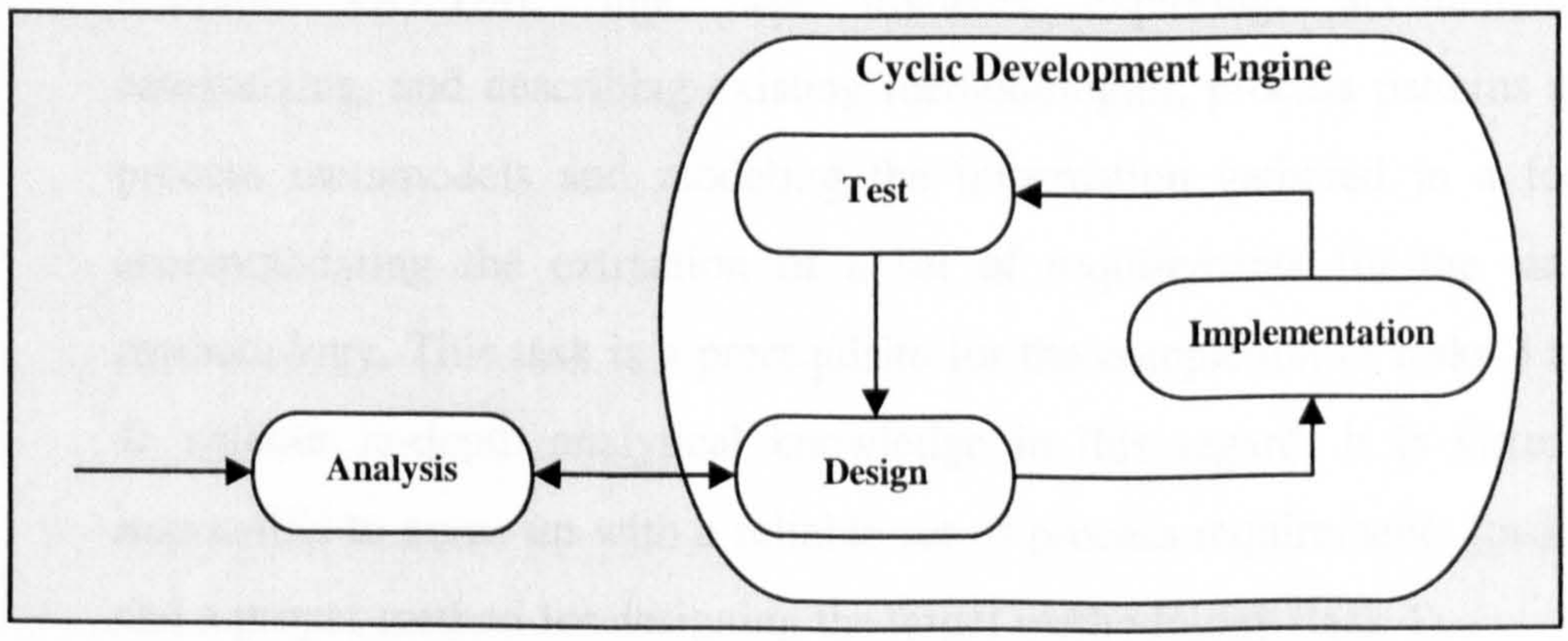

Figure 1. General methodology development lifecycle used in this thesis

High-level descriptions of the goals of the four generic phases and the tasks to be performed in each are given in the following subsections.

1.3.1 Analysis

Conforming to the specifications of the generic analysis phase, the goals and tasks of this phase focus on problem domain analysis and requirements elicitation.

Goals

- Identification and detailed analysis of the problem domain; the structure and behaviour of the problem domain should be modeled in order to abstract away redundant and irrelevant elements, enabling focus on features essential for identifying the requirements.
- Definition of a set of analysis criteria; the criteria will be used for analyzing the problem domain and producing the requirements.
- Determination of the scope of the target OOSDM, and delineating its requirements

Tasks

- **Task 1:** Research on the problem domain, encompassing existing methodologies, process patterns and process metamodels, which are the entities providing essential information as to the strengths and weaknesses of existing methodologies, in turn leading to the requirements of a desirable OOSDM; this task involves exploring, accumulating, categorizing, and describing existing methodologies, process patterns and process metamodels and modeling the information gathered in a form accommodating the extraction of a set of requirements for the target methodology. This task is a prerequisite for the completion of tasks 3 and 4: without in-depth analytical knowledge in this regard, it is virtually impossible to come up with a reliable set of process requirements (task 3) and a proper method for designing the target methodology (task 4).
- **Task 2:** Development of a criterion set for evaluating the methodologies, mainly in order to gain a better understanding of what is desirable, and what is undesirable, in an object-oriented software development

methodology; the criterion set will be used in task 3 for evaluating the problem domain entities described in task 1 (methodologies, process patterns, and process metamodels), and will ultimately be used as the basis for defining the requirements. The results of the evaluation performed in task 3 are in turn used for refining the criterion set; therefore, there is a two-way dependency between tasks 2 and 3, meaning that the two tasks should be carried out in parallel.

- **Task 3:** Development of a set of concrete requirements (based on the results of tasks 1 and 2), to be satisfied by the target methodology; this requires that a detailed analysis of the results of task 1 be first performed; and strengths and weaknesses of the methodologies, process patterns and process metamodels be identified using the criteria defined in task 2. The evaluation results and the evaluation criteria are ultimately used for defining the requirements.

1.3.2 Design

As expected, the design phase concerns producing a blueprint for the target methodology, to be implemented in the next phase as a detailed specification.

Goals

- Determination of a general process for the target OOSDM
- Production of a blueprint of the target methodology based on the general process defined and the requirements

Tasks

- **Task 4:** Determining the best method for designing the methodology based on the knowledge gained in task 1 and the requirements defined in task 3
- **Task 5:** Development of the design of the target methodology by applying the method selected in task 4; the design will include outlines of the phases, procedures, rules, techniques, tools, documentation and management issues, providing guidance as to the order of the activities, specifying what artefacts should be developed, and directing the tasks of the teams and individual developers.

1.3.3 Implementation

Analogous to the classic perception of the generic implementation phase, this phase produces the target methodology in a form usable by the users, i.e. system developers.

Goals

- Detailing the outline produced in the design phase
- Presenting the result in a form usable by potential users (developers)

Tasks

- **Task 6:** developing a user guide template for presenting the detailed specification of the methodology
- **Task 7:** producing detailed specifications of the target methodology's phases, procedures, rules, techniques, tools, and documentation and management issues, specifying detailed guidelines as to the order of the activities, the artefacts produced and the modeling language used, and the tasks of the team and individual developers; the user guide template defined in task 6 is extensively used in this task, practically guiding it through the detailed specification process by providing a structure to be filled in with the specifications produced.

1.3.4 Test

Testing a methodology is similar to testing any other type of system: develop test cases (in this case, sample systems), perform verification and validation, and correct the detected faults.

Goals

- Developing case studies for the target OOSDM to verify and validate the implemented methodology
- Testing the methodology by applying it to the sample systems, and debugging the detected failures

Tasks

- **Task 8**: Definition of realistic case studies in order to test the applicability of the produced methodology, and its conformance with the requirements; the domains to be covered are determined according to the scope and constraints imposed by the requirements.
- **Task 9**: Evaluation of the target methodology through the case studies developed in task 8, checking compliance with the requirements and the evaluation criteria; failures to comply are recorded and corrections made to the methodology.

1.4 Research Outcome

The following are the main results and contributions of the research reported herein:

1. A proposed object-oriented software development methodology addressing some of the problems found in existing methodologies; the following are the major contributions of this methodology:
   1.1. A model-based approach to the development of business systems integrating the agile feature-driven merits of the FDD methodology [Palmer and Felsing 2002] with design-based features of third-generation OOSDMs, particularly Catalysis [D'Souza and Wills 1998]; the methodology provides a middle way between integrated and agile methods, and addresses several key issues in OOSDMs. Defined as requirements and used as the basis for the development of the methodology, the most significant of these issues are: seamlessness, smoothness of transition, manageability of complexity, encouragement of active user involvement, practicability and practicality. Details of how the methodology conforms to the requirements are presented in Chapter 7.
   1.2. A modeling approach built into the methodology providing seamless and smooth transition from real-world models of the problem domain to system models, and ultimately to design

models; the model chain produced is based in the requirements, and traceability features have been incorporated. The approach features a novel technique for rectifying anomalies associated with real-world modeling [Isoda 2001]. The technique is based on designing the computer-based system initially as a homogeneous extension to the existing system structure (i.e. by using the same types of elements as those seen in the problem domain) and then applying pattern-based transformation to convert the models to software-system models. The technique also proposes the use of design patterns for introducing structure and behaviour into the system. In addition to seamlessness and smoothness of transition, the model chain also addresses key modeling requirements such as: testability, tangibility, manageability of complexity, and support for behavioural, structural and functional modeling of logical and physical views of the system at different levels of abstraction. Details of how the modeling approach conforms to the requirements are presented in Chapter 7.

2. A proposed methodology for developing object-oriented software development methodologies based on a software engineering approach; the following are the major contributions of this *meta-methodology*:
    - 2.1. An iterative-incremental lifecycle based on the generic activities of software development
    - 2.2. A process-centred template for describing OOSDMs; a total of 24 prominent object-oriented methodologies, process patterns and process metamodels have been described using this template, providing a rich process-centred review of the field.
    - 2.3. A criteria-based analysis method for identifying strengths and weaknesses in object-oriented methodologies, process patterns and process metamodels, ultimately producing a set of requirements for the target methodology; the method has been used for identifying strengths and weaknesses in the 24 methodologies, process patterns and process metamodels described using the process-centred template. Although mainly

used for the purpose of defining the requirements, the results are themselves a contribution of this thesis, since they provide an extensive critique of the research field.

2.4. An iterative-incremental requirements-based design method for producing the blueprint of the target methodology; the method has been designed in such a way as to provide flexible use of a multitude of design approaches.

2.5. A User Guide template for providing a pragmatic description of object-oriented software development methodologies; the template has been used for detailing and refining the target methodology, which in the context of the proposed meta-methodology, is analogous to *implementation* in software development.

Although many of the requirements of the methodology have been addressed in the final result, there remain requirements which have not been adequately met, and hence require further work. The most important of these requirements are: extensibility, configurability, flexibility, and support for formal modeling. Details of these shortcomings have been given in Chapter 7.

There are several potential courses for furthering or complementing the research reported in this dissertation, some of which are listed below:

- Engineering variants of the methodology targeting other types of systems, e.g. safety-critical
- Applying the methodology to case studies of larger scope
- Expressing the methodology and meta-methodology processes in a Process Modeling Language (PML) for static verification and/or enactment in a Process-centred Software Engineering Environment (PSEE) [Ambriola et al. 1997, Barthelmess 2003]
- Empirical analysis of the usability of the methodology
- Comparison of the methodology to other OOSDMs
- Application of the meta-methodology to the development of other methodology types

1.5 Overview of the Thesis

The chapter structure of this thesis is as follows:

- **Chapter 2 (*Background*)** presents the research background, delineating the relevant research areas and focusing on the evolution process leading to the status quo. Special attention has been given to object-oriented methodologies, process patterns and process metamodels, with *Method Engineering* explored and compared to the approach adopted in this thesis.
- **Chapter 3 (*Analysis*)** presents an explanation of the analysis process adopted in the Analysis phase, and reports the results. Template-based descriptions of a selection of methodologies, process patterns and process metamodels are presented, and the results of applying a criteria-based evaluation process to the selection are reported. A set of requirements for an object-oriented software development methodology is produced as a result, listed in the final section of this chapter.
- **Chapter 4 (*Design*)** presents an explanation of the iterative design process deployed in the design phase, and reports the results. The process is demonstrated by following the iterations through which the methodology is gradually formed. The resulting methodology design is then explained, with phases and tasks defined in outline.
- **Chapter 5 (*Implementation*)** presents an explanation of the implementation process and the user-guide template used for implementing the methodology. The major bulk of the chapter contains the implemented methodology, i.e. the resulting user guide providing detailed description of the methodology from three complementary viewpoints: *process-centred*, *work-product-centred*, and *role-centred*.
- **Chapter 6 (*Test*)** presents an explanation of the testing process, and reports the results of verifying and validating the implemented methodology through applying it to two sample information systems. The case studies are mainly focused on novel features of the methodology, since these features pose the greatest risk.
- **Chapter 7 (*Conclusion*)** presents a summary of the thesis and the results, discusses the degree to which the objectives were achieved, and examines

the shortcomings. Suggestions for furthering the research are also provided.

Chapter 2

Background

Although object oriented software development methodologies have become ubiquitous in software engineering circles, a brief look at the basic definitions and the history of their evolution is necessary for understanding the motivations behind this thesis, and the basis upon which it builds. A brief overview of the Method Engineering discipline is also presented, mainly in order to clarify the position of this thesis in regard to the discipline, and also to highlight the distinctions that separate this thesis from current Method Engineering practices.

2.1 Basic Definitions

A *Software Development Methodology (SDM)* is a framework for applying software engineering practices with the specific aim of providing the necessary means for developing software-intensive systems. Software development methodologies are therefore considered an integral part of the Software Engineering discipline, since methodologies provide the means for timely and orderly execution of the various finer grained techniques and methods of software engineering. Although a software development methodology can be loosely defined as "a recommended collection of phases, procedures, rules, techniques, tools, documentation, management, and training used to develop a system" [Avison and Fitzgerald 2003], it is easier to grasp when described as consisting of two main parts [OMG 2003]:

1. A set of modeling conventions comprising a *Modeling Language* (syntax and semantics).
2. A *Process*, which
    a. provides guidance as to the order of the activities,

b. specifies what artifacts should be developed using the *Modeling Language*,
c. directs the tasks of individual developers and the team as a whole, and
d. offers criteria for monitoring and measuring a project's products and activities.

Whereas the *modeling language* provides developers with a means to model the different aspects of the system, the *process* determines what activities should be carried out to develop the system, in what order, and how. In its most abstract form, a process is a sequence of steps – sometimes deprecatingly called a "recipe" – that aims to guide its users in applying the modeling language for accomplishing a set of software development tasks. The *process* thus acts as the dynamic, behavioural component of the methodology, governing the development (technical) and management subprocesses, and therefore encompassing the phases, procedures, rules, techniques, and tools prescribed by the methodology, as well as the issues pertaining to documentation and project management.

2.2 Object-Oriented Software Development Methodologies

An Object-Oriented Software Development Methodology (OOSDM) is specifically aimed at viewing, modeling and implementing the system as a collection of interacting objects, using specialized modeling languages, activities and techniques needed to address the specific issues of the object-oriented paradigm. Originally based on concepts introduced in system simulation, operating systems, data abstraction, and artificial intelligence, the object-oriented paradigm gained widespread popularity in the 1980s through object-oriented programming languages. The applicability of the object-oriented approach to systems analysis and design was recognized in the mid 1980s, and the subsequent enthusiasm has been such that a plethora of object-oriented software development methodologies have been since introduced. A brief description of the categories of OOSDMs and their trend of evolution will help further clarify the domain.

2.2.1 Seminal Methodologies: First and Second Generations

The first software development methodologies termed as object-oriented were in fact hybrid: partly structured and partly object-oriented. The analysis phase was typically done using Structured Analysis (SA) techniques, producing Data Flow Diagrams, Entity-Relationship Diagrams, and State Transition Diagrams, whereas the design phase was mainly concerned with mapping analysis results to an object-oriented blueprint of the software. These methods were hence categorized as *transformative* [Monarchi and Puhr 1992]. The methods prescribed by [Seidewitz and Stark 1986] and [Alabiso 1988] are the main methodologies in this category.

The first purely object-oriented methodologies appeared in 1986 [Booch 1986], and were influenced by structured and/or data-oriented approaches. This first generation of object-oriented methodologies spans methodologies developed between 1986 and 1992. The second generation of object-oriented methodologies evolved from the first generation and appeared between 1992 and 1996. This period signifies the famous "Methodology War", with more than 70 methodologies competing for a share in the software development industry. The sheer number of methodologies introduced became so prohibitive that choosing the right methodology for a software project was a major endeavour in itself. The frustration in the software engineering community soon led to efforts aimed at integration and unification, the first fruit of which was the Unified Modeling Language (UML), adopted by the Object Management Group (OMG) as the standard object oriented modeling language in 1997 [Booch et al. 1999, OMG 2004]. While UML was being developed, widespread attempts at integrating seminal methodologies were also being made, thus signifying the end of the second-generation era.

First- and second-generation methodologies are collectively referred to as "Seminal" methodologies, in that they pioneered the unexplored field of pure object-oriented analysis and design, and in doing so laid the groundwork for further evolution. Though by no means mature, the ideas set forth by these methodologies have deeply influenced the fast-growing field of object-oriented software engineering. Many of the concepts, modeling conventions and techniques introduced by these methodologies are still widely used today, and some of these

methodologies still have hosts of devoted followers, proving that seminal methodologies are by no means obsolete.

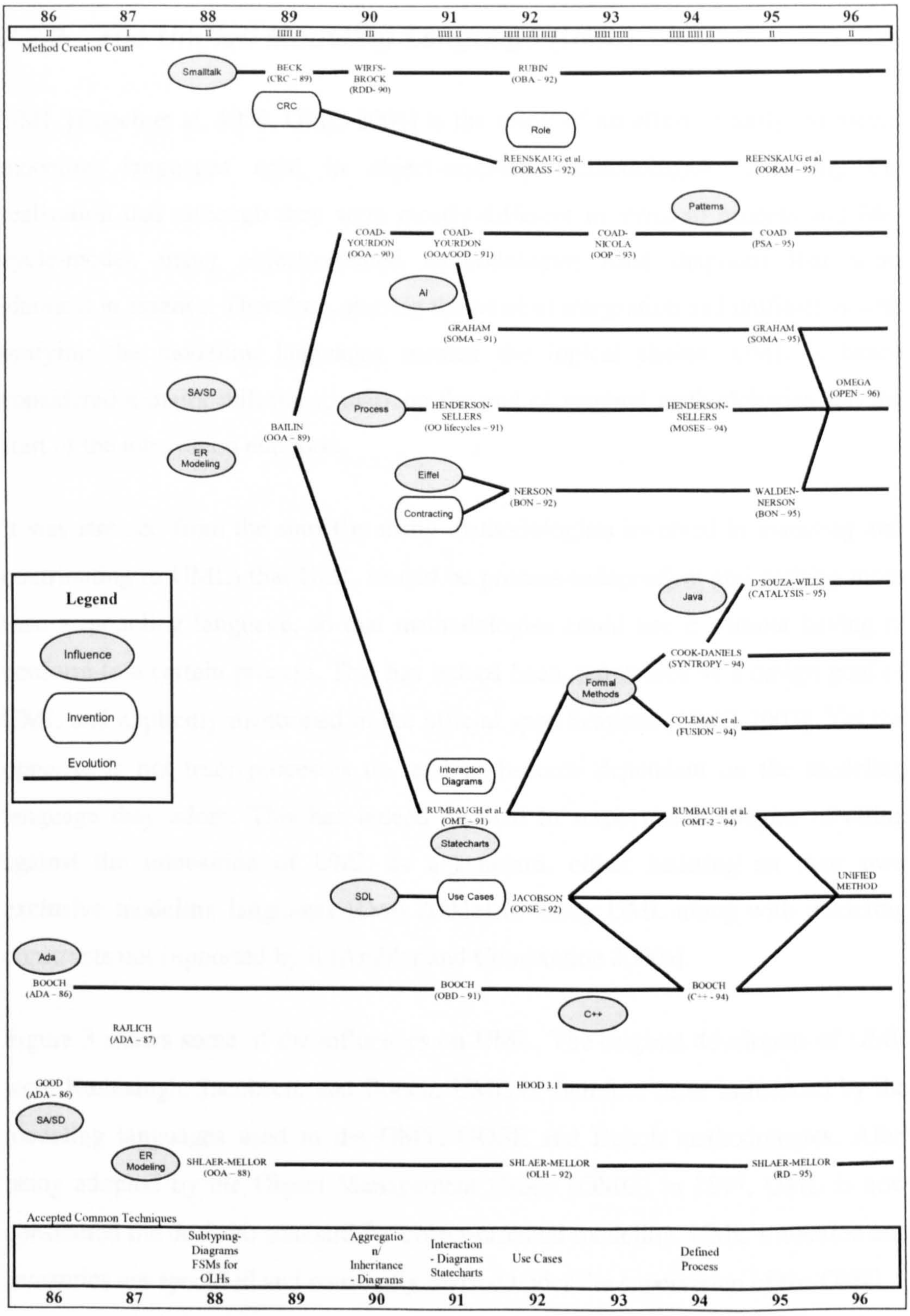

Figure 2. The evolution timeline of object-oriented methodologies up to 1996 – adapted from [Webster 1996]

Figure 2 shows the methodologies developed during the period from 1986 to 1996 [Webster 1996]. It also shows the evolution timeline and genealogical relationships between the methodologies, emphasizing the influences and the contributions.

2.2.2 The Unified Modeling Language (UML)

UML [Booch et al. 1999, OMG 2004] is the result of an effort to unify the visual modeling languages used in object-oriented methodologies, following the realization that although they were mostly different in terms of process and life-cycle-model, many object-oriented methodologies used diagrams that were identical in essence. Therefore, starting the trend of integration and unification with unifying the modeling languages seemed the logical choice. UML is hence considered a major milestone, marking the end of seminal methodologies and the start of the integration euphoria.

It was stressed from the start (by many methodologists involved in assessing and contributing to UML) that UML should be process-independent and nothing more than a modeling language, so that methodologies could use it without having to conform to a certain process. This has indeed been maintained as a design goal of UML and explicitly mentioned in the official specifications [OMG 2003]. Yet the opposite is not true: processes do tend to become dependent on the modeling language they adopt. This has indeed resulted in some methodologies rebelling against the imposition of UML as a standard, either insisting on their own exclusive modeling languages [Dori 2002a] or using UML along with modeling constructs not supported by it [Ambler and Constantine 2000a].

Figure 3 shows some of the influences on UML. The original developers of UML were Rumbaugh, Jacobson, and Booch; UML is therefore most influenced by the modeling languages used in the OMT, OOSE and Booch methodologies. After being adopted by the Object Management Group (OMG) in 1997, UML is now considered the de facto standard for object-oriented modeling. UML's notation and semantics are specified and constantly revised under the supervision of the OMG.

2.2.3 Integrated Methodologies: Third Generation

Methodologies in this category are results of integrating seminal methodologies and are characterized by their process-centred attitude towards software development, typically targeting a vast variety of software development applications. Integrations have resulted in huge monstrosities of methodologies, difficult to manage and enact [Boehm and Turner 2004]. In trying to achieve manageability, some of them have gone to extreme measures to ensure customizability (RUP), others have turned into generic process Frameworks that should be instantiated to yield a process (OPEN), and yet others have resorted to process patterns for customizability (Catalysis); yet, it was frustration with these methodologies that ultimately caused the agile movement [Highsmith 2000b]. Although unwieldy and complex, integrated methodologies have a lot to offer in terms of process components, patterns, and management and measurement issues. Furthermore, some of them propose useful ideas on seamless development, complexity management and modeling approach.

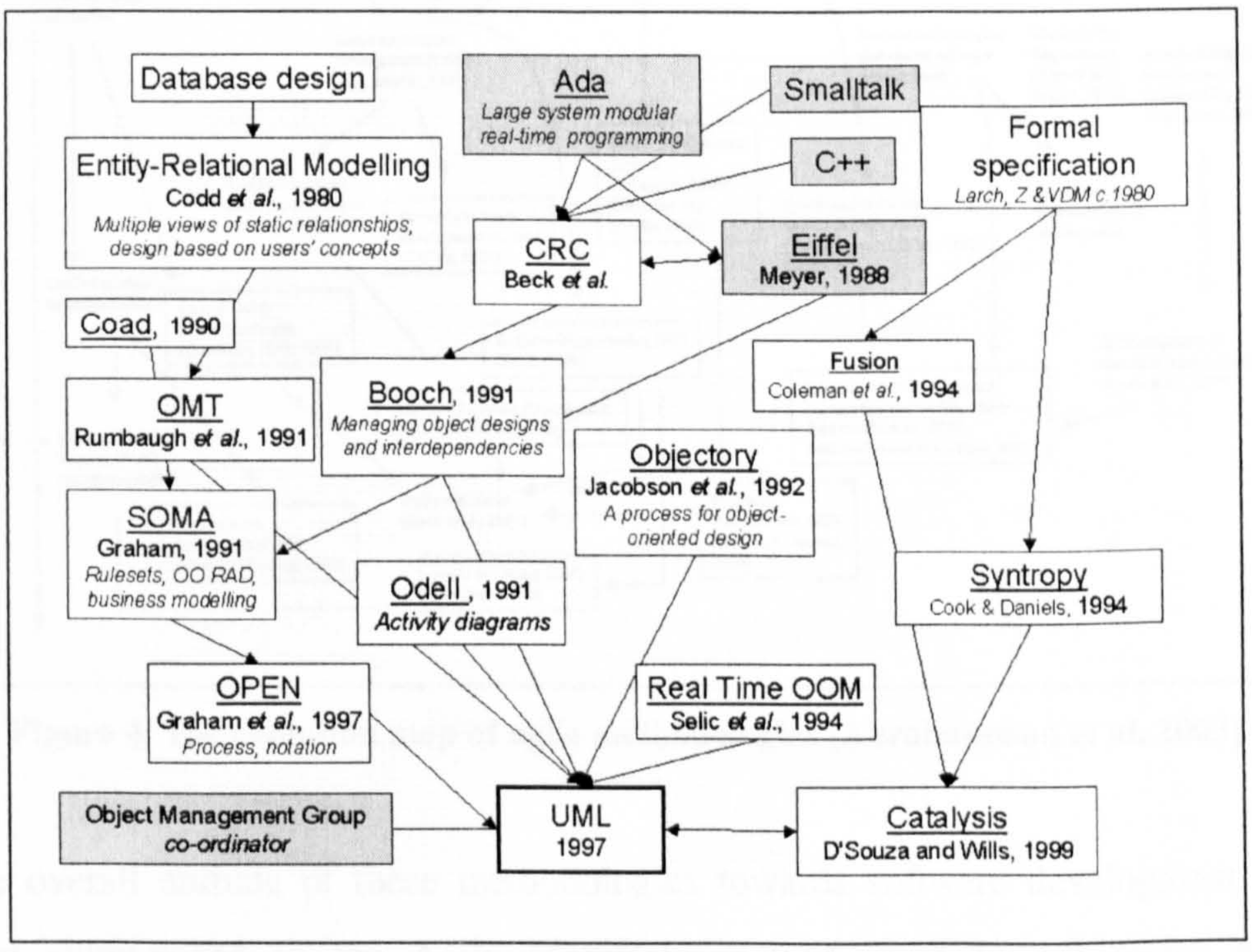

Figure 3. Influences on UML [Graham 2001]

2.2.4 Agile Methodologies

Agile methodologies first appeared in 1995 [Highsmith 2002, Abrahamsson et al. 2002, Schuh 2005]. The once-common perception that agile methodologies are nothing but controlled code-&-fix approaches, with little or no sign of a clear-cut process, is only true of a small – albeit influential – minority of these methodologies, which are essentially based on practices of program design, coding and testing that are believed to enhance software development flexibility and productivity. Most agile methodologies incorporate explicit processes, although striving to keep them as lightweight as possible. Figure 4 shows an evolution map for a number of these methodologies, emphasizing the ways previous methodologies and practices have influenced them.

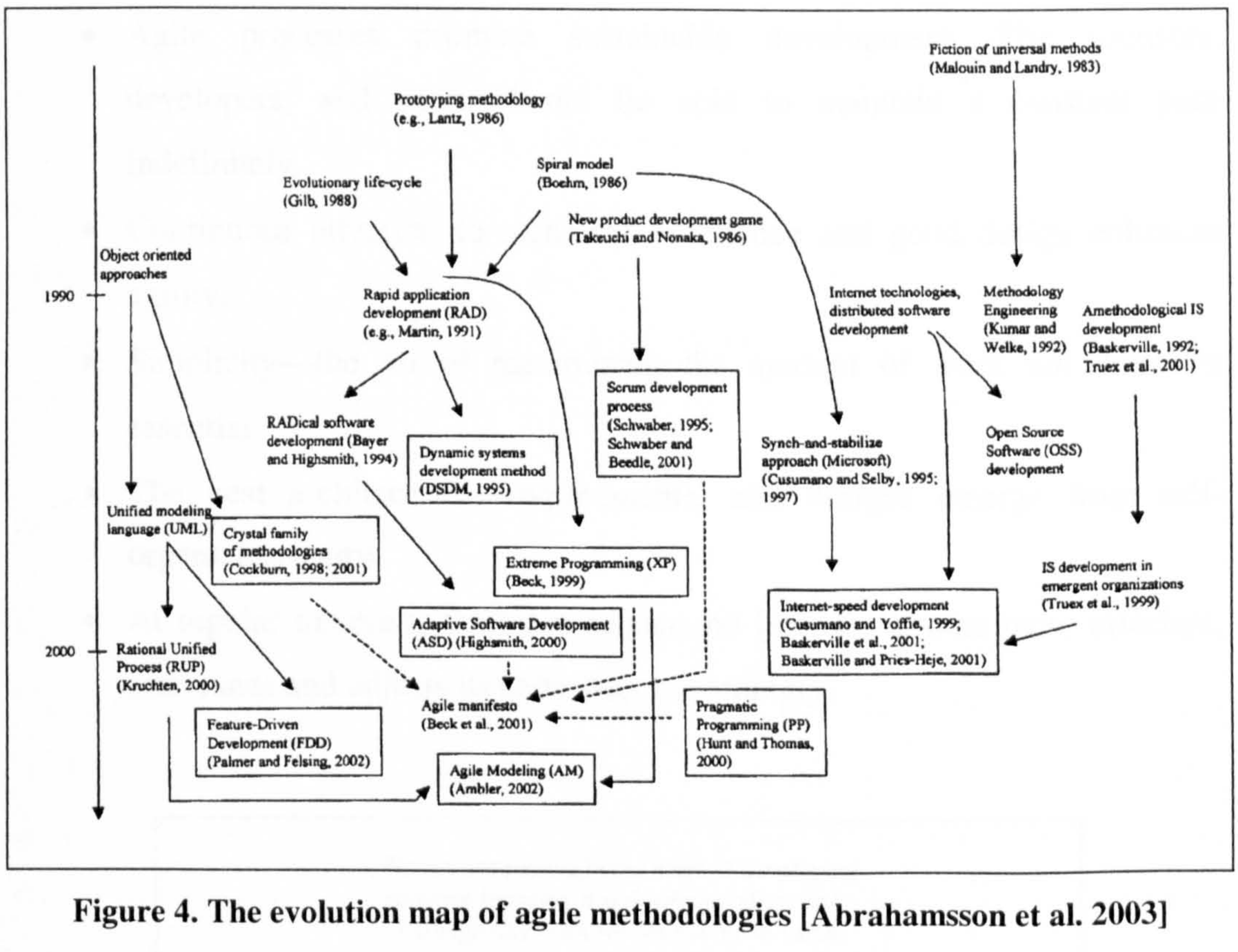

Figure 4. The evolution map of agile methodologies [Abrahamsson et al. 2003]

The overall attitude of these methodologies towards software development has been summarized in the Agile Manifesto, agreed upon by all major agile methodologists (Figure 5). Agile methodologists have also given a set of principles for agile development (quoted from [Beck et al. 2001]):

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Figure 5. The Agile Manifesto [Beck et al. 2001]

Although many agile methodologists claim that their methodologies are not process-centred, close examination usually reveals some sort of iterative-incremental process (sometimes quite elaborate). Whereas at the start of the agile movement words like "process" (even "methodology") were considered "dirty", agile methodologists are showing increasing interest in advertising their "agile" processes and methodologies [Schwaber and Beedle 2001, Schuh 2005].

2.3 Object-Oriented Process Patterns and Process Metamodels

The advent of UML has allowed methodologists to focus on processes instead of concerning themselves with devising new modeling languages, and the experience gained from the relatively long and adventurous history of OOSDMs has helped methodologists identify patterns and generalities among processes. Object-oriented process patterns are the results of applying abstraction to process components, thereby presenting ways for developing methodologies through composition of appropriate pattern instances [Ambler 1998a,b]. Object-oriented process metamodels, on the other hand, are the results of applying abstraction on the overall process, providing process generalizations, or metamodels; processes can then be built through instantiation of these metamodels [OMG 2002].

2.4 Method Engineering

Motivated by the prevalent belief that no one methodology fits all situations, *Methodology Engineering* was first introduced as a discipline aimed at constructing methodologies to match given organizational settings or specific development projects [Kumar and Welke 1992]. The discipline later came to be known as *Method Engineering*, a term proposed in [Brinkkemper 1996], with the definition broadened as: "The engineering discipline to design, construct, and adapt methods, techniques and tools for the development of information systems". The most well-known subfield of the discipline is *Situational* Method Engineering, which is concerned with the construction/adaptation of a methodology specifically attuned to the project at hand [Harmsen 1997].

There are several approaches to method engineering, the most prominent of which can be classified as follows [Ralyté et al. 2003, Ralyté et al. 2004]:

- *Ad-hoc*: Concerned with constructing a new methodology from scratch
- *Paradigm-based*: Concerned with instantiating, abstracting or adapting an existing meta-model in order to produce the target methodology
- *Extension-based*: Concerned with enhancing an existing methodology with new concepts and properties
- *Assembly-based*: Concerned with constructing the target methodology or enhancing an existing methodology through reusing parts of other methodologies.

Assembly-based method engineering is the foremost approach among the four listed above, and is also the main approach to situational method engineering. The assembly-based approach makes use of methodology components – called *method fragments* or *method chunks* – extracted from existing methodologies and stored in a repository. Assembly-based Method engineering has also inspired the use of process components in object-oriented software development, mainly through the OPEN methodology, as explained in Chapter 3 [Henderson-Sellers 2003, OPEN Consortium 2000].

The broad definition of method engineering means that the research reported in this dissertation can be categorized as belonging to this field; however, there are features in the approach adopted in this thesis which are either improvements to current method engineering practices, or set this thesis squarely apart from current trends of method engineering practice and research. The most important of these features are listed below:

- Whereas current practice and research related to method engineering is mainly focused on developing situational solutions, this thesis is concerned with developing a general methodology core.
- The *software engineering* approach adopted in this thesis, and the iterative-incremental methodology development lifecycle devised, are different from their sequential, non-design-based counterparts in method engineering [Ralyté et al. 2003, Ralyté et al. 2004].

- Requirements-based development is not new in method engineering, where requirements are defined according to the situation at hand; in this thesis, on the other hand, requirements are specified through analyzing existing methodologies.
- A hybrid design approach has been devised and applied in this thesis, which provides a framework allowing flexible application of four methodology development approaches, two of which – i.e. *Instantiation* and *Composition* – are analogous to the *Paradigm-based* and *Assembly-based* approaches of method engineering, but the remaining two – i.e. *Integration* and *Artefact-oriented* – are relatively novel in this context. The *Integration* approach is particularly nonconformist in comparison to usual method engineering practices, in that it promotes integrating ideas and techniques directly from existing methodologies, instead of first dissecting the methodologies into fragments (as is common practice in assembly-based method engineering, where a fragment repository is used). The motivation behind this approach is the author's personal observation that methodologies are synergistic entities, and while using repositories of process fragments is not without merit – and is indeed one of the constituent methods of the hybrid design approach adopted in this thesis – breaking down the methodologies into fragments may result in loss of functional capacity.

Chapter 3

Analysis

As in any engineering project, an effort aiming at developing software development methodologies should start by clearly defining what the requirements of such a methodology are. However, eliciting the requirements from a problem domain as vast, varied and controversial as object-oriented software development methodologies is by no means straightforward. The following are some of the problems facing such an effort:

- Methodologies are products; many are even marketed as such (e.g. RUP [Kruchten 2003] and DSDM [DSDM Consortium 2003]). Treating methodologies as merchandise frequently results in redundant decorative clutter, attractive yet obscuring wrappings, and uninformative, sometimes even advert-like, descriptions.
- Methodologies are complex. Even methodologists that try to be scientific and professional in their approach to defining their processes, too often end up giving too little or too much detail at the wrong level. OPM is an example, as elaborated later in this chapter.
- Methodologists are not objective and impartial towards their own creations (and should not be expected to). Features stressed by methodologists are most often not the essential ones for solving the problems of the domain, but those that the methodologist sees as important or unique.

Therefore, requirements elicitation in this thesis called for a concentrated effort aimed at gathering essential information about methodologies, process patterns and process metamodels (entities of the problem domain) through abstracting away the irrelevant features and laying bare the core philosophy and process.

The analysis method used in this thesis starts with summarizing problem domain entities (methodologies, process patterns and process metamodels) using a template accentuating the development processes that they offer. Analysis then proceeds with defining a set of criteria for analyzing the object-oriented software development processes thus highlighted. The criteria can be enriched by the analysis results along the way, and since they underline the strengths and weaknesses of software development processes, the final set of criteria is ultimately used for defining the requirements.

3.1 Analysis Approach

The merits of criteria-based analysis as a source of insight into the capabilities and shortcomings of software development methodologies has long been recognized, as shown in previous research on software development methodologies in general [Karam and Casselman 1993] and *object-oriented* software development methodologies in particular [Walker 1992, Monarchi and Puhr 1992, Abrahamsson et al. 2003]. The results obtained from such analyses are prevalently used for selecting, tailoring and effective usage of methodologies, but they can also be used for other purposes, as suggested in this thesis. The main problem that any researcher attempting to exercise such analyses faces is the definition of a suitable set of criteria.

An iterative-incremental approach to criteria-based analysis of software development methodologies was devised for the purposes of this thesis, using the analysis results themselves for refining the criteria. The method is based on the observation that the strengths and weaknesses of methodologies (identified through analysis) provide further ideas as to what is and what is not desirable in methodologies; this can in turn lead to the identification of new criteria and/or refinements to the existing ones. This means that a set of criteria can be built recursively (i.e. through iterative application of the criterion set to methodologies), starting from an initially unpolished and incomplete set of criteria. A method can thus be devised to incrementally build an initially incomplete set of analysis criteria through their iterative application to software development methodologies, until the criteria and the analysis results are stabilized (i.e. the iterations no longer have a significant effect on the criteria and the analysis results).

The analysis results are one of the main contributions of this process, yet it is the final criterion set that will provide the ultimate objective: a set of requirements for the target object-oriented software development methodology. This will be achieved by evolving each criterion into a requirement through adding the level of support that the target methodology is expected to provide for that criterion, taking into account the lessons learnt from existing methodologies (as inferred from the analysis results).

It should be noted that this approach is possible because of the relatively long history of software development methodologies, especially object-oriented ones, during which many development problems have been encountered and addressed [Graham 2001, Capretz 2003, Ramsin and Paige 2004]. The degree of maturity enjoyed today by methodologies is the main enabling factor for this approach, since it relies on the methodologies themselves for providing the criteria and the requirements.

The following sections contain a highlight of the method and the results of its application to the problem domain. The analysis results are reported along with the final criterion set, which are ultimately used for defining a set of requirements for the target OOSDM.

3.2 Analysis Process

The method consists of the following steps, during which the criterion set and the analysis results are incrementally built, and the final criteria are turned into requirements (Figure 6):

1. Selection of a set of software development methodologies to be analyzed; since the richness of the reviews and the analysis results is of utmost importance when defining the requirements, the set of methodologies should be comprehensive enough to provide extensive coverage of major features offered by object oriented methodologies. Therefore, a set of object-oriented process patterns and process metamodels were also included in the review and analysis.
2. Summarization and review of the selected methodologies and process patterns/metamodels using a process-centred template, abstracting away

the less important features of the methodologies and accentuating their core processes

3. Definition of an initial set of criteria to act as the seed for the iterative-incremental stage of the process; the criteria should be such that their application to methodologies triggers wider and deeper exploration of the methodologies (process and modeling language), giving rise to new criteria and/or refinements to the existing ones.
4. Iteration of the following steps during which the analysis results are incrementally built, and the criterion set is gradually refined; the cycle is repeated until the analysis results and the criterion set are stabilized:
    4.1. Analyzing the selected methodologies based on the criterion set, determining their significant strengths and weaknesses; the criteria are used as focus-pointers, concentrating the analysis on areas where significant strengths and weaknesses are most likely to be found.
    4.2. Updating the criterion set with new criteria and/or refinements to existing criteria or their structure, using the analysis results as a resource
5. Using the stabilized criterion set as a framework and detailing it using the analysis results in order to obtain the requirements

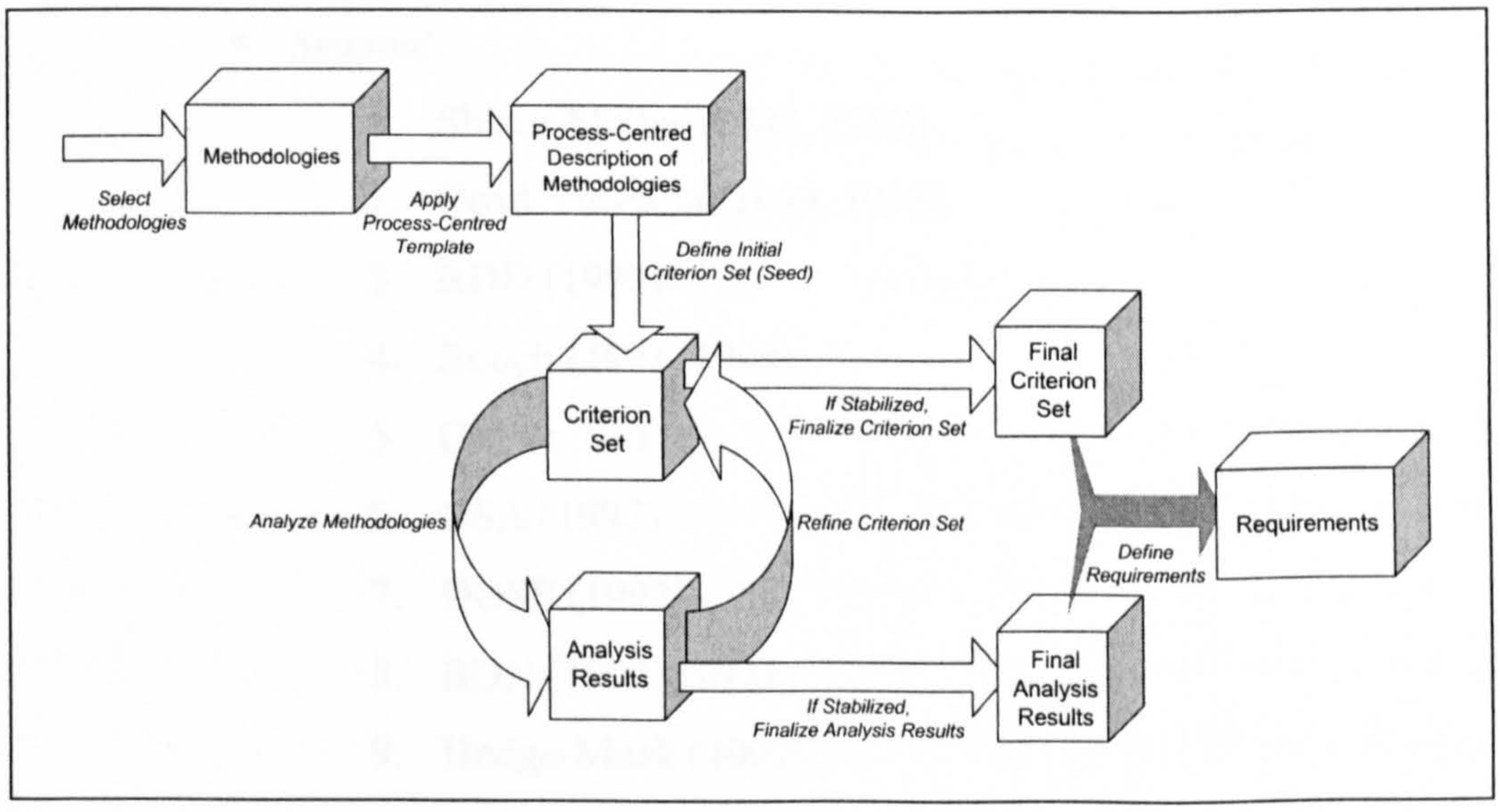

Figure 6. The analysis process

It is advisable to explore the methodologies as extensively as possible (i.e. beyond the scope of the initial criterion set) during the first iteration. This is not absolutely necessary, but will definitely prove a valuable time-saving measure, compensating for the deficiencies of the initial criterion set. The following sections report the results of applying this method to a set of object oriented software development methodologies.

3.3 **Process-Centred Review**

A total of 24 object oriented methodologies were chosen for analysis, spanning all the three classes of object-oriented methodologies: *seminal*, *integrated* and *agile*. A set of process patterns and process metamodels were also added in order to complement the set of methodologies, thereby enriching the feature set to be used in defining the requirements. It should be noted that object oriented software development approaches which lack a detailed process or a reasonably defined process metamodel have not been considered for inclusion in this review. MDA is the most important of these: still in its infancy, the approach is attractive as a development philosophy, yet is vaguely defined, and its practicability remains to be tested. The following are the methodologies, process patterns and process metamodels that were reviewed:

- **Methodologies**
  - *Seminal*
    1. Shlaer-Mellor (1988, 1992)
    2. Coad-Yourdon (1989, 1991)
    3. RDD (1990)
    4. Booch (1991, 1994)
    5. OMT (1991)
    6. OSA (1992)
    7. OOSE (1992)
    8. BON (1992, 1995)
    9. Hodge-Mock (1992)
    10. Syntropy (1994)
    11. Fusion (1994)

- *Heavyweight*
    1. OPM (1995, 2002)
    2. Catalysis (1995, 1998)
    3. OPEN (1996)
    4. RUP/USDP (1998, 1999, 2000, 2003)
    5. EUP (2000, 2005)
    6. FOOM (2001)
- *Agile*
    1. DSDM (1995, 2003)
    2. Scrum (1995, 2001)
    3. XP (1996, 2004)
    4. ASD (1997, 2000)
    5. dX (1998)
    6. Crystal (1998, 2004)
    7. FDD (1999, 2002)

♦ **Process Patterns**

1. Ambler (1998)

♦ **Process Metamodels**

1. OPF – as part of the OPEN methodology (2001)
2. SPEM (2002)

This section contains an overview of the process-centred description template, along with descriptions of the selected set of object-oriented methodologies, process patterns and process metamodels, summarized using the template.

3.3.1 Process-Centred Description Template

The selected methodologies are summarized and reviewed using a process-centred template, highlighting the activities prescribed in each methodology while keeping the description and discussion of the artefacts produced and modeling languages used (mainly diagrams and tables) as secondary to the activities. The description produced using this template offers little critique on the methodologies – and indeed that is not the goal – yet abstracts and structures them in a way that enables elaborate analysis of individual methodologies. The description of a methodology based on this template consists of the following parts:

1. An introductory preface providing a brief account of the methodology's history and distinguishing features, as well as an abstract overview of the development process prescribed by the methodology.
2. A number of subsections, one for each high-level subprocess in the methodology's development process, each consisting of:
    a. Details of the activities performed in the subprocess and the order in which they are performed.
    b. A concise description of the artefacts produced and the modeling languages used in the subprocess, described as part of their relevant activities; modeling languages, although necessary for fully understanding the mechanisms used in a methodology's process, tend to clutter the description of a methodology and obscure the process. Describing the modeling language as secondary to the process alleviates this problem. Scrutinizing notations, however, is beyond the scope of this analysis; notational conventions have therefore been left out of the descriptions in this thesis. The reader is referred to [Ramsin and Paige 2004] for the full descriptions.

3.3.2 Methodologies: Seminal

Due to the large number of these methodologies, only those most renowned and influential have been examined; methodologies that, according to the evolution timeline of [Webster 1996], either have started, or are apt representatives of, individual branches. Each methodology utilizes its own modeling language, which should also be covered if the description of the methodology is to be of any good.

3.3.2.1 Shlaer-Mellor (1988, 1992)

The Shlaer-Mellor methodology for object-oriented analysis and design was introduced through two separate books. In their first book [Shlaer and Mellor 1988], Shlaer and Mellor focused on analysis, leaving design to their second book [Shlaer and Mellor 1992]. Their analysis method considered objects as data entities rather than encapsulations of both data and behaviour, thus neglecting object methods. Therefore, it was mainly considered an information modeling method,

rather than a full-fledged object-oriented methodology [Coad and Yourdon 1991a]. The introduction of the design method and later enhancements turned this initially inadequate method into a competitive methodology [Shlaer and Mellor 1996]. The final version of the process covers the analysis, design, and implementation phases of the software development lifecycle. It can be broken down into eight steps (typically performed sequentially):

1. Partitioning the system into domains according to the four domain types defined in the methodology. The partitions practically divide the structure, functionality and behaviour of the software system into four tiers: problem domain, application-independent services, physical architecture, and physical implementation.
2. Analyzing the application (problem) domain.
3. Confirming the analysis through static and dynamic verification.
4. Extracting the requirements for the application-independent service domains supporting the application domain.
5. Analyzing the service domains.
6. Specifying the components of the architectural domain (physical configuration of the software).
7. Building the architectural components.
8. Translating (implementing) the analysis models of relevant domains into the architectural components.

These steps are briefly described in the following sections.

Partitioning the system into domains (Shlaer-Mellor)

The system is first partitioned into a number of *domains*. There are four types of domains, one or more of which (according to the following list) are defined in every system:

- An *Application Domain*: the domain specifically pertinent to the end user (problem domain).
- A number of *Service Domains*: relatively general, application-independent domains directly supporting the application domain; examples include the user interface and the sensors-and-actuators domain in real-time systems.

- An *Architectural Domain*: depicting the physical software configuration of the system and concerned with the organization of data, control and algorithm within the system as a whole.
- A number of *Implementation Domains*: comprising the readily available, implementation-level components supporting the software system at runtime; e.g. the operating system and the programming language constructs and components.

The domains are organized in client-server relationships, with the client domains depending on the server domains to provide them with necessary services. The results of this step are modeled in a *Domain Chart*, depicting the domains and their client-to-server relationships (referred to as *bridges*).

Analyzing the application domain (Shlaer-Mellor)

The next step involves applying Shlaer-Mellor OOA (Object-Oriented Analysis) to the application domain. Shlaer-Mellor OOA models are made up of three separate parts, built in the following order:

1. An *Object Information Model* is built that defines the objects of the domain, and the relationships between them.
2. *State Models* are built that show the lifecycle (behaviour) of each object.
3. *Action Specification Models* are built that depict the processing taking place in the state models. Usually there is one action specification for each state in each object's lifecycle. Action specifications are usually done in *Action Data Flow Diagrams* (ADFD).

If a domain is too large to be analyzed as a unit, it may be necessary to partition it into subsystems. Three models are constructed to show relationships between subsystems within a domain. These models are:

1. *Subsystem Relationship Model*: showing the structural relationships between objects in different subsystems.
2. *Subsystem Communication Model*: showing event communications between objects in different subsystems.

3. *Subsystem Access Model*: showing data accesses between objects in different subsystems.

The methodology also prescribes the production of a number of *derived models* for each of the subsystems, as listed below:

1. *Object Communication Model*: showing the event communications between objects.
2. *Event List*: showing events being sent within or between state models.
3. *Object Access Model*: showing the data accesses between objects.
4. *State Process Table*: showing the processes in all ADFDs.
5. *Thread-of-Control Chart*: showing the sequences of actions executed in response to each and every external event.

Confirming the analysis (Shlaer-Mellor)

A set of rules, described in [Lang 1993], forms the basis for static verification of the OOA model-set. Furthermore, a process is prescribed for dynamic verification of the model-set by simulating the execution of the models. The simulation of a desired behaviour is done in four steps:

1. Establish the desired initial state of the system in data values in the object information model.
2. Initiate the desired behaviour with an event sent to a state model.
3. Execute the processing as specified by the action specification models and as sequenced by the state models.
4. Evaluate the outcome against the expected results (according to the desired behaviour).

Extracting the requirements for the service domains (Shlaer-Mellor)

In the domain chart, each bridge between domains represents what the client domain requires of the server domain. These requirements are assigned to the service domains and form the basis for analyzing the remaining domains in the system.

Analyzing the service domains (Shlaer-Mellor)

After specifying the requirements of all the client domains, Shlaer-Mellor OOA is applied to each of the remaining service domains. After analyzing each domain, its behaviour is dynamically verified by executing its OOA models. This process continues downwards along the bridges until domains are reached that either belong to the system-wide architecture (i.e. the architectural domain) or already exist (implementation domains such as the operating system, the programming language or the communication network).

Specifying the components of the architectural domain (Shlaer-Mellor)

As the last domain to be analyzed, the architectural domain is mainly concerned with system design issues and specifies generic, system-wide components for managing data, function and control, thereby laying out the physical configuration of the system and defining rules for translating the OOA models into this configuration.

The architectural domain is specified in two types of components: *mechanisms* and *structures*. *Mechanisms* represent architecture-specific capabilities that must be provided in order to realize the system and are realized as traditional software tasks and library components. A mechanism may be regarded as the actual code that can be linked into the final system to implement elements of the models (state machines, event receiving queues, etc.). *Structures* represent a prescription for translating the OOA models of the client domains. They are realized as templates for code fragments that are filled in (populated) based on elements in the OOA model (e.g. archetypes for C++ classes populated from OOA model objects).

The architectural domain can be designed using Shlaer-Mellor OOA notations, but may also be designed using other methods. If an *object-oriented* design is required, Shlaer/Mellor's Object Oriented Design LanguagE (OODLE) is recommended. OODLE uses four types of diagrams (arranged into a layered structure) to model the design of an object oriented program, library or environment:

- *Inheritance Diagrams*, which show the inheritance relationships between classes.

- *Dependency Diagrams*, showing usage (client/server) and friend relationships between classes.
- *Class Diagrams*, which show the external view of each class.
- *Class Structure Charts*, showing the structure of the methods and the flow of data and control within a class.

Building the architectural components (Shlaer-Mellor)

Mechanisms and structures indicating the physical design of the system (specified in the previous task) are detailed and set up in this task. Architectural *mechanisms* realizing the system-wide data management, functionality and control are constructed, and architectural *structures* are detailed in order to define unambiguous templates for adding the functionality of the client domains to the mechanisms. The stage is thus set for the implementation of components pertaining to the client domains; these will be constructed and embedded into the architecture in the next task. As this task and the next deal with implementation issues, extensive use is made of components and constructs provided by the implementation domain.

Translating the models of each domain (Shlaer-Mellor)

Models pertinent to those domains that are direct or indirect clients of the architectural domain are implemented into the architectural configuration using the structures detailed in the previous task. The details of the final step depend a great deal on the design chosen for the system, and the architectural components created. For example, considering the general case of multitasking and multiprocessor systems, the essential activities would be to:

1. Allocate instances of objects to tasks, and tasks to processors.
2. Create the tasks through translating the OOA models.

3.3.2.2 Coad-Yourdon (1989, 1991)

Like many other early object-oriented methodologies, the Coad-Yourdon Methodology had a two-phase introduction. Coad and Yourdon introduced their Object-Oriented Analysis (OOA) method in 1989. A second edition of their book

on analysis appeared in 1991 [Coad and Yourdon 1991a], and their landmark book on Object-Oriented Design (OOD) was published the same year [Coad and Yourdon 1991b]. The Coad-Yourdon Methodology is comparatively simplistic in its approach, yet it served its purpose as an introductory object-oriented methodology at a time when inertia in adopting object-oriented techniques seemed too great to overcome. Although the Coad-Yourdon methodology is generally considered to only span the generic analysis and design phases, it does offer guidelines for implementation, by suggesting techniques for translating the design models into code. The general process for applying the analysis and design methods is shown in Figure 7 (called the "Baseball Model"). The activities and deliverables of OOA and OOD as prescribed by the Coad-Yourdon methodology are covered in the next sections.

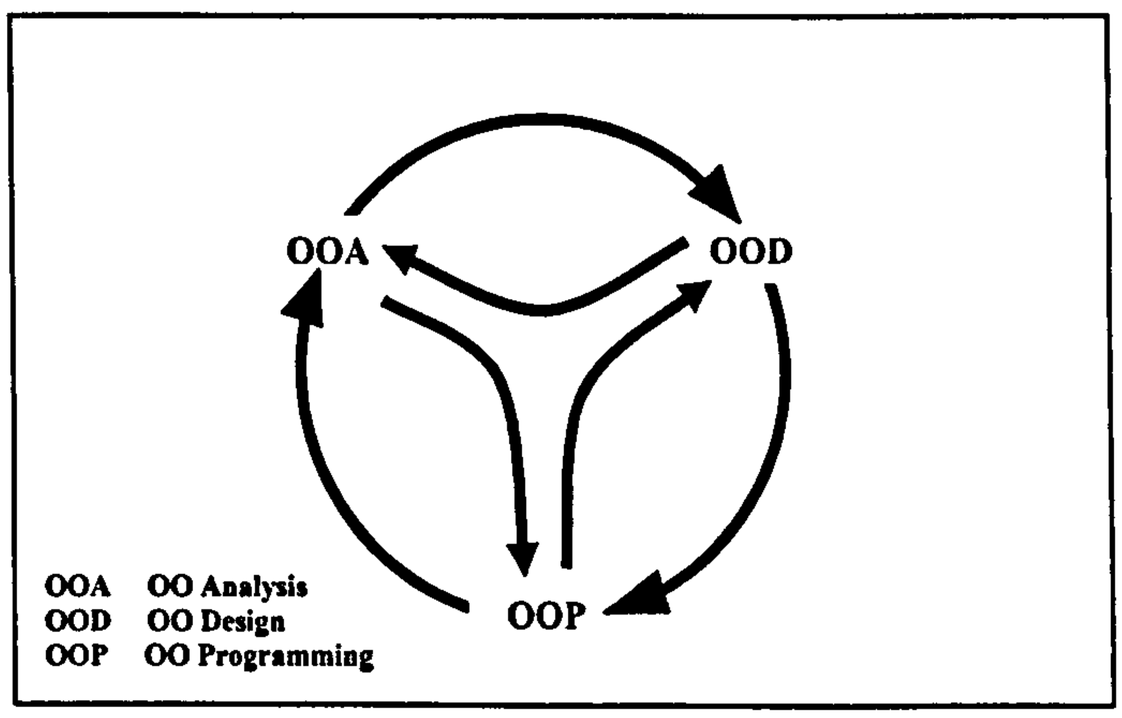

Figure 7. The Coad-Yourdon model for software development [Coad and Yourdon 1991a]

Analysis (Coad-Yourdon)

The analysis (OOA) part of the methodology consists of five principal activities:

1. Finding "Classes" (*abstract* classes) and "Class-&-Objects" (*concrete* classes)
2. Identifying "Structures" (*generalization-specialization* and *whole-part* relationships between classes)
3. Identifying "Subjects" (*partitions/subsystems*)
4. Defining *attributes*, and "Instance-Connections" (*association* relationships between classes)

5. Defining "Services" (class *operations*) and "Message-Connections" (*invocations* of operations)

Coad and Yourdon emphasize that although *initiated* sequentially, these activities are not sequential steps, since jumping from one activity to another, especially to a previously-initiated one, is inevitable. Results of these activities are reflected in a special *Class-&-Object Diagram* that is the pivotal model of the system. In accordance to these major activities, the resulting class-&-object diagram consists of five layers, each on top of the previous one, thus adding the detail in a controlled manner. These layers are:

1. *Subject* layer: which shows the overall partitions of the system. Hierarchical models of the system can be built through nesting the subjects, providing further means for complexity management.
2. *Class-&-Object* layer: showing the abstract and concrete classes of the system.
3. *Structure* layer: which shows the generalization-specification and whole-part relationships between the classes.
4. *Attribute* layer: showing the attributes of the classes and the association relationships between classes.
5. *Service* layer: which shows the operations of the classes and the potential message-passing between the objects (even the sequence of the messages can be modeled).

The class-&-object diagram is supplemented with various behavioural diagrams produced during the identification of the operations and the message-connections (activity 5 of the analysis phase). Typically, the dynamic behaviour of each class is captured in an *Object State Diagram*, a simple form of State Transition Diagram, and the algorithm that has to be applied for each of the significant services (i.e. the operation body) is described by a simple kind of flowchart, referred to as a *Service Chart*.

Design (Coad-Yourdon)

During the design phase of the methodology (OOD) the system is designed in four components, each of which provides certain functionality needed to realize the requirements and implement the system. The components are listed below:

1. *Problem Domain Component (PDC)*: initially contains the results of the analysis phase. During OOD, it is improved and enriched with implementation detail, yet still represents the part of the design containing features related to the user domain; that is, the requirements.
2. *Human Interaction Component (HIC)*: handles sending and receiving messages to and from the user. The classes in the human interaction component have names taken from the user interface language, e.g. window and menu.
3. *Task Management Component (TMC)*: for systems needing to implement multiple threads of control, the designer must construct a task management component to organize the processing, coordinate the tasks (processes) and provide means for inter-task communication. This component contains the classes that supply these services.
4. *Data Management Component (DMC)*: provides the infrastructure to store and retrieve objects. It may be a simple file system, a relational database management system, or even an object-oriented database management system. Classes in this domain typically represent relational tables, and/or more complex data/object servers.

The main diagram in each component is the class-&-object diagram (with the same five-layered architecture). Dynamic diagrams (object state diagrams and service charts) are used to augment and supplement the information they convey.

3.3.2.3 RDD (1990)

Wirfs-Brock, Wilkerson and Wiener introduced Responsibility-Driven Design (RDD) in 1990 [Wirfs-Brock et al. 1990]. The RDD process starts when a detailed requirement specification of the system has already been provided. This means that certain typical analysis activities, including requirements elicitation, have been left out of the methodology, leaving it to the engineer to decide what method to use for

producing the requirements specification. Despite this, RDD has had a great impact on modern object-oriented software engineering, since the very useful notion of *responsibility* was first demonstrated and used to perfection in this methodology.

A new version of RDD using ideas from UML and use case driven practices has also been released [Wirfs-Brock and McKean 2002].

RDD models an application as a collection of objects that collaborate to fulfil their responsibilities. Responsibilities include two key items:

1. The knowledge an object maintains.
2. The actions an object can perform.

The process is divided into two phases: the *Exploratory Phase* and the *Analysis Phase* (Figure 8). A brief overview of each phase is given in the next sections.

Exploratory Phase (RDD)

The major tasks in this phase are to:

1. discover the classes required to model the application,
2. determine what behaviour the system is responsible for and assign these responsibilities to specific classes, and
3. determine what collaborations must occur between classes of objects to fulfil the responsibilities.

As seen in the diagram depicting the RDD process, the three activities of *Identifying Classes*, *Identifying Responsibilities* and *Identifying Collaborations* should be performed iteratively in order to be effective, since the results of each activity will affect the outcome of the others. The responsibility-driven design method uses *CRC (Class-Responsibility-Collaborator) cards* – first introduced by Cunningham and Beck – in order to capture classes, responsibilities and collaborations. These cards also record subclass-superclass relationships and common responsibilities defined by superclasses.

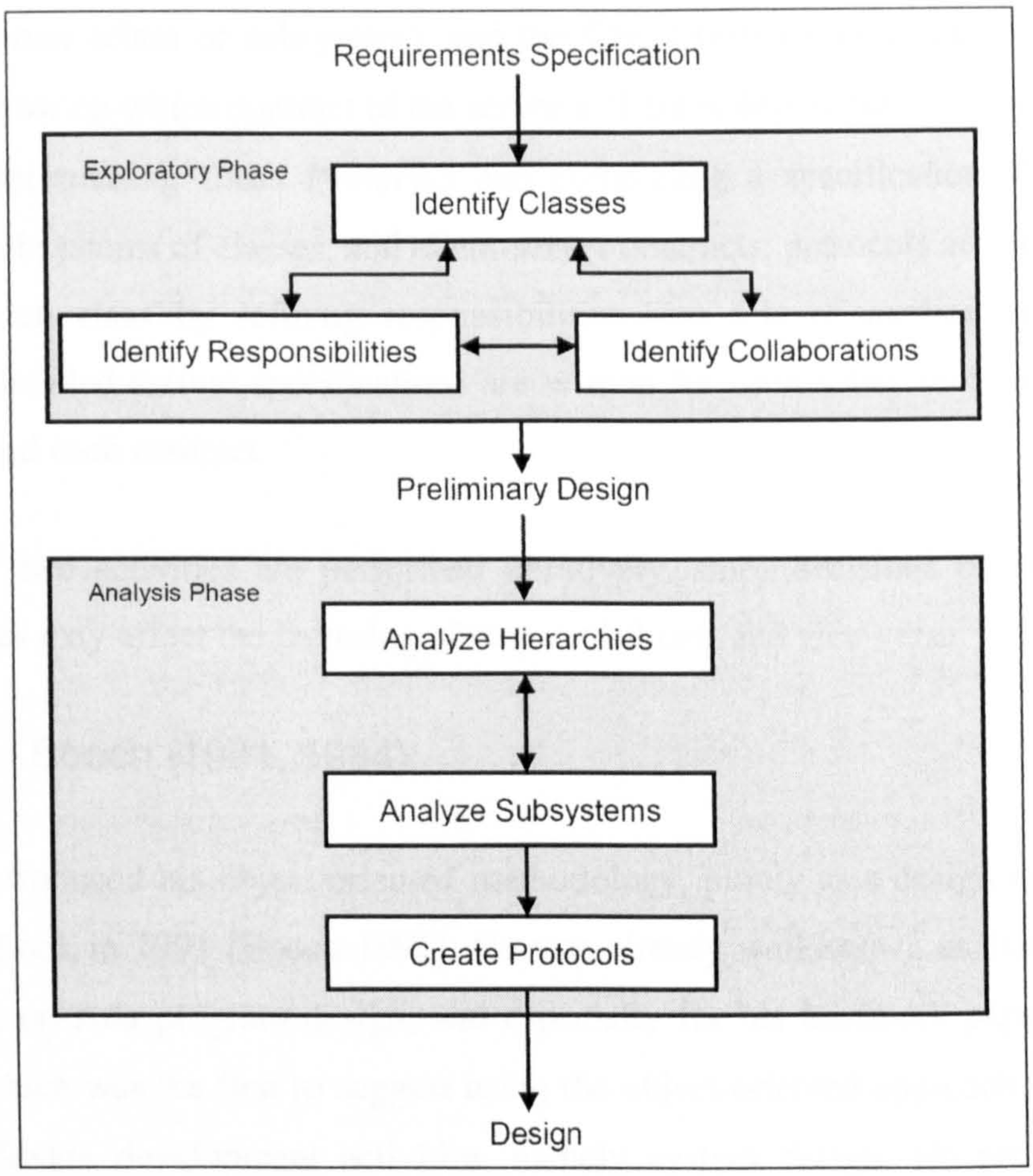

Figure 8. The RDD process [Wirfs-Brock et al. 1990]

Analysis Phase (RDD)

During the second phase of RDD, the following activities are primarily performed:

1. Factoring the responsibilities into inheritance hierarchies to get maximum reusability from class designs. Inheritance hierarchies are modeled in *Inheritance Graphs*. Responsibilities of each class are clustered into *contracts*, i.e. the list of requests that a client can make of the class. A class may support numerous contracts, showing different behaviour to different clients.
2. Identifying possible subsystems of objects and modeling the collaborations between objects in more detail. This activity involves modeling the structure of the subsystems and their contents (objects and other subsystems), along with the client-server relationships between them, in *Collaboration Graphs*. These diagrams also show the contracts of each

server (class or subsystem), and the Client-server relationships explicitly show on which contract of the server a client is dependent.

3. Determining *Class Protocols* and completing a specification of classes, subsystems of classes, and client-server contracts; protocols are defined for each class by refining responsibilities into sets of method signatures. Detailed textual specifications are written for each subsystem, each class, and each contract.

The first two activities are performed iteratively, since decisions on subsystem boundaries may affect the factoring of responsibilities, and vice-versa.

3.3.2.4 Booch (1991, 1994)

Booch introduced his object-oriented methodology, purely as a design method, in his first book in 1991 [Booch 1991]. He was already well known at that time for his work on Ada program design, and especially for his landmark paper [Booch 1986], which was the first to suggest using the object-oriented approach in higher-level software development activities, namely system design. He presented an extended version of his methodology, which also covered analysis, in his second book [Booch 1994]. Booch has modeled object-oriented design as a repeating process (referred to as "The Micro Process") within a lifecycle-level repeating process (referred to as "The Macro Process"). It has been likened to a *wheel* (the micro process) spinning along a *road* (the macro process).

The macro process serves as a controlling framework for the micro process. It represents the activities of the development team on the scale of weeks to months. Many parts of this process are basic software management practices such as quality assurance, code walkthroughs, and documentation. The focus at this level is more upon the customers and their desires for things such as quality, completeness, and scheduling. Figure 9 shows the macro process as prescribed by Booch (the self-iterations represent the micro process).

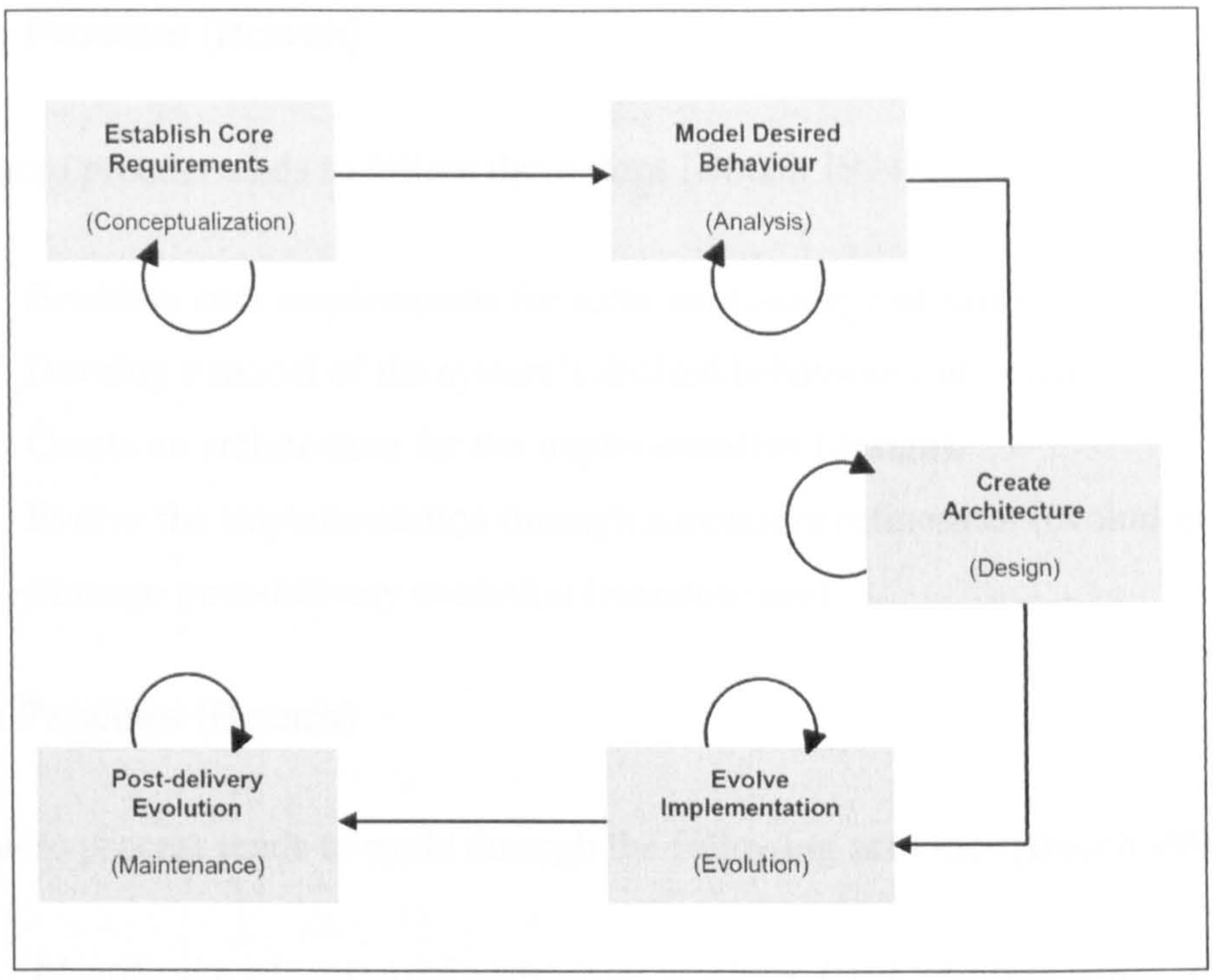

Figure 9. The Macro Process of the Booch methodology [Booch 1994]

The micro process is driven by the scenarios and architectural specifications that emerge from the macro process. It represents the daily activities of the individual or small group of developers. Figure 10 shows the various tasks involved in the micro process.

These two processes are further described in the next sections.

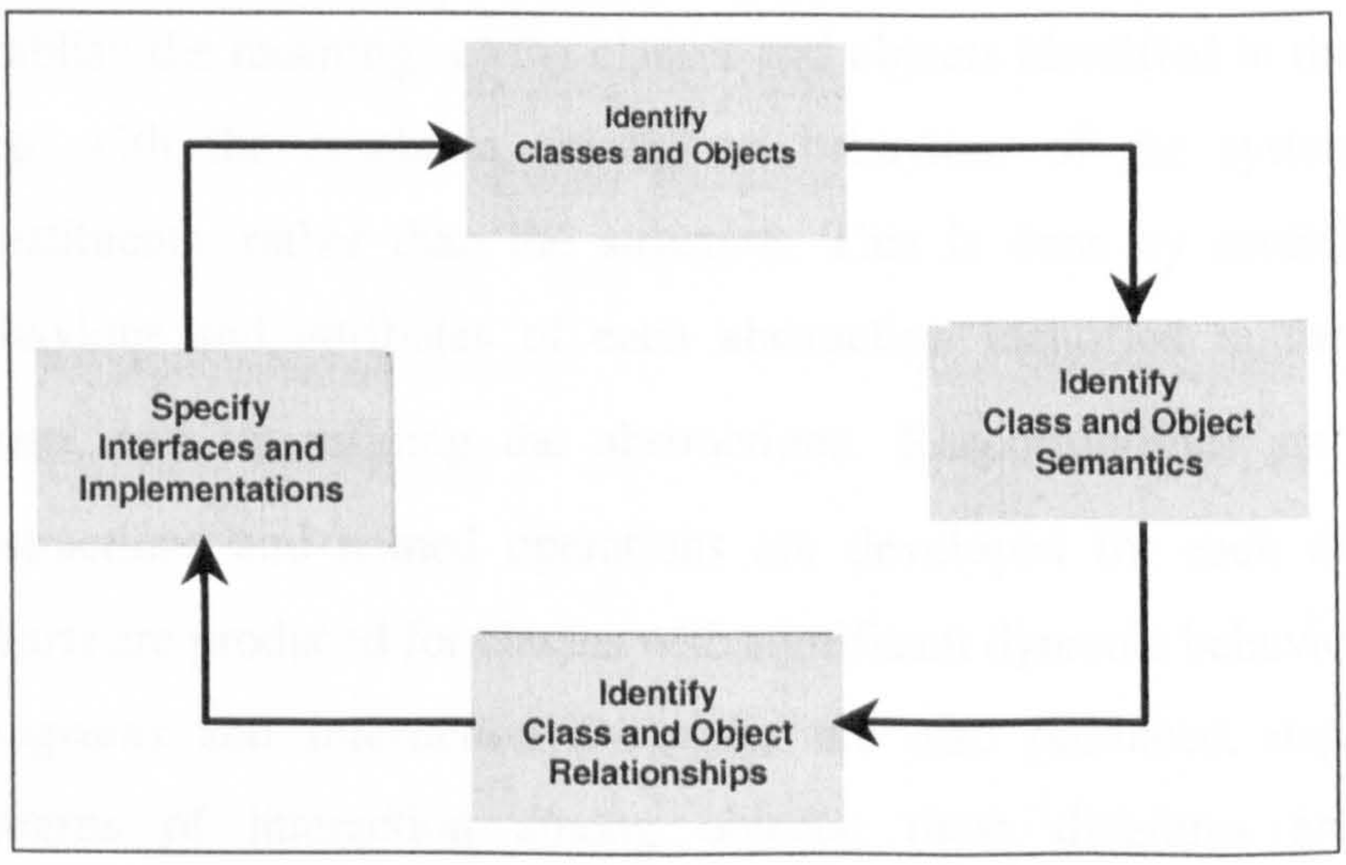

Figure 10. The Micro Process of the Booch Methodology [Booch 1994]

Macro Process (Booch)

The macro process tends to follow these steps [Booch 1994]:

1. Establish core requirements for software (conceptualization).
2. Develop a model of the system's desired behaviour (analysis).
3. Create an architecture for the implementation (design).
4. Evolve the implementation through successive refinement (evolution).
5. Manage post-delivery evolution (maintenance).

Micro Process (Booch)

The micro process tends to cycle through the following activities [Booch 1994]:

1. *Identify the classes and objects at a given level of abstraction*, through establishing the boundaries of the problem, finding abstractions in the problem domain, constraining the problem and identifying what is and is not of interest, and generating a *Data Dictionary*, which specifies all classes and objects in the development. Due to the iterative nature of the micro process, the data dictionary can change during development. Booch advocates the use of CRC cards (explained in Section 3.3.2.3) throughout the process. Classes identified during the earlier phases of the macro process mainly belong to the problem domain, while those added during design typically belong to the implementation.
2. *Identify the semantics of classes and objects*. The purpose of this step is to establish the meanings of the classes and objects identified in the previous step, with the emphasis chiefly on behaviour of the system and its constituents, rather than the structure. This is done by establishing the behaviour and attributes of each abstraction identified in the previous phase, and by refining the abstractions. Responsibilities are added to abstractions and named operations are developed for each class. *State Charts* are produced for classes with significant dynamic behaviour. *Object Diagrams* and *Interaction Diagrams* are also produced, depicting the patterns of interaction among objects; these diagrams are actually isomorphic, with the former stressing the static *relationships* among objects, and the latter emphasizing the *sequence* of the interactions among

objects. Object diagrams are also quite useful in the earlier stages of the macro process for showing the structural relationships among objects. In this context, only the links between the objects are shown, and the arrows and sequence numbers, which determine the behavioural aspects (message passing), are left out, to be added in later stages. These simple object-diagrams are built during the third step of the micro process (identifying relationships); in other words, simple object diagrams built during the *third* activity of the micro process in the earlier stages of the project (first two phases of the macro process), are adorned with behavioural detail during the *second* activity in later iterations.

3. *Identify the relationships among classes and objects.* Once behaviour is identified, the next step is to determine the relationships among classes and objects. This is done by establishing exactly how things interact within the system. Patterns among classes which permit reorganization and simplification of the class structure are sought. Visibility decisions are made at this time. The end result of this step is the production of class, object and module diagrams. *Class Diagrams* show the classes and their relationships (association, inheritance, and aggregation). *Module Diagrams* are typically built during later stages of the macro process and are used to show the physical modules and the interdependencies among them, thus depicting the physical architecture of the system.
4. *Specify the interface and implementation of classes and objects.* Design decisions are made concerning the representation of the classes and objects already identified. Classes and objects are allocated to modules, and processes implementing these modules are allocated to processors. Module diagrams are adorned with additional detail, and *Process Diagrams* are produced. A process diagram shows the hardware platform architecture of the system by depicting the processors and devices and their interconnections. It also shows which processes are allocated to each processor.

Typically, the stress is gradually shifted from the earlier activities of the micro process to the later ones as the project moves through the macro process, from conceptualization to analysis and then to design. However, due to the iterative

nature of the overall process, it is likely that earlier activities of the micro process will be revisited throughout the design.

3.3.2.5 OMT (1991)

OMT (Object Modeling Technique) was introduced by Rumbaugh et al. in 1991 [Rumbaugh et al. 1991]. The methodology is categorized as *combinative* [Monarchi and Puhr 1992], since it uses three different models (analogous to the old structured SA/SD methodology [DeMarco 1978, Yourdon and Constantine 1979]) and then defines a method for integrating them. The three models by which OMT graphically defines a system are:

1. *The Object Model (OM)*: The object model is the pivotal model. It depicts the object classes in the system and their relationships, as well as their attributes and operations, and thus represents the static structure of the system. The object model is represented graphically by a *Class Diagram.*
2. *The Dynamic Model (DM)*: The dynamic model indicates the dynamics of the objects and their changes in state. It captures the essential behaviour of the system by exploring the behaviour of the objects over time and the flow of control and events among the objects. Scenarios of the flow of events are captured in *Event-Trace Diagrams*. These diagrams, along with *State Transition Diagrams (State Charts)*, compose the OMT dynamic model.
3. *The Functional Model (FM)*: The functional model is a hierarchical set of *Data Flow Diagrams (DFDs)* of the system and describes its internal processes without explicit concern for how these processes are actually performed.

Each model describes one aspect of the system but contains references to the other models: the object model describes the data structure that the dynamic and functional models operate on; the operations in the object model correspond to events in the dynamic model and functions in the functional model; the dynamic model describes the control structure of objects, showing decisions that depend on object values and which cause actions that change object values and invoke functions; the functional model describes functions invoked by operations in the

object model and actions in the dynamic model; functions operate on data values specified by the object model; the functional model also shows constraints on object values.

The OMT process consists of five phases, as shown in Figure 11. A use case driven version of OMT, coined OMT-2, was proposed by Rumbaugh in 1994 [Rumbaugh 1994]; in OMT-2, *Use Case Diagrams* and *Object Interaction Diagrams* replace DFDs as constituents of the functional model.

The first three phases (*Analysis*, *System Design* and *Object Design*), which are considered the primary features of the OMT process, are described in the next sections.

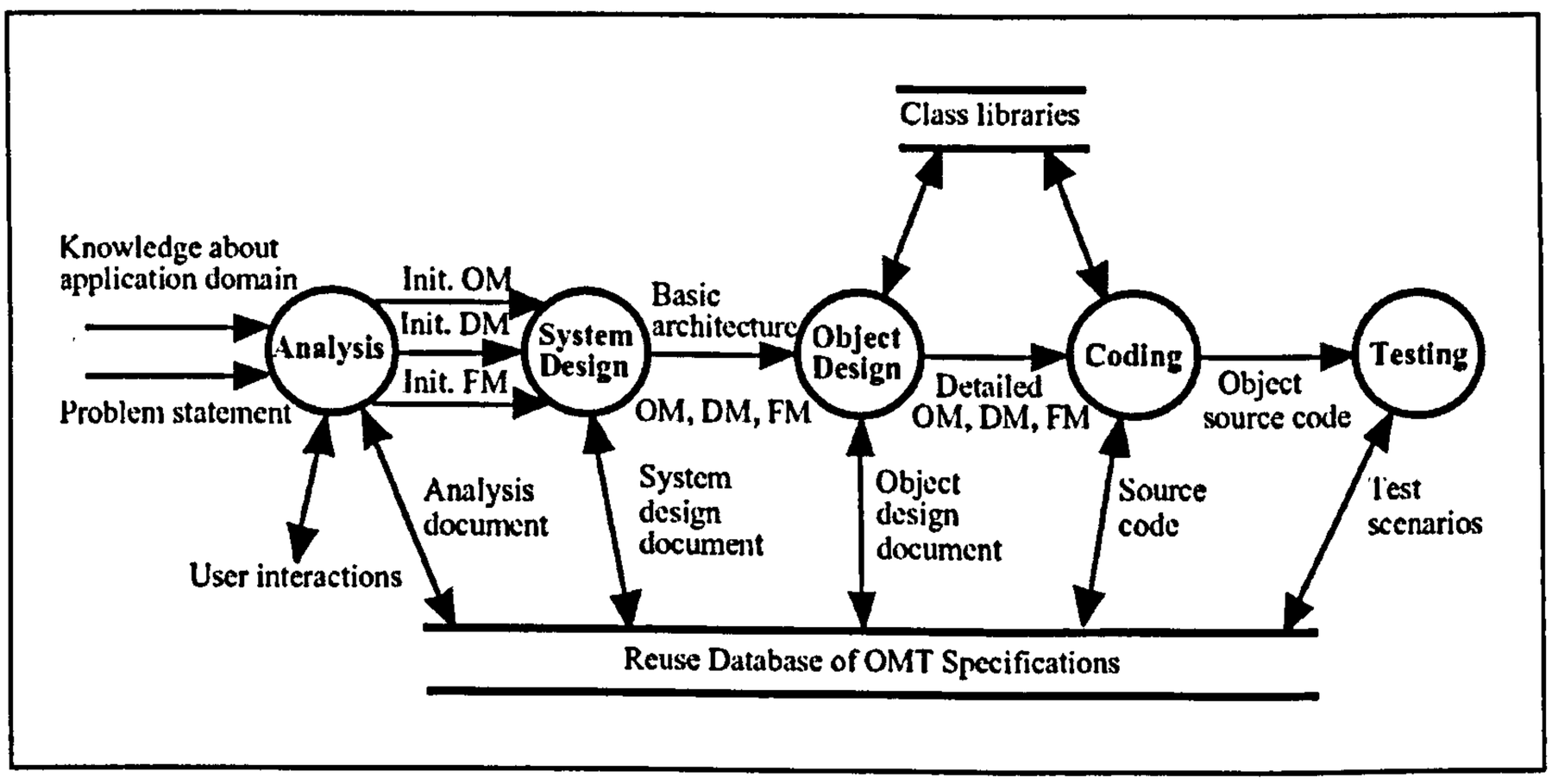

Figure 11. The OMT process and its deliverables [Derr 1995]

Analysis (OMT)

The goal of analysis is to build a correct and comprehensible model of the real world. Requirements of the users, developers and managers provide the information needed to develop the initial problem statement. Once the initial problem is defined, the following tasks are carried out:

1. Building the object model, including a *Class Diagram*, depicting the classes of the system and their relationships, and a *Data Dictionary*.

2. Developing the dynamic model, including *State Transition Diagrams* and global *Event-Trace Diagrams*. The OMT identifies the following steps in constructing the dynamic model:
    2.1. Identifying patterns of system usage and preparing scenarios of typical interaction sequences.
    2.2. Identifying events between objects and preparing an event-trace diagram for each scenario.
    2.3. Preparing an event-trace diagram for the system, showing events flowing at the boundary of the system.
    2.4. Developing state transition diagrams for classes with important dynamic behaviour.
    2.5. Checking for consistency and completeness of events shared among the state transition diagrams.
3. Constructing the functional model including *Data Flow Diagrams* and constraints.
4. Verifying, iterating, and refining the three models.

System Design (OMT)

During system design, the high-level structure of the system is chosen. The decisions that will be addressed during system design are:

1. Organizing the system into subsystems.
2. Identifying concurrency.
3. Allocating subsystems to processors and tasks.
4. Choosing the strategy for implementing data stores in terms of data structures, files, and databases.
5. Identifying global resources and determining mechanisms for controlling access to them.
6. Choosing an approach to implementing software control.
7. Considering boundary conditions.
8. Establishing trade-off priorities.

Object Design (OMT)

Object design is concerned with fully specifying the existing and remaining classes, associations, attributes, and operations necessary for implementing the system. Operations and data structures are fully defined along with any internal objects needed for implementation. In essence, all of the details for fully determining how the system will be implemented are specified during object design.

3.3.2.6 OSA (1992)

OSA (Object-oriented Systems Analysis) was introduced in 1992 by Embley, Kurtz and Woodfield [Embley et al. 1992]. OSA is only concerned with object-oriented analysis and does not include other phases of the generic software development lifecycle. It is considered a *model-driven* technique, in that it provides a pre-specified set of fundamental concepts with which to model the system under study, and therefore lacks a prescribed, step-by-step process. This is in contrast to the *method-driven* approach (which is typical of lifecycle-span methodologies), casting a shadow of doubt on whether it should at all be considered a methodology. Nevertheless, there are those who believe that OSA *is* an analysis methodology, categorizing it as such alongside its method-driven counterparts [Meyer 1997]. In any case, OSA's influence on later methodologies is significant, justifying its inclusion in this review.

In OSA, the system is modeled from three perspectives: object structure, object behaviour, and object interaction. An OSA model of the system consists of three parts:

1. *Object-Relationship Model (ORM)*, which describes objects and classes as well as their relationships with each other and with the "real world".
2. *Object-Behaviour Model (OBM)*, which provides the dynamic view through states, transitions, events, actions and exceptions (analogous to a state-transition diagram).
3. *Object-Interaction Model (OIM)*, which specifies possible interactions among objects.

Complexity is managed by providing means for model layering, showing details of high-level model elements in separate lower-level diagrams. These models are briefly described in the next sections.

Object-Relationship Model – ORM (OSA)

ORM components describe *objects*, *object classes*, *relationships*, *relationship sets*, and *constraints*. An object is any identifiable entity, and an object class is a set of objects that share common properties or behaviour. A relationship links two or more objects. A relationship set is a set of relationships that associate objects from the same collection of object classes. ORM components include three special kinds of relationship sets: *generalization/specialization*, *aggregation*, and *association*. High-level object classes and high-level relationship sets are complex object classes and relationship sets described in more detail in separate ORM diagrams.

Object-Behaviour Model – OBM (OSA)

The OBM describes the behaviour of objects in a system. It consists of a collection of *state nets*, each of which defines the behaviour for the members of an object class. The primary building blocks for state nets are *states* and *transitions*. An object may be in several different states at any time. A transition consists of a *trigger* and an optional *action*. High-level states and high-level transitions are states and transitions described by other state nets.

Object-Interaction Model – OIM (OSA)

An OIM captures information about interactions between objects. OIM components include *objects*, *interactions* and various types of *constraints*. High-level interactions are those described by more detailed OIM diagrams.

3.3.2.7 OOSE (1992)

OOSE (Object-Oriented Software Engineering) was introduced by Jacobson in 1992 [Jacobson et al. 1992]. It is a simplified version of Jacobson's *Objectory* methodology, first introduced in 1987 [Jacobson 1987] and later the property of Rational Corporation (recently acquired by IBM). Covering the full generic

lifecycle, the OOSE process consists of three main phases, each producing a set of models:

1. *Analysis*: focus is on understanding the system and creating a conceptual model of it. This phase consists of two non-sequential, iterative subphases:
   1.1. *Requirements Analysis*, aiming at eliciting and modeling the requirements of the system. A *Requirements Model* is produced as a result of this activity.
   1.2. *Robustness Analysis*, aiming at modeling the structure of the system in terms of interface, data and control objects and also by specifying the subsystems making up the overall system. An *Analysis Model* is produced as the result of this activity.
2. *Construction*: focus is on creating a blueprint of the software and producing the code. This phase consists of two subphases:
   2.1. *Design*, aiming at modeling the run-time structure of the system, and also the inter-object as well as intra-object behaviour necessary to realize the requirements. A *Design Model* is produced as the result of this activity.
   2.2. *Implementation*, aiming at building the software. An *Implementation Model* (including the code) is produced as the result of this activity.
3. *Testing*: focus is on verifying and validating the implemented system. A Test Model is produced during this phase.

Figure 12 shows the OOSE process and the models produced. Although each model is built in a specific phase of the process, models are usually revisited and refined during later phases. A brief description of each phase and subphase, and the corresponding models, is given in the next sections.

Analysis (OOSE)

Concerned with understanding and modeling the system, this phase lays the groundwork for later phases, especially by producing the *Use Case Model*, which is the pivotal model of the whole process. The two subphases are executed iteratively, thereby deriving the *Requirements* and *Analysis Models* from the informal customer requirements.

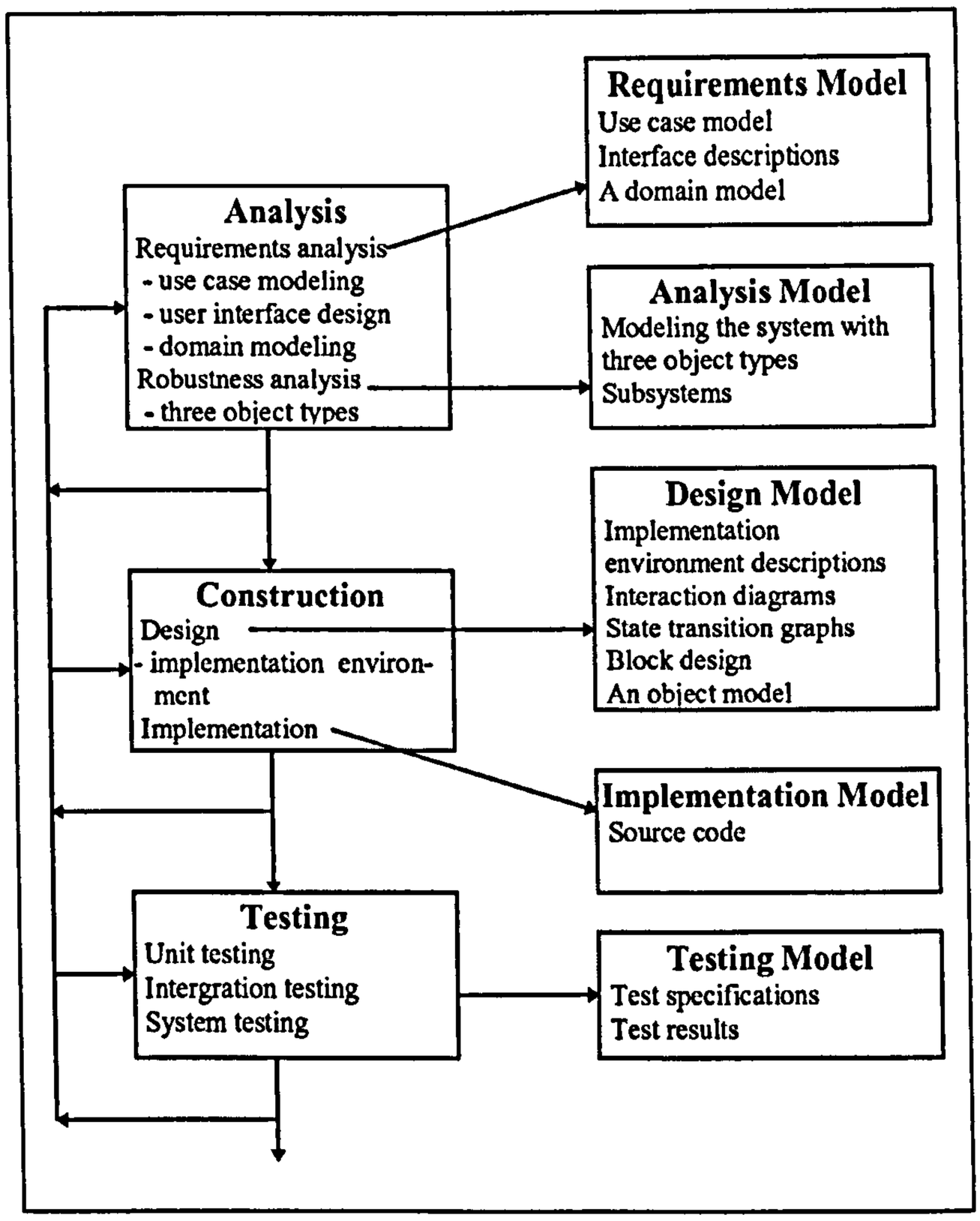

Figure 12. The OOSE process and the models produced [Jacobson et al. 1992]

Requirements Analysis

The aim of the requirements analysis subphase is to specify and model the functionality required of the system, typical means and forms of interacting with the system, and the structure of the problem domain.

The model to be developed is the *Requirements Model*, further divided into three submodels:

- A *Use Case Model*, which delimits the system and describes the functional requirements from the user's perspective. The use case model specifies the complete functional behaviour of the system by defining what entities interact with the system from outside (*actors*) and the specific ways these external entities use the system (*use cases*). A use case is defined as "a particular form or pattern or example of usage, a scenario that begins with

some user of the system initiating some transaction or sequence of interrelated events" [Jacobson et al. 1992]. In addition to depicting the relationship between the actors and their corresponding use cases (those they communicate with), a use case model can also show the relationships between the use cases themselves: a use case may *extend* another use case's behaviour, or *use* another use case in order to perform its own functionality.

- A *Domain Object Model*, which consists of objects representing entities derived from the problem domain, and their *inheritance, aggregation* and *association* relationships.
- *Interface Descriptions*, which provide detailed logical specifications of the user interface and interfaces with other systems.

The use case model is the central model of OOSE; use cases are the basis on which the whole process rests. They are directly involved in the construction of other models and enable the developers to keep constant focus on the requirements. Hence, OOSE is considered a "use case driven" methodology, and the first of an influential dynasty.

Robustness Analysis

The aim of the robustness analysis subphase is to map the requirements model to a logical configuration of the system that is robust and adaptable to change. The model to be developed is the *Analysis Model*, which shows how the functionality of each and every use case is realized by collaboration among typed objects (called *analysis objects*). These objects can be of three types:

1. *Entity*: objects of this type represent entities with persistent state, typically outliving the use cases they help realize. They are usually derived from the domain object model.
2. *Interface*: objects of this type represent entities that manage transactions between the system and the actors in the outside world.
3. *Control*: objects of this type represent functionality not inherently belonging to other types of objects. They typically act as controllers or coordinators of the processing going on in the use cases.

The developers of OOSE believe that this kind of typing improves robustness and adaptability by enhancing separation of concern among the objects.

The analysis model is derived from the use case model by spreading the behaviour in each use case among typed objects, showing how they communicate and interact in order to realize the use case. The analysis submodels thus constructed (one per use case) can also show the inheritance and aggregation relationships between the objects. In more complex systems, the analysis model also includes information on how the system can be partitioned into *subsystems*, represented as packages of analysis classes.

Construction (OOSE)

This phase is concerned with mapping the models so far produced to a physical configuration of the system. It constructs the software system by focusing on implementation issues, modeling the run-time structure and behaviour of the system, and producing the final code. The two subphases closely correspond to the generic lifecycle activities of the same names.

Design

The aim of the design subphase is to refine the analysis model by taking into account implementation features. The model to be developed is the *Design Model*, which describes the features of the implementation environment, the details of the design classes (referred to as *blocks*) necessary to implement the system, and the way run-time objects should behave and interact in order to realize the use cases. The design subphase can be broken down into three activities:

1. Determination of the features of the implementation environment; such as the DBMS, programming language features, and distribution considerations.
2. Definition of *blocks* (design classes) and their structure; each object in the Analysis Model is initially mapped to a design class, called a *block*. Implementation-specific blocks are then added and the collection is revised. The set of blocks is partitioned into *packages*, which represent the

actual implementation elements of the system. Interfaces of blocks and semantics of their operations are explicitly and comprehensively defined.

3. Specification of the sequences of interactions among objects and the dynamic behaviour of each block; an *Interaction Diagram* is drawn for each of the use cases, describing the sequence of communication among block instances at run-time for realizing the use case. OOSE interaction diagrams provide support for use cases with extensions, by using special symbols called *probe positions* for indicating a position in the use case that is to be extended (the extension use case is to be plugged into it) if a given condition is satisfied. In addition to interaction diagrams, a *State Transition Graph* is used to describe the behaviour of each block.

Implementation

The aim of the implementation subphase is to produce the code from the specifications of the packages and blocks defined in the design model. The model to be developed is the *Implementation Model*, which consists of the actual source code and accompanying documentation.

Testing (OOSE)

The aim of the testing phase is to verify and validate the implementation model. The model to be developed is the *Testing Model*, which mainly consists of the test plan, the test specifications and the test results. As usual, testing is done at three levels: starting from the lowest level, blocks are tested first, use cases are tested next, and finally, tests are performed on the whole system.

3.3.2.8 BON (1992, 1995)

The BON Methodology was first introduced in a paper by Nerson in 1992 [Nerson 1992], with the acronym standing for "*Better* Object Notation". A revised and far more detailed version of the methodology was put forward in 1995 [Walden and Nerson 1995]; this time the acronym stood for "*Business* Object Notation". Whatever the 'B' should stand for, BON is certainly not a mere notation, but a complete methodology spanning the analysis and design phases of the generic software development lifecycle. The methodology strives to be language-

independent; however, it is deeply influenced by Eiffel's assertion mechanisms and the notion of *Design by Contract* [Meyer 1997].

The BON process consists of nine steps, or *tasks*. A summary of the process tasks in the order of execution is shown in Figure 13. Tasks 1-6 focus on analysis and tasks 7-9 deal with design. The developer is allowed to change the order of the tasks if it helps achieve the goals of the project, but it is required that all the necessary models be eventually produced.

TASK		DESCRIPTION	BON DELIVERABLES
GATHERING	1	**Delineate system borderline.** Find major subsystems, user metaphors, use cases.	SYSTEM CHART, SCENARIO CHARTS
	2	**List candidate classes.** Create glossary of technical terms.	CLUSTER CHARTS
	3	**Select classes and group into clusters.** Classify; sketch principal collaborations.	SYSTEM CHART, CLUSTER CHARTS, STATIC ARCHITECTURE, CLASS DICTIONARY
DESCRIBING	4	**Define classes.** Determine *commands, queries,* and *constraints.*	CLASS CHARTS
	5	**Sketch system behaviors.** Identify events, object creation, and relevant scenarios drawn from system usage.	EVENT CHARTS, SCENARIO CHARTS, CREATION CHARTS, OBJECT SCENARIOS
	6	**Define public features.** Specify typed signatures and formal contracts.	CLASS INTERFACES, STATIC ARCHITECTURE
DESIGNING	7	**Refine system.** Find new design classes, add new features.	CLASS INTERFACES, STATIC ARCHITECTURE, CLASS DICTIONARY, EVENT CHARTS, OBJECT SCENARIOS
	8	**Generalize.** Factor out common behavior.	CLASS INTERFACES, STATIC ARCHITECTURE, CLASS DICTIONARY
	9	**Complete and review system.** Produce final static architecture with dynamic system behavior.	Final static and dynamic models; all BON deliverables completed.

Figure 13. The BON process: the tasks and their deliverables [Walden and Nerson 1995]

Each task in the BON process has a set of *input sources,* is controlled by *acceptance criteria,* and produces a set of *deliverables*. The deliverables that are created or updated as a result of each task are listed opposite the task entry in Figure 13 (the initial version of each deliverable is underscored). The goal of the BON process is to gradually build the deliverables, which provide static and dynamic descriptions of the system being developed. The static descriptions form the *static model* of the system. This model contains formal descriptions of *classes*

and their grouping into *clusters* as well as *client-server*, *inheritance*, and *aggregation* relationships between them, thereby showing the system structure. The dynamic descriptions, on the other hand, make up the system's *dynamic model*. This model specifies system *events*, what object types are responsible for the *creation* of other objects, and system execution *scenarios* representing selected types of system usage with diagrams showing object message passing.

The BON deliverables are dependent on each other; there are close mappings between some of them, and although the static and dynamic models are two very different types of system description, they are closely related, since the communicating objects in the dynamic model correspond exactly to the classes in the static architecture.

A short description of each of the BON tasks is given in the next sections. The description of each task includes a brief overview of the deliverables that are first produced in that task (underscored in Figure 13).

Delineating System Borderline (BON)

This task is concerned with the main *view* of the world that is to be understood, and the system that is going to be modeled. Through well-established information gathering and systems analysis techniques, the scope of the system and its subsystems is identified, user metaphors are compiled, and the system functionality is defined as typical usage scenarios. Overall reuse policy is also established in this task, since it will affect other tasks of the process.

The major activity in this task is to analyze the problem domain and decide which parts of it belong to the system. In BON, a system (or even the whole problem domain) consists of one or more *clusters*, each of which contains a number of classes and/or sub-clusters. Clustering is essentially a mechanism for grouping classes, yet it is also used for representing subsystems. Major subsystems are identified in this first task of the BON process if the system is overly complex. Each subsystem is modeled as a top-level cluster, later to contain classes implementing the structure and behaviour of the subsystem. The *System Chart* (one per system) contains a brief description of each top-level cluster in the system. *User metaphors* are also identified, mainly to be used for identifying classes in later tasks, yet they also help

to define the borderline of the system: combined with structural analysis of the problem domain and the system, the metaphors help indicate what parts of the problem domain reside inside the system as seen from the viewpoint of its users and domain experts, and what belongs to the outside world, thus delineating the system boundary.

Other activities of this task focus on the system and its functionality as seen from the users' perspective. Outgoing and incoming information flow is identified, major system functionality is defined, and typical use cases are determined and described as *system scenarios*. A system scenario is a description of a possible partial system execution. It is a sequence of events initiated by one or more stimuli (internal or external) and shows the resulting events in the order they occur. Some interesting system scenarios are usually collected to illustrate important aspects of the overall system behaviour. A description of the scenarios, depicting the actions fulfilled in each, is then tabulated as *Scenario Charts*.

Listing Candidate Classes (BON)

This task is mainly concerned with extracting a list of candidate classes from the problem domain. This list is entered in special tables called *Cluster Charts*. Although initialized with a list of candidate classes, the cluster charts will be refined and completed during the BON process and will ultimately contain descriptions of the classes and sub-clusters in a cluster. The analysts will also compile a glossary of technical terms and concepts used in the problem domain. All the deliverables produced are then reviewed and validated by end-users and domain experts.

Selecting Classes and Grouping into Clusters (BON)

In this task, beginning with the list of candidates produced in task 1, an initial set of concepts is formed; these concepts will then be modeled as classes, which are then grouped into clusters. This task also involves the identification of relationships (*inheritance*, *client-server*, and *aggregation*) among the classes in a cluster, and among the clusters themselves. A set of diagrams (called the *Static Architecture*), describing the relationships between the classes and clusters in the system, is the main deliverable produced. A *Class Dictionary* is also produced which is a sorted list

of the classes, containing their textual descriptions. The System Chart and Cluster Charts are updated with the results of this task.

Defining Classes (BON)

Having selected and grouped an initial set of classes, the next task is to define each class in terms of its state (the information it can provide), its behaviour (the operations it can perform), and the general rules that must be obeyed by the class and its clients. This amounts to filling in the BON *Class Charts* with: *queries,* which are functions that return information about the system state without changing it (corresponding to attributes); *commands,* which do not return any information but may change the state (corresponding to operations), and *constraints,* which are the general business rules and consistency conditions as pertinent to the class. The results of this task are then reviewed and validated by the end-user/customer.

Sketching System Behaviour (BON)

In this task, the dynamic model of the system is elaborated. Initial Scenario Charts capturing the most important types of system usage have already been constructed as a result of task 1, which are of great value for finding initial candidate classes and selecting between alternative views of the problem domain. However, a comprehensive and more detailed model of potential system usage should be built, which is the main objective of this task. External (incoming) events that trigger object communication, and also the important internal (outgoing) events that are indirectly triggered by the incoming events, are identified and listed in *Event Charts.* Classes that are instantiated during system execution and those classes that instantiate them are specified and tabulated in *Creation Charts.* For each System Scenario (depicting a typical use case of the system), the sequence of message communications between objects aimed at fulfilling the scenario is specified and modeled in an *Object Scenario*; this typically necessitates perfecting and refining the scenario charts. The dynamic model thus constructed is checked for consistency with the static model, and ultimately, reviewed and validated by the end-user/customer.

Defining Public Features (BON)

In this task, the informal class descriptions filled into the class charts during task 4 (Defining Classes) are translated into formal class interfaces (*features*) with software contracts. *Queries* become functions – which return information and typically correspond to attributes, and *commands* become procedures – which may change the system state and typically correspond to operations; the functions and procedures thus defined are referred to as *features*. *Constraints* translate into pre- and post-conditions on the operations and invariants for the whole class, thus constructing the contract. The signature of each public feature (function or procedure) is also specified. The results are shown in *Class Interfaces*, which are charts showing detailed, typed and formal descriptions of the classes and their relationships, with feature-signatures and contracts elaborated. Typing of features usually results in new client relations being discovered between classes, which are also modeled in the charts. The Static Architecture is updated to reflect the refinements done in this task.

Refining the System (BON)

This task begins the design part of the BON process, and therefore includes a repetition of many activities already performed for the analysis classes, now applied to new design classes. The existing classes (especially features, contracts and relationships) are also modified and refined in order to accommodate the design classes and implement the design decisions made. These changes in turn necessitate refinements to the dynamic model. The relevant diagrams and tables – including the Static Architecture, Class Interfaces, Event Charts, Object Scenarios, and the Class Dictionary – are updated accordingly.

Generalizing (BON)

This task concerns improving the inheritance hierarchy of the classes by factoring common state and behaviour into deferred (abstract) superclasses. The relevant diagrams and tables – including the Static Architecture, Class Interfaces, and the Class Dictionary – are updated accordingly.

Completing and Reviewing the System (BON)

In this final task, the models are polished and completed, and the overall system consistency is checked. This typically involves reviewing and perfecting the static and dynamic models, syntactic verification of the classes, and checking the consistency of class invariants and the pre- and post-conditions of routines. The relevant diagrams and tables – especially the Static Architecture, Class Interfaces, Event Charts, Object Scenarios, and the Class Dictionary – are updated accordingly.

3.3.2.9 Hodge-Mock (1992)

The methodology introduced by Hodge and Mock in 1992 was the result of research to find an object-oriented software development methodology for use in a simulation and prototyping laboratory, the sole purpose of which was to explore the feasibility of introducing higher levels of automation into Air Traffic Control (ATC) systems [Hodge and Mock 1992]. The research concluded that, of the many existing methodologies investigated, none was suitable for the purpose [Mock and Hodge 1992]. The team therefore set out to develop a methodology through integrating and extending existing methodologies, including Coad-Yourdon and Booch, with a special emphasis on incorporating *seamlessness*, *traceability* and *verifiability*. The resultant methodology is extremely rich as to the types of diagrams and tables produced during the development process, yet due to strong mapping relationships among them, versions of most diagrams and tables are directly derivable from those initially produced; the methodology, therefore, lends itself to automation and is applicable as a general-purpose methodology, despite its complexity.

The Hodge-Mock process consists of five phases:

1. *Analysis*: focusing on refining the requirements and identifying the scope, structure and behaviour of the system. This phase in turn consists of four subphases:
    1.1. *Requirements Analysis*: with the focus on eliciting the requirements of the system.

1.2. *Information Analysis*: with the focus on determining the classes in the problem domain, their interrelationships, and the collaborations among their instances.

1.3. *Event Analysis*: with the focus on identifying the behaviour of the system through viewing the system as a stimulus-response machine. The findings are then used for verifying and complementing the class structure of the system.

1.4. *Transition to System Design*: with the focus on providing a more detailed view of the collaborations among objects.

2. *System Design*: with the focus on adding design classes to the class structure of the system and refining the external behaviour of each of the classes.
3. *Software Design*: with the focus on adding implementation-specific classes and details to the class structure of the system, and specifying the internal structure and behaviour of each class.
4. *Implementation*: with the focus on coding and unit testing.
5. *Testing*: with the focus on system-level verification and validation.

Figure 14 shows these phases and the deliverables produced or updated in each. It also shows the order in which the deliverables are produced, emphasizing the interdependencies among the deliverables. Although the phases are primarily sequential, the methodology explicitly prescribes cyclic returns to previous phases and iterative development of deliverables.

A short description of each of the first three phases (*Analysis*, *System Design* and *Software Design*) is given in the next sections; the methodology does not propose a specific procedure for the Implementation and Testing phases, suggesting instead that these phases should be performed according to object-oriented programming and testing practices. The description of each phase includes a brief overview of the major deliverables that are first produced in that phase (underscored in Figure 14).

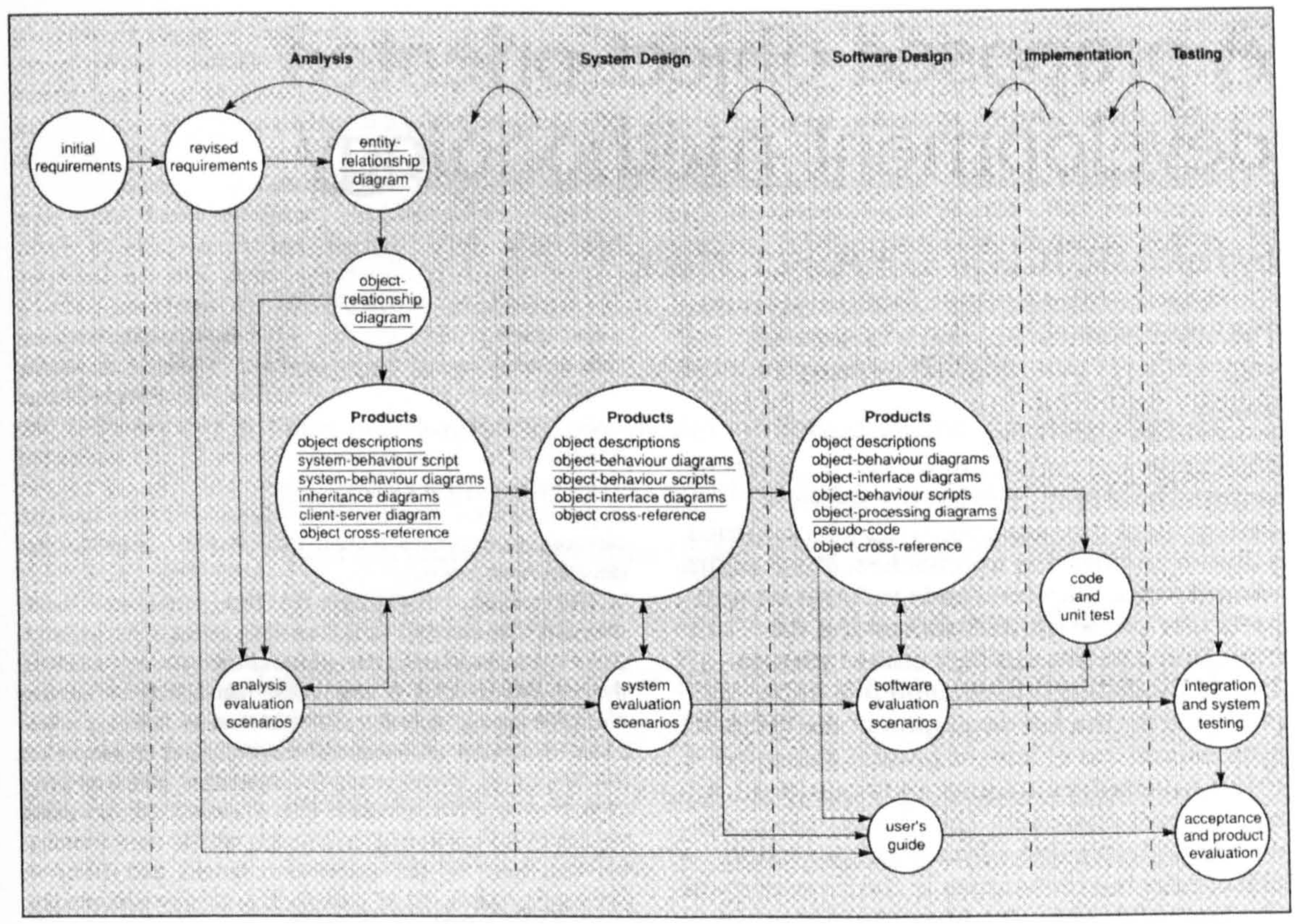

Figure 14. The Hodge-Mock process: the phases and their deliverables [Hodge and Mock 1992]

Analysis (Hodge-Mock)

The tasks performed during the analysis phase of the Hodge-Mock methodology mainly deal with requirements elicitation and problem-domain modeling. The rest of this section describes the tasks performed in each of the four subphases of analysis.

1. *Requirements Analysis*: using requirements elicitation techniques and starting from the typically ambiguous, incomplete and inconsistent problem-statement supplied by the client, the development team strives to produce a clear statement of the system's scope and its main functional and non-functional requirements. The system scope and requirements specifications thus identified will be extensively used in generating other deliverables, and will in turn be updated and refined according to later findings.
2. *Information Analysis*: The following tasks are performed in this subphase:

2.1. Using the requirements identified in the previous subphase, structural modeling of the problem domain starts with the familiar information-modeling practice of *entity-relationship modeling*: data elements, *entities*, of the problem domain, along with their attributes and interrelationships are identified and modeled in an *Entity-Relationship Diagram (ERD)*.

2.2. The entity-relationship model produced in the previous task is translated into a model of problem domain *classes*, together with their attributes, operations (*services*), and interrelationships. This is done by considering each and every entity as a candidate for being mapped onto a problem-domain class. Entities ultimately end up as either classes or attributes of classes. The resultant model is depicted as an *Object-Relationship Diagram (ORD)*. As a mechanism for managing the complexity of the ORD, the classes in the ORD can be partitioned into *subjects*, which group classes of close functional or structural relationships together.

2.3. Each of the classes identified and modeled in the ORD is described and documented in detail using a standard template. These *Object Description (OD)* documents contain detailed information about all the particulars of the classes they describe, and are gradually completed during the development process.

2.4. Class instances (objects) typically collaborate with each other in order to fulfil their expected functionalities. Identifying and summarizing these collaborations at the class-level is a major task in the Hodge-Mock methodology. For each of the classes identified so far, a list is made of its services and the services that the class requires from other classes in order to be able to provide its expected functionality. The findings are tabulated in the *Object Cross-Reference (OCR)* table.

2.5. Using the class structure identified so far, especially the structure (attributes) and behaviour (services) of individual classes, generalization-specialization (*is-a*) relationships existing between the classes are identified and modeled separately in an *Inheritance Diagram (ID)*.

3. *Event Analysis*: The following tasks are performed in this subphase:
    3.1. Through viewing the system as a stimulus-response machine, a list of external stimuli to which the system should respond is prepared based on the purpose of the system as determined in previous subphases. The activities that the system should perform in response to these stimuli are also specified. A number of these activities are categorized as *fundamental* activities, which are directly attributable to and in support of the system's purpose, while the rest are regarded as *custodial*, in that they are secondary activities providing support to fundamental activities. The functionality of the system thus identified is summarized in a tabular form in a *System Behaviour Script (SBS)*.
    3.2. The external behaviour of the system is captured in a *System Behaviour Diagram (SBD)*, which is a State Transition Diagram showing the *states* the system can be in and state *transitions* triggered by external stimuli (*events*).
    3.3. Based on system behaviour determined in previous tasks (stimuli and activities), data elements and objects required to provide the behaviour are identified. Work starts with identifying the problem-domain entities that accompany the stimuli or the system responses, or are otherwise involved in the activities performed by the system. The set of entities, their attributes and the relationships they have among themselves is then used for verifying or updating the ERD. Based on this revised ERD, problem domain classes are determined, giving special attention to determining the classes' services and collaborations in such a way as to realize the modeled behaviour of the system. Results are used for verifying/updating the ORD, ODs, OCR, and ID.
4. *Transition to System Design*: The following tasks are performed in this subphase:
    4.1. A functional view of the interactions in the system is depicted through modeling the objects inside the system, their relationships, and the messages they pass among themselves as well as messages passed between objects residing inside the

system and the users outside. This model is shown as a *Client-Server Diagram (CSD)*. Since this model implicitly shows the boundary of the system and sets the stage for delving deeper into the dynamics of object interactions inside the system, it is considered a transition from problem domain analysis to system design.

4.2. Simple scenarios showing typical user interactions with the system are compiled in order to verify the integrity of the models produced during the analysis phase, as well as validate them as traceable to the system requirements. These *Analysis Evaluation Scenarios* are based on the latest version of the requirements specifications and are regarded as validation criteria for the set of models. The analysis models are then reviewed and, if necessary, revised in order to make sure that the scenarios can be accommodated, thereby satisfying the requirements.

System Design (Hodge-Mock)

The following tasks are performed in this phase:

1. Design classes are added to the models so far developed. These are classes that are needed for developing the target system as a computer-based system, but at the same time keep it independent from any specific implementation by assuming unlimited processing and storage capacity. Examples include generic data-structure classes such as "Linked List".
2. Based on the system-level object-interaction model shown in the CSD, an *Object Interface Diagram (OID)* is developed for each of the classes identified, showing interactions between instances of the individual class with other objects, be they clients of the class's services or providers of service to instances of the class. The OID is a transition from the collective view of the CSD showing all the classes, to the *single-class view*, which focuses on individual classes.
3. In order to further specify the behaviour of each class, an *Object Behaviour Script (OBS)* is built for each class, tabulating the class's services and their corresponding inner activities. In addition, a state transition diagram is

produced for every class with significant state-driven behaviour. Analogous to the SBD and following the same notation, this class-level state transition diagram is called the *Object Behaviour Diagram (OBD).*

4. Class definitions in tables and diagrams are refined in order to include the detailed signature of class services. Especially affected are the ODs and the OCR.
5. Based on the analysis evaluation scenarios, *System Evaluation Scenarios* are developed in order to verify the integrity of the models produced during the system design phase, as well as validate them as traceable to the system requirements. The system design models are then reviewed and, if necessary, revised in order to make sure that the scenarios can be accommodated, thereby satisfying the requirements.

Software Design (Hodge-Mock)

The following tasks are performed in this phase of the process:

1. Implementation-specific classes are added in order to support the physical implementation of the system in the intended execution environment. Furthermore, implementation specific refinements are made to all classes, and all the relevant tables and diagrams are updated accordingly. Interfacing with the hardware/software platform, providing support for object persistence, and satisfying non-functional requirements are the major issues necessitating additions and refinements to the models.
2. The internal structure and behaviour of each object is further refined in order to show the way data flows among the operations. This is done by producing an *Object Processing Diagram (OPD)* for every class in the system. The OPD is in fact a Data Flow Diagram (DFD) at the class level, showing the class's operations as DFD processes, the attributes as DFD data stores, and other classes (interacting with the class being modeled) as external entities. Messages to the class are shown as invocations adorned with input/output parameters, and private and public operations are discriminated.
3. Operations (*services*) with complex bodies (algorithms) are modeled with pseudo-code in order to facilitate coding and testing.

4. Based on the system evaluation scenarios, *Software Evaluation Scenarios* are developed in order to verify the integrity of the models produced during the software design phase, as well as validate them as traceable to the system requirements. The software design models are then reviewed and, if necessary, revised in order to make sure that the scenarios can be accommodated, thereby satisfying the requirements.
5. A user's guide is prepared for the system using the design models and the final version of the requirements specifications.

3.3.2.10 Syntropy (1994)

Syntropy, introduced in 1994 by Cook and Daniels [Cook and Daniels 1994], is the result of integrating object-oriented modeling techniques (based on OMT and Booch) with formal specification elements derived from Z [Wordsworth 1992], and covers the analysis and design phases of the generic software development lifecycle. Although its developers prefer it be described as a collection of modeling techniques rather than a step-by-step process, Syntropy does suggest a definite process through the levels of modeling it prescribes, since a specific sequence should be followed for developing the models. The three distinct, yet integrated, model levels used in Syntropy are:

1. *Essential Model*, which models the problem domain, totally disregarding software as a component of the system.
2. *Specification Model*, which abstractly models the requirements of the software system, treating the system as a stimulus-response mechanism, and assuming a computing environment with unlimited resources.
3. *Implementation Model*, which models the software system's run-time structure and behaviour in detail, taking into account considerations pertaining to the computing environment, and elaborating on how the software objects should communicate.

Each model may be expressed along structural and behavioural *views*. There are three kinds of views in Syntropy:

- *Type View* (similar to the Class Diagram used in OMT): provides the structural view by describing object types (classes), their static properties and their relationships.
- *State View* (containing diagrams similar to the State Transition Diagram used in OMT): provides the behavioural view by describing the states each object type can be in and the way it responds to stimuli by changing state and generating responses.
- *Mechanism Diagram* (similar to the Interaction Diagram used in the Booch methodology): solely used in the Implementation Model for describing the flow of messages between objects in response to stimuli.

Syntropy supports the notion of *domain*: a sub-system defined as a set of object types. It also supports the concept of *viewpoint*: a subset of an object's overall interface; thus enabling the designer to describe various interfaces to the same object.

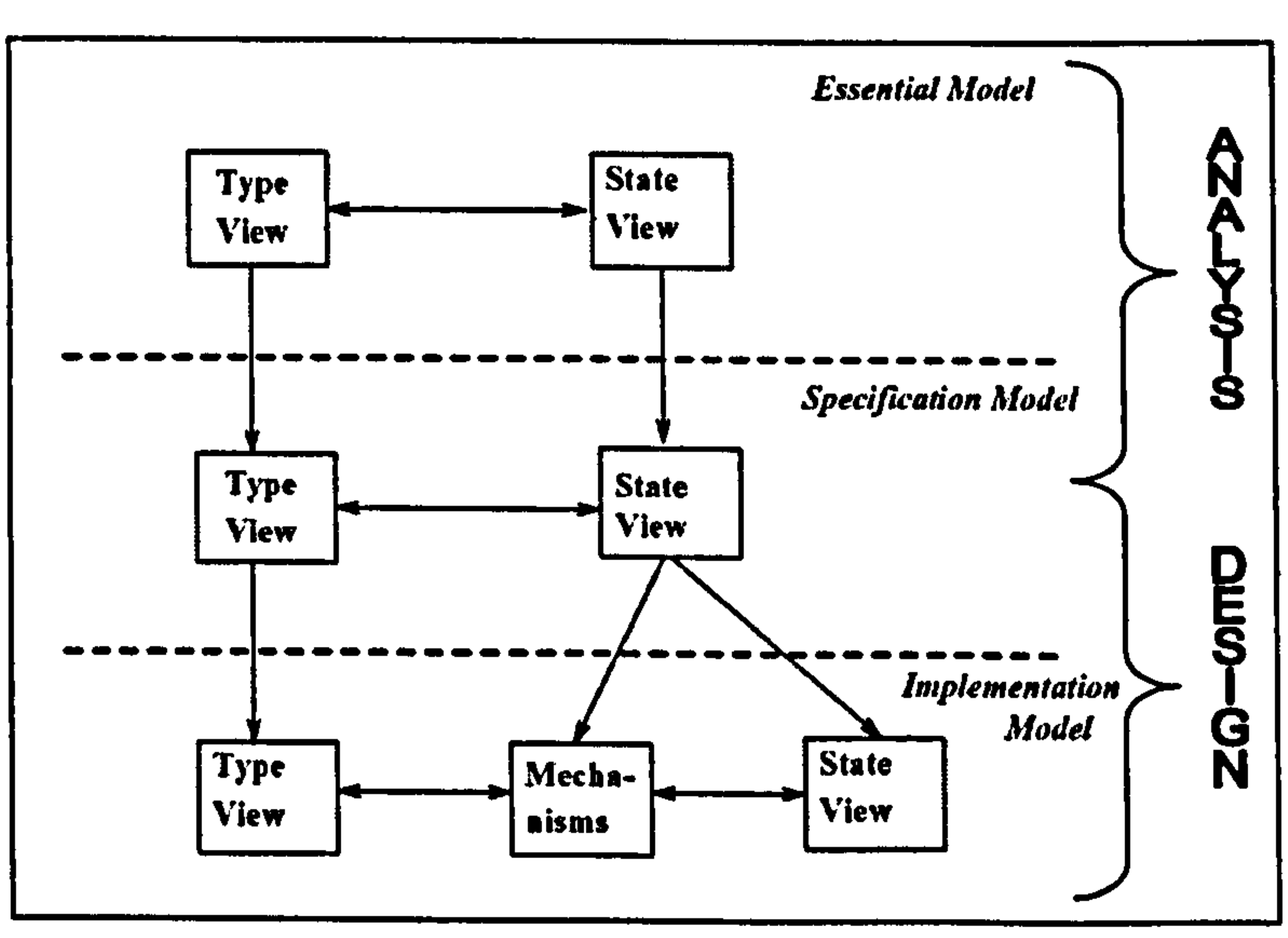

Figure 15. The implicit Syntropy process: models, views and their interdependencies [Cook and Daniels 1994]

Figure 15 shows the three models, their views and the interdependencies. This figure can also be interpreted as the process of the Syntropy methodology: the generic concept of System Analysis fits aspects of the Essential Model (analysis of the problem domain) as well as a part of the Specification Model (analysis of the required system functionality and behaviour); likewise, the generic concept of System Design is seen in the remaining part of the Specification Model (design of

state-charts and interactions between them in order to achieve required responses to external events) and the Implementation Model (algorithm construction, transformation of event generation into message-passing, etc.). Therefore, a seamless transition from Analysis to Design takes place during the construction of the Specification Model.

The next sections contain brief descriptions of the models, views and diagramming notations used.

Essential Model (Syntropy)

The Essential Model models the problem domain as a collection of objects and events. The objects' properties can only change as the result of events, and a specific event may change the properties of several objects simultaneously. The essential model consists of a *type view,* which represents the types of objects in the problem domain, and a *state view,* which represents the way objects change as a result of events.

The type view is represented by a kind of *Class Diagram,* supplemented with Z specifications for types and invariants.

The state view consists of *Statecharts,* one for each object type, showing how objects of the type respond to events. The statecharts are supplemented with information about the details of object creation, and the particulars of the events to which objects of the type can respond, including Z specifications for pre- and post-conditions of the events.

Specification Model (Syntropy)

The Specification Model describes the states that the *software* can be in, and shows how it changes state and produces events in response to stimuli. The specification model is described by the same views as the essential model; that is, a type view and a state view. The type view of the specification model represents the conceptual decomposition of the software into objects and the state view represents the behaviour of the software objects in response to stimuli, either external or issued by other objects. External stimuli are observable to all the objects in the

model simultaneously. An external stimulus may trigger several transitions in several statecharts.

To build a specification model, the system boundary should be defined. This is done by determining external entities (called *agents),* which affect or are affected by the software. Furthermore, it must be decided for each event in the essential model whether it is to be detected by the software, generated by the software, or simply ignored. The specification model should also show how undesirable events are handled, an issue neglected in the essential model.

Implementation Model (Syntropy)

The Implementation Model describes the flow of control inside the software. Stimuli are mapped to messages and all message-passing and method executions are modeled using *Mechanism Diagrams.* These diagrams specify the run-time objects, their inter-relationships (links), and the sequence of the messages passed between these objects in order to implement the external functionality of the system. A mechanism diagram is generally very similar to a Booch Interaction Diagram. The implementation model must also deal with implementation issues such as concurrency, persistence, finite resources, errors, and exceptions.

3.3.2.11 Fusion (1994)

The Fusion methodology was first introduced in 1992 by a team of practitioners at Hewlett-Packard Laboratories [Coleman et al. 1992]. A revised and detailed version of the methodology was released in 1994 [Coleman et al. 1994]. The methodology is the result of the integration, unification and extension of a number of older methodologies, mainly OMT, Booch, Objectory and RDD; hence the name *Fusion.*

The designers of Fusion describe it as a full-coverage method, in that it covers all stages of the development lifecycle from requirements to implementation, although the analysis phase starts when a preliminary informal requirements document is already available, and is in fact the main input to the whole process. Fusion provides consistency and completeness checks between phases to enable orderly

and reliable progression through system development stages. It also suggests criteria for determining when to move from one phase to the next in the lifecycle.

The Fusion process consists of three phases:

1. *Analysis*: the focus is on what the system does. The system is described from the standpoint of the user. The requirements of the system are mapped to the *System Specification,* which is expressed through a set of models. The models produced in this phase describe:
    a. classes and objects of interest found in the application domain, and the relationships which exist between these classes and objects,
    b. the operations which are to be performed by the system, and
    c. the proper ordering of these operations.
2. *Design*: the focus is on how the system is to do what has been defined during analysis. The specification of the system (the result of the previous phase) is mapped to a blueprint for the implementation of the system. The design phase models describe:
    a. realization of system operations in terms of cooperating objects,
    b. how these objects are linked together,
    c. how the classes, to which the objects belong, are specialized and refined (the inheritance structure of the classes), and
    d. the detailed particulars of each class's attributes and methods.
3. *Implementation*: the focus is on the actual coding of the system. The system design is mapped to a particular programming environment. Design classes are mapped to language specific classes and object communications are encoded as implementation methods.

Figure 16 shows the Fusion process, the models produced, and the interdependencies between the models, describing what the models contribute to each other. This figure also shows the construction of a *Data Dictionary* as an ongoing task throughout the phases of Fusion. This dictionary is a repository of detailed information, including constraints and assumptions, about all the elements in the models.

Each phase in the Fusion process consists of a number of sub-phases. A brief description of each phase, the sub-phases and the models produced is given in the next sections.

Analysis (Fusion)

The analysis phase is concerned with capturing the requirements of the system completely, consistently and unambiguously. The requirements specification document is the standard input to the analysis phase. Two models are produced in this phase: a *System Object Model* and a *System Interface Model*, the latter further divided into two models, the *Life-Cycle Model* and the *Operation Model*, all using the data dictionary as a central repository.

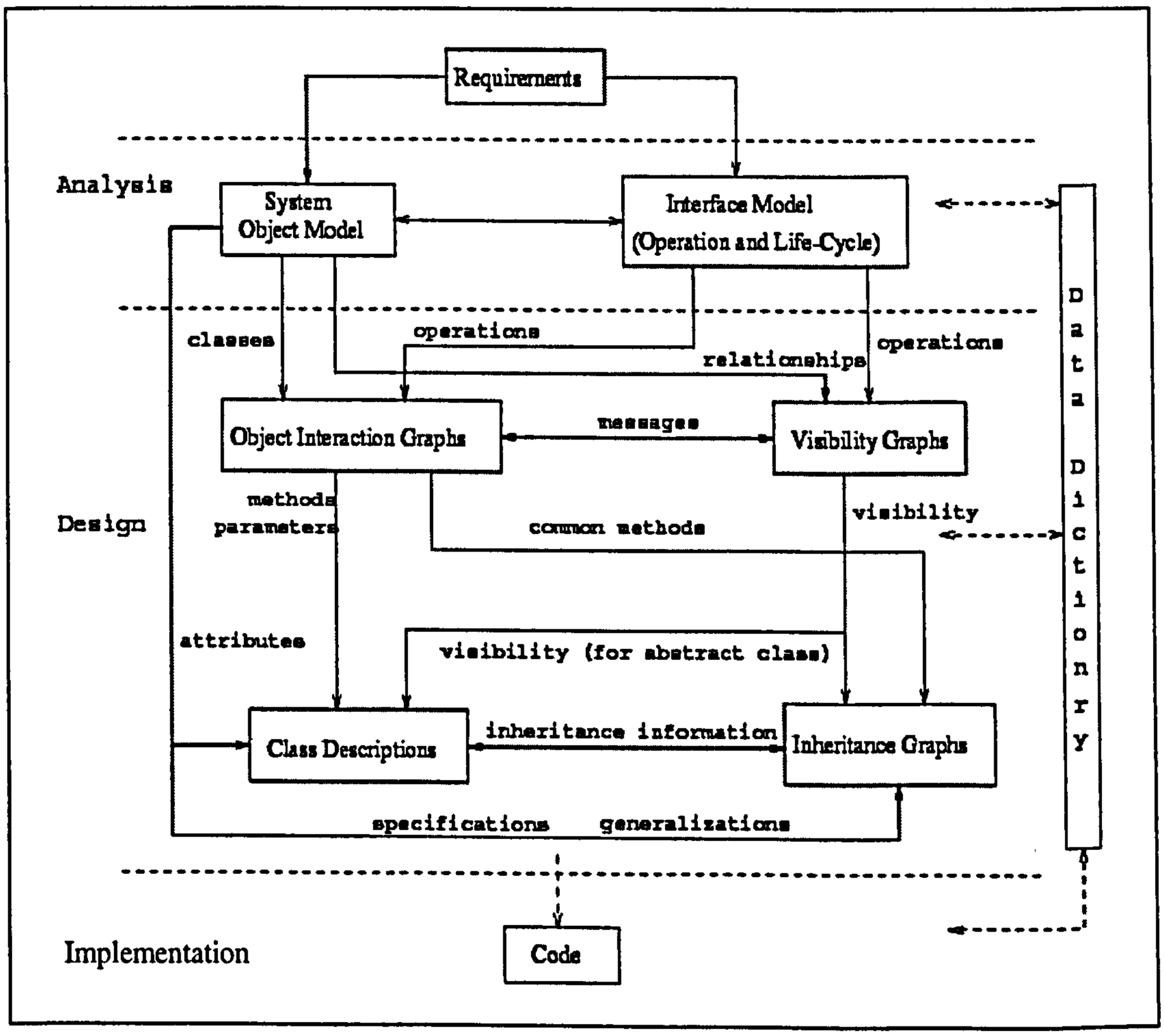

Figure 16. The Fusion process and its deliverables [Lano et al. 2000]

The analysis phase consists of the following steps:

1. Develop an overall *Object Model* encompassing the system and its environment; the static structure of the problem domain is specified in terms of objects and their relationships. The initial list of objects, and the classes to which they belong (along with their attributes), is produced by grammatically parsing the informal requirements document. The list is then completed through close observation of the system and communication with the domain experts. The results are modeled in a static structural Object Diagram.
2. Develop the *System Object Model*; the collection of classes in the *overall* object model, produced in the previous step, will include classes that belong to the environment as well as classes that belong to the system. The *system* object model, on the other hand, excludes the environment classes and focuses on the system classes by explicitly showing the boundary of the system. This model is produced through:
   2.1. Determining interaction patterns between the system and outside *agents* (users, devices or other systems); agents interact with the system by means of *events*. Input events typically lead to state changes in the system, possibly leading to output events. An input event and its effect on the system are collectively called a system *operation*. Typical interactions are modeled as *Transaction Scenarios,* explicitly showing the time ordering of the events by using *time-lines*.
   2.2. Specification of the *System Interface Diagram* (not to be confused with the System Interface *Model*), showing all the events interchanged between the system and outside agents, regardless of the time order; this in fact is the result of integrating all transaction scenarios previously identified.
   2.3. Producing the system object model by adding a boundary to the overall object model; by identifying the agents that interact with the system, the operations of the system, and events affecting or generated by the system, a good idea is obtained of which objects belong inside the system boundary, and which belong to the environment. This in turn enables the analyst to add a system

boundary to the overall object diagram resulting in the system object model. It is important to note that although class *attributes* are specified in this diagram, class *operations* (methods) are intentionally ignored, since Fusion leaves their specification to the design phase.

3. Develop the *System Interface Model* through:
   3.1. Developing the *Life-Cycle Model*: a life-cycle model shows the allowable sequences of system operation invocations throughout the lifetime of the system. The ordering of the events (input and output) is specified in terms of a regular-expression-like language.
   3.2. Developing the *Operation Model*: the operation model captures the details of all the system operations already depicted in the interface diagram and the life-cycle model. Each system operation is textually and semi-formally described by an *Operation Schema*. The resulting schemata make up the operation model.
4. Check the analysis models; Fusion provides detailed checklists for verifying the completeness and consistency of the analysis models.

Design (Fusion)

The purpose of the design phase is to find a strategy for implementing the specification of the system, which has been developed during the analysis phase. The output of the design phase consists of four parts: a set of *Object Interaction Graphs* describing how objects interact for implementing system operations; a set of *Visibility Graphs* describing object communication paths; a set of *Class Descriptions* providing detailed descriptions of class interfaces; and a set of *Inheritance Graphs* elaborating the inheritance relationships between classes.

The design phase consists of the following steps:

1. Develop the *Object Interaction Graphs*; object interaction graphs are used to develop system operations described in the operation model. Each system operation should be realized by an object interaction graph, which describes how the system operation is implemented through object

interactions and message passing. Typically, in every interaction graph, one of the objects (termed the *controller*) initiates the message sequence in response to an input event.

2. Develop the *Visibility Graphs*; a visibility graph describes the *server* objects that a *client* object needs to reference and specifies the kind of reference that is needed. The visibility of objects is described using the following characteristics: Reference Lifetime (temporary or permanent), Server Visibility (exclusive or shared), Server Binding (the degree of lifetime-dependency between the client and the server), and Reference Mutability (whether a server can be changed).
3. Specify the *Class Descriptions*; class descriptions store detailed information about classes, including class name, immediate superclasses, attributes, and methods. A class description is built for every class in the system.
4. Develop the *Inheritance Graphs*; generalization-specialization hierarchies previously identified among analysis classes are enhanced by factoring out common structure and behaviour in order to increase reusability and maintainability. The result is summarized in inheritance graphs.

Implementation (Fusion)

This phase concentrates on the conversion of the design models into a suitable language. Design features are mapped to code as follows:

1. Inheritance, references, and attributes are implemented using corresponding features of the target language.
2. Object interactions are implemented as methods in the appropriate classes.
3. State machines are developed for implementing permissible sequences of operations.

3.3.3 Methodologies: Integrated

After the initial disastrous fan-out of object-oriented methodologies, along came the inevitable fan-in, yet integration of methodologies was not as successful as integration of modeling languages: whereas the latter resulted in the advent of UML, the former produced over-complex mega-methodologies. Although many of

these Integrated (*Third-Generation*) methodologies have adopted UML as their modeling language, they share little else, particularly as pertaining to process and modeling approach.

3.3.3.1 OPM (1995, 2002)

Object-Process Methodology (OPM) was introduced by Dori in 1995, primarily as a novel approach to analysis modeling that advocated combining the classic process-oriented modeling approach with object-oriented modeling techniques [Dori 1995]. Over the years, it has evolved into a full-lifecycle methodology [Dori 2002a], yet its unique modeling approach is still the main feature attracting researchers and developers.

OPM's modeling strength lies in the fact that only one type of diagram is used for modeling the structure, function and behaviour of the system. This single-model approach avoids the problems associated with model multiplicity, but the model that is produced can be complex and hard to grasp.

The single diagram type is called the *Object-Process Diagram (OPD)*, and uses elements of types *object* and *process* to model the structural, functional and behavioural aspects of whatever is being modeled (hence the prefix Object-Process in OPD and OPM). The basic OPD notation was later expanded to also include elements of type *state*, which were particularly useful in modeling real-time systems. Variants of the notation were also developed for modeling other types of systems, including web-applications, semantic web services, and multi-agent systems.

Every OPD can also be expressed in textual form; a constrained natural language called the OPL (Object-Process Language) is provided by the OPM for this purpose. OPL equivalents can be automatically generated from the OPDs and are typically used as documentation complements of the OPDs, based on the assumption that they are more intelligible to the users and domain-experts and easier to convert to code than the OPDs [Dori 2002a].

In OPM, a set of OPDs is built for the system being developed, typically forming a hierarchy, somewhat analogous to the hierarchy of Data Flow Diagrams built in

classic process-oriented methodologies. This layering of OPDs is applied as a complexity management technique and helps improve the intelligibility of the models, yet the multi-dimensional nature of the OPDs makes it difficult to focus on a particular aspect of the system (such as structure), without being distracted by other aspects. Elements depicting different aspects are so intertwined that separating them in order to examine them in their own context can be a formidable task. Furthermore, some important orthogonal behavioural aspects of systems (such as object interactions, especially with regard to message sequencing) cannot be adequately captured in OPM models.

In contrast with OPM's strong emphasis on the modeling approach and the associated notational conventions, the OPM *process* is little more than an abstract framework. It resembles the generic software development process described in basic software engineering textbooks. This may be a consequence of the single-model approach: the lack of multiple models (whose relationships and interdependencies are often reflected in processes) seems to have had a simplifying effect on the process.

The OPM process consists of three high-level subprocesses:

1. *Initiating*: with the focus on preliminary analysis of the system, determining the scope of the system, the required resources, and the high-level requirements.
2. *Developing*: with the focus on detailed analysis, design and implementation of the system.
3. *Deploying*: with the focus on the introduction of the system into the user environment, and the subsequent maintenance activities performed during the operational life of the system.

In the following sections, a short description is given for each of the above subprocesses.

Initiating (OPM)

The following activities are performed during this subprocess:

1. *Identifying*: the needs and/or opportunities justifying the development of the system are determined.
2. *Conceiving*: the system is "conceived" through determining its scope and ensuring that the resources necessary for the development effort are available.
3. *Initializing*: the high-level requirements of the system are determined.

Developing (OPM)

The following activities are performed during this subprocess:

1. *Analyzing*: mainly concerned with eliciting the requirements, modeling the problem domain and the system in OPDs (and their OPL equivalents), and selecting a skeletal architecture for the system.
2. *Designing*: the major activities of which are adding implementation-specific details to the models (OPDs and their OPL equivalents), and refining the architecture of the system by determining its hardware, middleware and software components. Designing the software components mainly involves detailing the process logic (to be implemented as the program), the database organization, and the user interface.
3. *Implementing*: mainly focused on constructing the components of the system and linking them together. Construction typically involves coding and testing the software components (mainly consisting of the process logic of the system, the database and the user interface), setting up the hardware architecture, and installing the software platform (including the middleware). Design models (OPDs and their OPL equivalents) can be used for automatic or semi-automatic generation of the code.

Although seemingly sequential, the above activities can be performed in an iterative and incremental fashion; in fact, the methodology suggests return-loops from implementation to design and from design to analysis.

Deploying (OPM)

The following activities are performed during this subprocess:

1. *Assimilating*: concerned with introducing the implemented system into the user environment, mainly involving training, generation of appropriate documents, data and system conversion, and acceptance testing.
2. *Using and Maintaining*: spanning the period during which the system is being used. The activities performed also include maintenance tasks necessary to keep the system in working order.
3. *Evaluating Functionality*: checking that the current system possesses the functionality needed to satisfy the requirements. This activity is typically performed during the Using-and-Maintaining activity in order to check whether the current system still satisfies the functional and non-functional requirements of the users; if not, a new generation of the system is needed, and the next activity in this list should be performed.
4. *Terminating*: concerned with declaring the current system as dead, applying the usual post-mortem procedures, and prompting the generation of a new system.

3.3.3.2 Catalysis (1995, 1998)

Catalysis was introduced by D'Souza and Wills in 1995, originally as a component-based formalization of OMT deeply influenced by Fusion, Objectory, Booch and Syntropy [D'Souza and Wills 1995]. A UML-based, refined version of the methodology appeared in 1998 [D'Souza and Wills 1998].

Instead of one, all-purpose process, Catalysis proposes a set of process patterns to be selected and applied according to the characteristics of the project in hand. However, it does propose a specific process for developing business systems, as shown in Figure 17. This process is used in the following sections for describing the general attitude of Catalysis towards software development, as well as the models produced in the methodology.

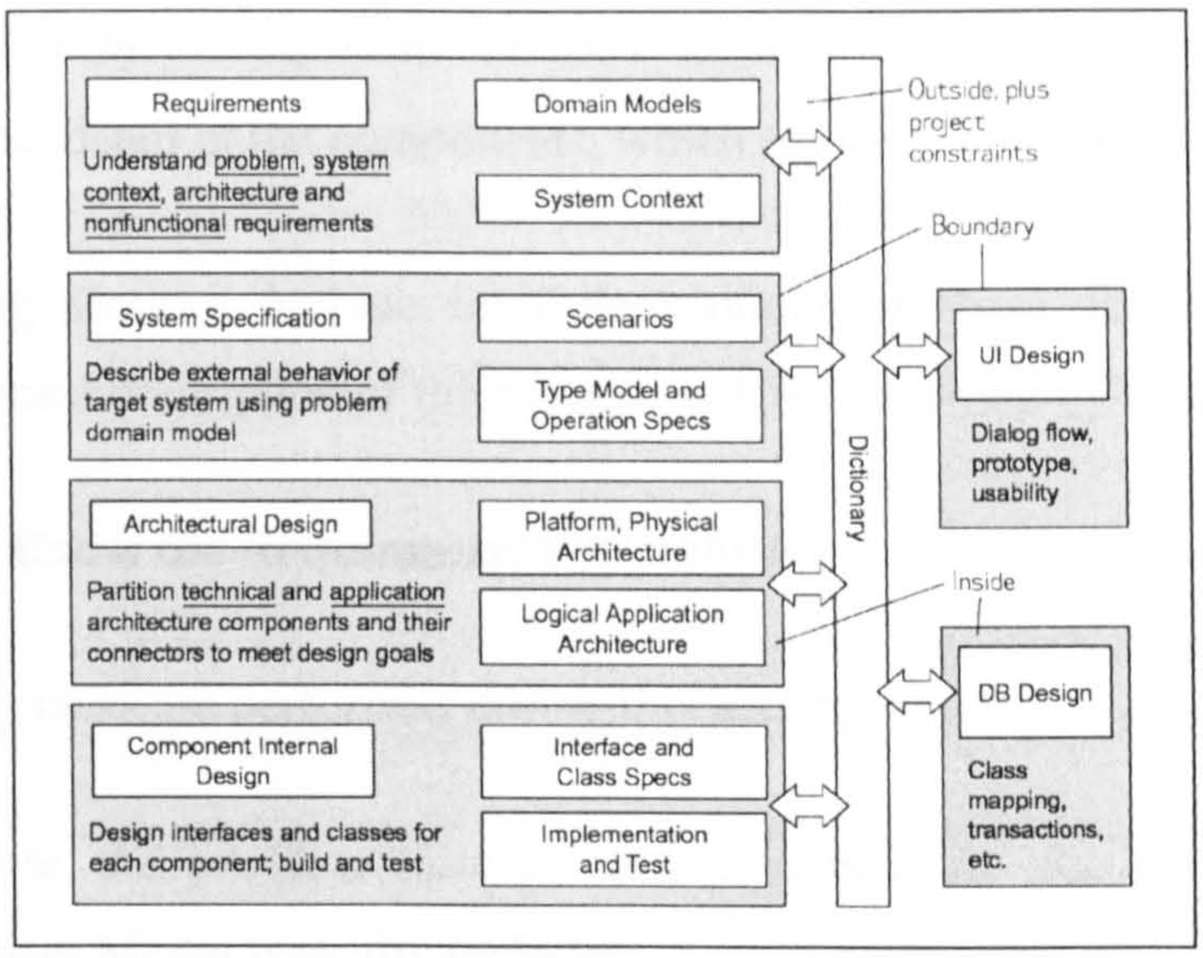

Figure 17. The Catalysis process for developing typical business systems [D'Souza and Wills 1998]

Business Systems Development Process (Catalysis)

This process consists of the following activities, gradually moving from examining and modeling the *context* of the system to specifying the system at its *boundary* and, ultimately, to designing the *interior* of the system [D'Souza and Wills 1998]:

1. *Identify and Model the Requirements*: focusing on exploration and modeling of the problem domain and the requirements of the system.
2. *Develop the System Specification*: focusing on identifying and modeling the functionality and high-level class-structure of the system. Designing the User Interface (UI) usually overlaps with this activity. UI design typically involves developing UI prototypes and UI specifications describing the screens, dialog flows across windows, information presented and required, and reports.
3. *Develop the Architectural Design*: focusing on designing the internal component (logical) architecture of the system, as well as the technical (physical) architecture defining the domain-independent parts of the system, such as the hardware and software platform. The design of the database architecture should also start at this stage, including mapping the object models to the database and definition of transaction boundaries.

4. *Develop the Component Internal Design*: focusing on designing the internal detail of the components, which are then implemented and tested.

The following sections include brief descriptions of these activities. The last section contains a description of the process patterns proposed by Catalysis.

Identify and Model the Requirements (Catalysis)

The following tasks are performed during this activity:

1. *Explore the problem domain and construct the Business Model*: the Business Model typically includes:
   - *class diagrams* depicting the *object-types* (analogous to classes) in the problem domain and their relationships,
   - special *collaboration diagrams* showing the *actions* that problem domain objects perform during interactions (without specifying the order),
   - *sequence diagrams* showing the sequence of the actions, and
   - a *glossary*, listing the terms used to define the problem domain.
2. *Identify and model the functional requirements of the system*: functional requirements are typically modeled using a *System Context Diagram* showing the system as an object in the problem domain interacting with other objects. Actions on the system are nothing but use cases, and scenarios of interaction are expressed by sequence diagrams.
3. *Identify the non-functional requirements*: such as performance, reliability, scalability, and reuse goals.
4. *Identify and model the known platform or architectural constraints*: machines, operating systems, middleware, legacy systems, and interoperability requirements are identified and modeled as package diagrams. Interactions between these physical components are captured in collaboration diagrams and sequence diagrams.
5. *Identify the project and planning constraints*: pertaining to issues such as budget, schedule, staff, and user involvement.

Develop the System Specification (Catalysis)

The system specification mainly consists of a class (*type*) diagram showing the system as a type, emphasizing its attributes (internal types) and its associations with other types in the problem domain. The system also has a set of operations, depicting the actions that it performs (functionality). The detailed behaviour of the system is usually captured in statecharts.

Develop the Architectural Design (Catalysis)

The following tasks are performed during this activity:

1. *Identify the components comprising the system and their architecture*: The component (*application*) architecture is usually described with package diagrams showing the components and their inter-relationships. Specification types (system attributes) identified during the previous activity are split across different components. Interaction among components is modeled through collaboration diagrams.
2. *Identify the architecture of the domain-independent parts of the system*: hardware and software platforms, infrastructure components (such as middleware and databases), utilities for logging/exception-handling/start-up/shutdown, design standards and tools, and the choice of component architecture (such as JavaBeans or COM), are all modeled in the *Technical Architecture*. Package diagrams are used to show these physical components and their inter-relationships. Interactions are shown in collaboration diagrams.

Develop the Component Internal Design (Catalysis)

During this activity, each and every component is designed, implemented and tested. Design is done by identifying the programming language interfaces and classes, or pre-existing components, that constitute the component. The architecture of these parts inside each component is modeled using a package diagram showing the internal constituent parts and their inter-relationships. Interactions are shown by sequence and collaboration diagrams.

Process Patterns (Catalysis)

Even though the Business Systems Development Process is but one way of applying the methodology, it clearly shows Catalysis's general approach to systems development. Analysis usually starts by modeling the problem domain as a collection of *types* (classes), with their own inter-relationships and interactions. Then the system is added to the context, treated like another problem domain *type*, whose state (the types it contains), operations (functionality) and behaviour are carefully modeled. The focus is then shifted into the system itself, modeling it as a collection of components, again with their own inter-relationships and interactions. Finally, each component is modeled as a collection of implementation-level classes, interfaces and off-the-shelf components, yet again with their own inter-relationships and interactions.

This sort of gradual refinement is an essential practice in Catalysis. So is the recursive (fractal) modeling approach: applying the same view (constituents, their inter-relationships and interactions) by the same set of diagrams at each and every level of refinement. These two practices are at the heart of the Catalysis process, yet there are many ways of actually applying them to a project: they can be applied sequentially, or in an iterative-incremental fashion, or according to any other development lifecycle deemed appropriate by the developers.

To help developers apply the methodology, Catalysis proposes four process patterns for four different kinds of projects:

1. *Object Development from Scratch*: for when there is no existing system.
2. *Reengineering*: for when the objective is to improve an existing system.
3. *Business Process Improvement*: for applying object technology to organizations and systems other than software.
4. *Separate Middleware from Business Components*: for handling legacy systems as well as for insulating a system from certain changes in technology.

Catalysis proposes detailed sets of activities for each pattern and guidelines for their application [D'Souza and Wills 1998].

3.3.3.3 OPEN (1996)

OPEN (Object-oriented Process, Environment, and Notation) was first introduced in 1996 as the result of the integration of four methodologies: MOSES, SOMA, Synthesis and Firesmith [Henderson-Sellers and Graham 1996]. This initial version of OPEN was later deeply influenced by BON and OOram [Reenskaug et al. 1996]. The advent of UML compelled the OPEN Consortium (an international group of experts and tool-vendors that maintains OPEN) to tailor it in order to catch up with the new wave of standardization. However, OPEN has kept its own modeling language, OML (OPEN Modeling Language), as a more suitable alternative to UML in terms of compatibility with the specific modeling needs in OPEN [Graham et al. 1997].

OPEN is presented as a framework called OPF (OPEN Process Framework). OPF is a process metamodel defining five classes of components and guidelines for constructing customized OPEN processes (Figure 18). OPEN also contains a component library from which individual component instances can be selected and put together to create a specific process instance tailored to fit the project in hand. The OPF component classes and the instantiation method for constructing OPEN processes are discussed in the following sections.

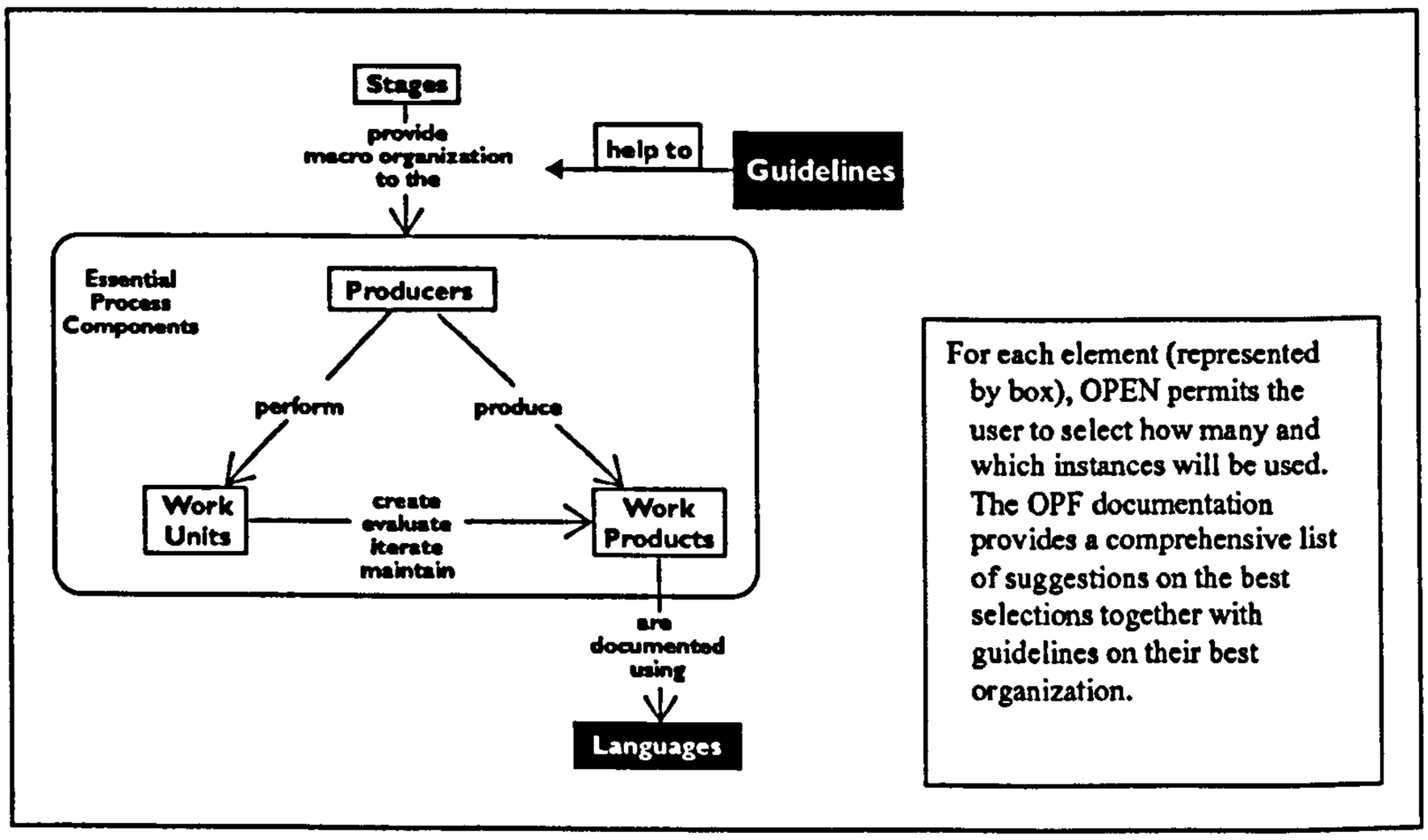

Figure 18. The OPF components (OPEN) [OPEN Consortium 2000]

OPF Component Classes (OPEN)

As depicted in Figure 18, OPF consists of five major classes of components:

1. *Work Products*: any significant thing of value (document, diagram, model, class, application) developed during the project.
2. *Languages*: the media used to document work products, such as natural languages, modeling languages such as UML or OML, and implementation languages such as Java, SQL, or CORBA-IDL.
3. *Producers*: active entities (human or nonhuman) that develop the work products.
4. *Work Units*: operations that are performed by producers when developing work products. One or more *producers* develop a *work product* during the execution of one or more *work units*. Work units are of three types:
    a. *Activity*: a major work unit consisting of a related collection of jobs that produce a set of work products. Activities are coarse-grained descriptions of *what* needs to be done. Some important instances defined by OPEN are: Project Initiation, Requirements Engineering, Analysis and Model Refinement, Project Planning, and Build (evolutionary development or OOA/OOD/OOP together with verification and validation, user review and consolidation).
    b. *Task*: the smallest atomic unit of work. Tasks are small-scale jobs associated with and comprising the activities, resulting in the creation, modification, or evaluation of one or more work products.
    c. *Technique*: defines *how* the jobs are to be done. Techniques are ways of doing the tasks and activities.
5. *Stages*: durations or points in time that provide a high-level organization to the work units. Stages are of two types:
    a. *Milestone (Instantaneous Stage)*: a point in time marking the occurrence of an event.
    b. *Stage with Duration*: The high-level periods during which work units are performed. There are seven significant types:
        i. *Project*: covering a single individual project.
        ii. *Cycle*: Iterative set of work units varying in span and scope from short-span cycles (such as Development

Cycles and Delivery Cycles) to the long-span *Lifecycle*, which is a sequence of phases covering the whole temporal extent of a significant engineering effort. The following types of lifecycle have been defined in OPF:

1. *Project Development Lifecycle*: the duration over which the project is conceived and products are constructed.
2. *Project Lifecycle*: covering the project development lifecycle and the maintenance stage.
3. *Delivery Lifecycle*: focusing on the repetitive delivery of product versions.
4. *Enterprise Lifecycle*: in which business modeling and business re-engineering occur.
5. *Programme Lifecycle*: larger in scale than the project lifecycle, this is a cycle related to a *programme* of projects, and as such is the sum of all the relevant project life cycles *plus* a Strategy Phase in which high-level business planning across all projects is performed.

iii. *Phase*: a stage of development consisting of a sequence of one or more *builds*, *releases* and *deployments* (explained later in this section). Instances of phase are assigned to one or more of the above life cycles. Seven phases have been defined in OPF:

1. *Inception*: during which the development is started and appropriate preparations are made.
2. *Construction*: during which the work products are developed and prepared for release.
3. *Usage*: during which the work products are released to the user organization and put into service.
4. *Retirement*: when the software is withdrawn from service.

5. *Strategy*: in which cross-project considerations at the business level are analyzed.
6. *Business Modeling*: in which a modeling technique is applied to model the business itself (irrespective of whether or not software has any role in the business).
7. *Business Reengineering*: in which the processes in the business are analyzed and reconsidered.

iv. *Workflow*: a sequence of tasks during which producers collaborate to produce a work product. Examples of workflows defined in OPF are requirements and architectural workflows such as: Vision Statement Workflow, System Requirements Specification Workflow, Software Requirements Specification Workflow, and Software Architecture Document Workflow.

v. *Build*: during which tasks are undertaken. Builds are the only kinds of *stage* that occur within the Inception Phase; in other phases they are generally complemented by releases, deployments, and milestones.

vi. *Release*: in which the results of a build are delivered to the user.

vii. *Deployment*: when the user receives the product and puts it into service.

Process Instantiation (OPEN)

As shown in Figure 19, the following tasks are performed (through applying the guidelines proposed by OPF) in order to instantiate, tailor and extend an OPEN process [Firesmith and Henderson-Sellers 2001]:

1. Instantiating the OPEN library of predefined component-classes to produce actual process components.
2. Choosing the most suitable process components from the set of instantiated components.
3. Adjusting the fine detail inside the chosen process components.

4. Extending the existing class library of predefined process components to enhance reusability.

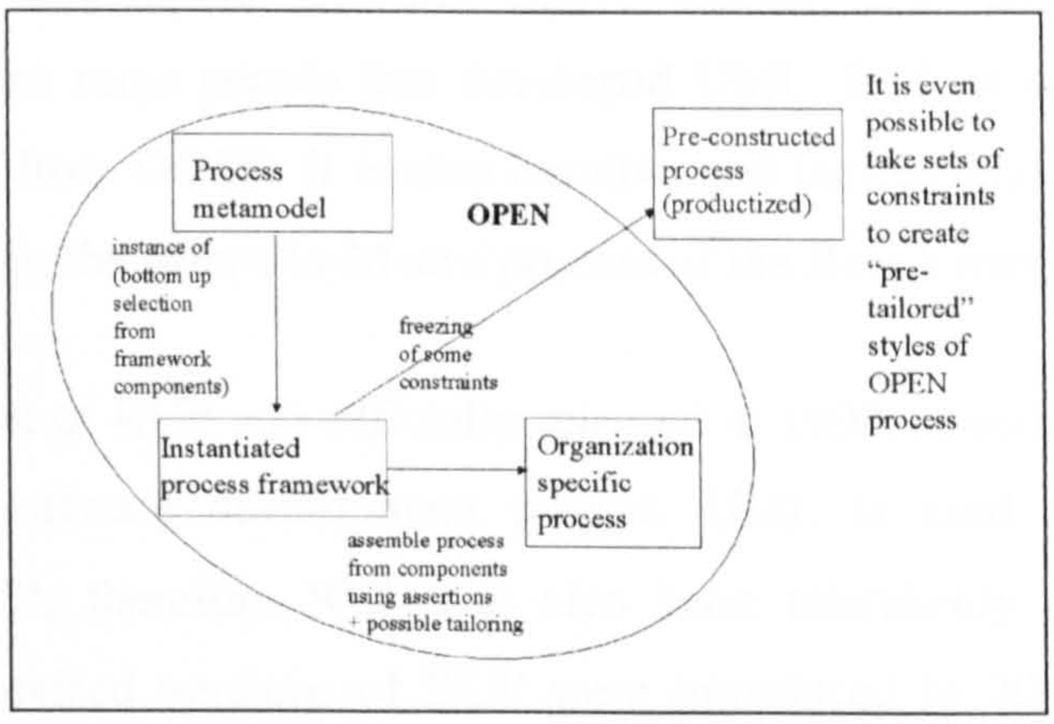

Figure 19. Using the OPF to instantiate, tailor and extend a process (OPEN) [OPEN Consortium 2000]

Figure 20 shows an example of an instantiated OPEN process. This process is usually (and wrongly) referred to as "The OPEN Process", yet it is just one instance (though fairly general) of the processes that can be constructed in OPEN.

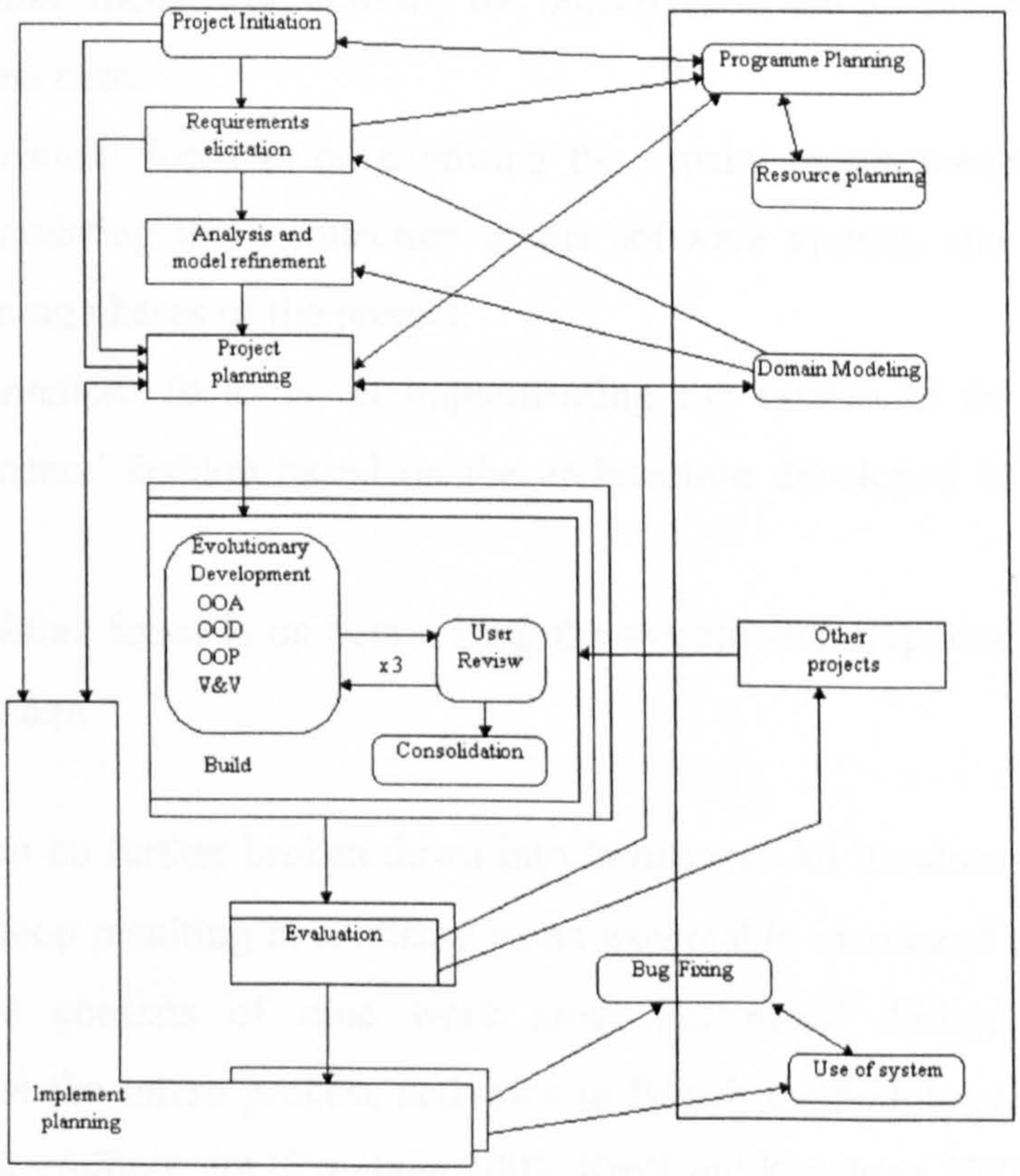

Figure 20. Example of an instantiated OPEN process [Graham et al. 1997]

3.3.3.4 RUP/USDP (1998, 1999, 2000, 2003)

RUP (Rational Unified Process) was developed at Rational Corporation by the three principal developers of the OMT, Booch and OOSE (Objectory) methodologies, the same people that developed UML. RUP is use case driven, a feature inherited from OOSE. It is also iterative and incremental, with the overall process resembling the Micro-in-Macro process of the Booch methodology.

The initial version of RUP was officially released in 1998, covering all the generic activities in a software development project. UML is used as the modeling language in RUP; therefore RUP has also been mistakenly called the UML Methodology. Revised versions of RUP were introduced in 2000 and 2003, the most recent of which will be described in this section [Kruchten 2003]. The developers of RUP introduced a non-proprietary, somewhat less complex variant of RUP, called USDP (Unified Software Development Process) in 1999 [Jacobson et al. 1999].

The overall RUP development *cycle* consists of four *phases* [Kruchten 2003]:

1. *Inception*: focus is on defining the objectives of the project, especially the business case.
2. *Elaboration*: focus is on capturing the crucial requirements, developing and validating the architecture of the software system, and planning the remaining phases of the project.
3. *Construction*: focus is on implementing the system in an iterative and incremental fashion based on the architecture developed in the previous phase.
4. *Transition*: focus is on beta-testing the system and preparing for releasing the system.

Each phase can be further broken down into *iterations*. An iteration is a complete development loop resulting in a release of an executable increment to the system. Each iteration consists of nine work areas performed during the iteration (somewhat like the micro process activities in Booch methodology). These work areas, called *disciplines,* are [Kruchten 2003, Kroll and Kruchten 2003]:

1. *Business Modeling*: concerned with describing business processes and the internal structure of a business in order to understand the business and determine the requirements for software systems to be built for the business. A *Business Use Case Model* and a *Business Object Model* are developed as the result of this discipline.
2. *Requirements Management*: concerned with eliciting, organizing, and documenting requirements. The *Use Case Model* is produced as the result.
3. *Analysis and Design*: concerned with creating the architecture and the design of the software system. This discipline results in a *Design Model* and optionally an *Analysis Model*. The design model consists of design classes structured into design packages and design subsystems with well defined interfaces, representing what will become components in the implementation. It also contains descriptions of how objects of these design classes collaborate to perform use cases.
4. *Implementation*: concerned with writing and debugging source code, unit testing, and build management. Source code files, executables, and supportive files are produced.
5. *Test*: concerned with integration-, system- and acceptance testing.
6. *Deployment*: concerned with packaging the software, creating installation scripts, writing end-user documentation and other tasks needed to make the software available to its end-users.
7. *Project Management*: concerned with project planning, scheduling and control.
8. *Configuration and Change Management*: concerned with version- and release management and change-request management.
9. *Environment*: concerned with adapting the process to the needs of a project or an organization, and selecting, introducing and supporting development tools.

Figure 21 shows how the disciplines are performed during the iterations.

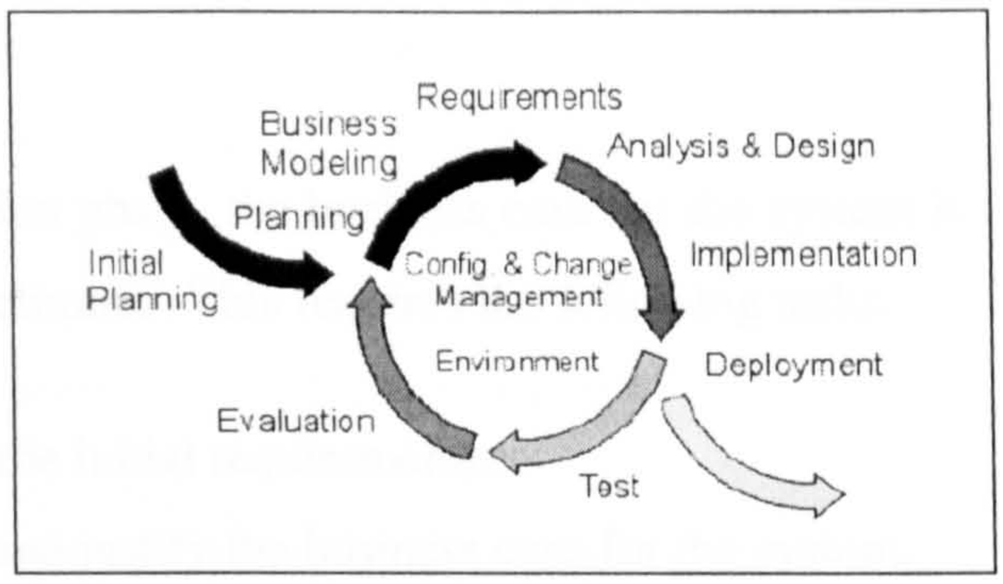

Figure 21. Disciplines in iterations (RUP) [Kruchten 2003]

The disciplines do not have equal emphasis during an iteration: the amount of effort expended on a discipline depends on the phase in which the iteration is taking place. Business modeling and requirement take a lot of emphasis during earlier phases, whereas during later phases, most of the effort is put into deployment and testing. Figure 22 shows the phases, disciplines and example iterations in the RUP lifecycle model, and shows the relative amount of emphasis put on each discipline during the iterations and phases.

For each discipline, RUP defines a set of *artefacts* (work products), *activities* (units of work on the artefacts), and *roles* (responsibilities taken on by development team members).

A brief description of each of the phases in RUP and the artefacts produced is given in the next sections.

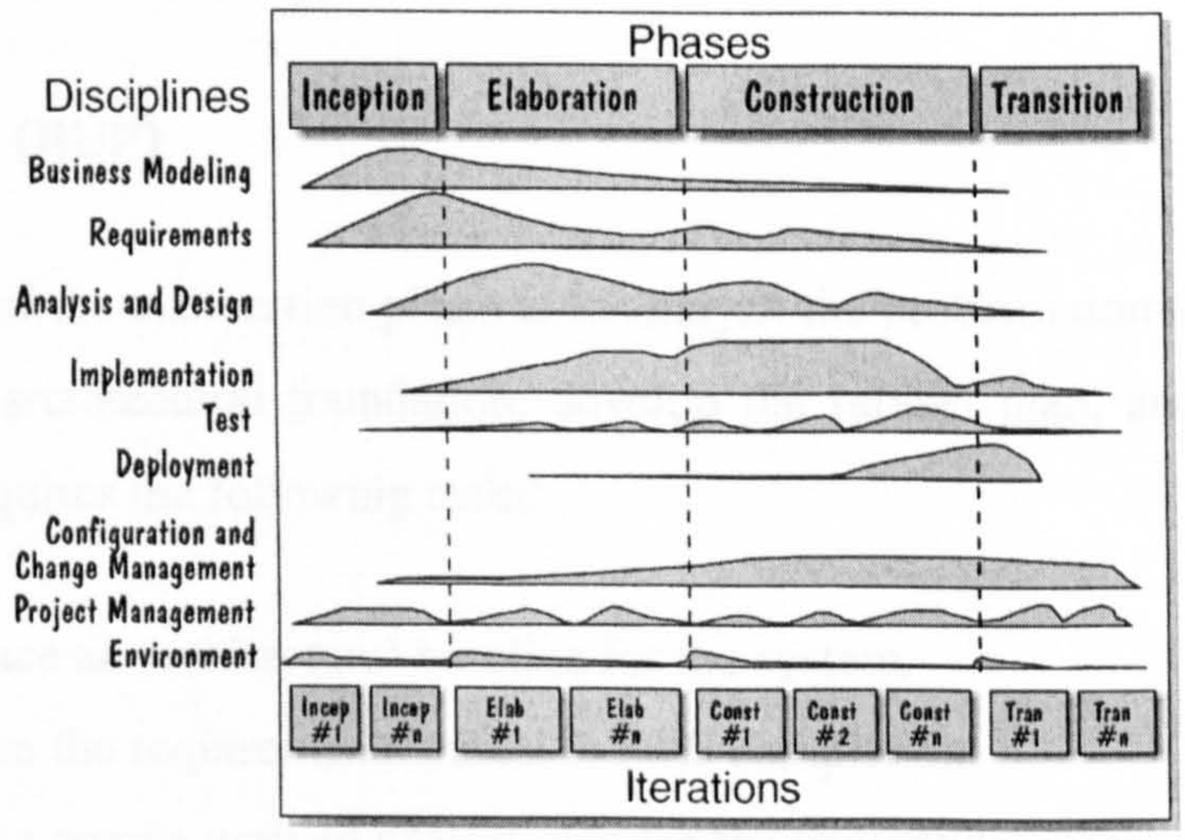

Figure 22. A typical RUP lifecycle model [Kroll and Kruchten 2003]

Inception (RUP)

During the inception phase, the business case for the system is established and the project scope is delimited. This requires the following tasks:

1. Describe the initial requirements.
2. Develop and justify the business case for the system.
3. Determine the scope of the system.
4. Identify the people, organizations, and external systems that will interact with the system.
5. Develop initial risk assessment, schedule, and estimates.
6. Configure the initial system architecture.

The following artefacts are usually produced during this phase:

- A vision document: a general description of the core project's requirements, key features, and main constraints.
- An initial use case model (10% -20% complete).
- An initial project glossary.
- An initial business case: business context, success criteria, and financial forecast.
- An initial risk assessment.
- A project plan.
- A business model (optional).
- A number of prototypes.

Elaboration (RUP)

The purpose of the elaboration phase is to analyze the problem domain, establish a system-level architectural foundation, develop the project plan, and mitigate the risks. This requires the following tasks:

1. Produce an architectural baseline for the system.
2. Evolve the requirements model to 80% completion.
3. Draft a coarse-grained project plan for the construction phase.

4. Ensure that critical tools, processes, standards, and guidelines have been put in place for the construction phase.
5. Understand and eliminate high-priority risks of the project.

The following artefacts are usually produced during this phase:

- A use case model (at least 80% complete) — with all use cases and actors identified, and most use case descriptions developed.
- Supplementary requirements capturing the non-functional requirements and those requirements that are not associated with any specific use case.
- A software architecture description.
- An executable architectural prototype.
- A revised risk list and a revised business case.
- A development plan for the overall project, including the coarse-grained construction plan, showing iterations and evaluation criteria for each iteration.
- An updated development case specifying the process to be used.
- A preliminary user manual (optional).

Construction (RUP)

During the construction phase, the remaining components and features are developed and integrated into the product, and all features are thoroughly tested. This requires the following tasks:

1. Describe the remaining requirements.
2. Develop the design of the system.
3. Ensure that the system meets the needs of its users and fits into the organization's overall system configuration.
4. Complete component development and testing, including both the software product and its documentation.
5. Minimize development costs by optimizing resources.
6. Achieve adequate quality.
7. Develop useful versions of the system.

The following artefacts are usually produced during this phase:

- The software product.
- The user manuals.
- A description of the current release.

Transition (RUP)

The purpose of the transition phase is to transition the software product to the user community. This requires the following tasks:

1. Test and validate the complete system.
2. Integrate the system with existing systems.
3. Convert legacy databases and systems to support the new release.
4. Train the users of the new system.
5. Deploy the new system into production.

The following artefacts are usually produced during this phase:

- Final product baseline of the system.
- Training materials for the system.
- Documentation, including user manuals, support documentation, and operations documentation.

3.3.3.5 EUP (2000, 2005)

EUP (Enterprise Unified Process) was introduced by Ambler and Constantine in 2000 as an extended variant of RUP. A revised and refactored version was introduced in 2005 [Ambler et al. 2005]. The developers believe that RUP suffers from serious drawbacks (which they claim to have corrected in EUP), namely [Ambler and Constantine 2000a]:

- RUP does not cover system support and eventual retirement.
- RUP does not explicitly support organization-wide infrastructure development.

- The iterative nature of RUP is both a strength and a weakness, since the iterative nature of the lifecycle is hard to grasp for many experienced developers.
- Rational's approach to developing RUP was initially tools-driven; hence the resulting process is not sufficient for the needs of developers.

The lifecycle model of EUP is shown in Figure 23. It extends RUP by adding two new *phases* and two new *disciplines* (one of which was further broken down into seven disciplines in the 2005 version of the methodology), and also by extending the activities in some of the old disciplines.

EUP's viewpoint to modeling is also somewhat different from RUP. Whereas RUP advocates adherence to UML, EUP makes use of some older modeling notations too. An example of this is the use of Data Flow Diagrams for business modeling. Furthermore, EUP stresses that use cases are not enough for modeling the requirements; consequently, use cases in EUP do not have the pivotal role they have in RUP.

The following sections briefly describe the additions and changes EUP has made to RUP.

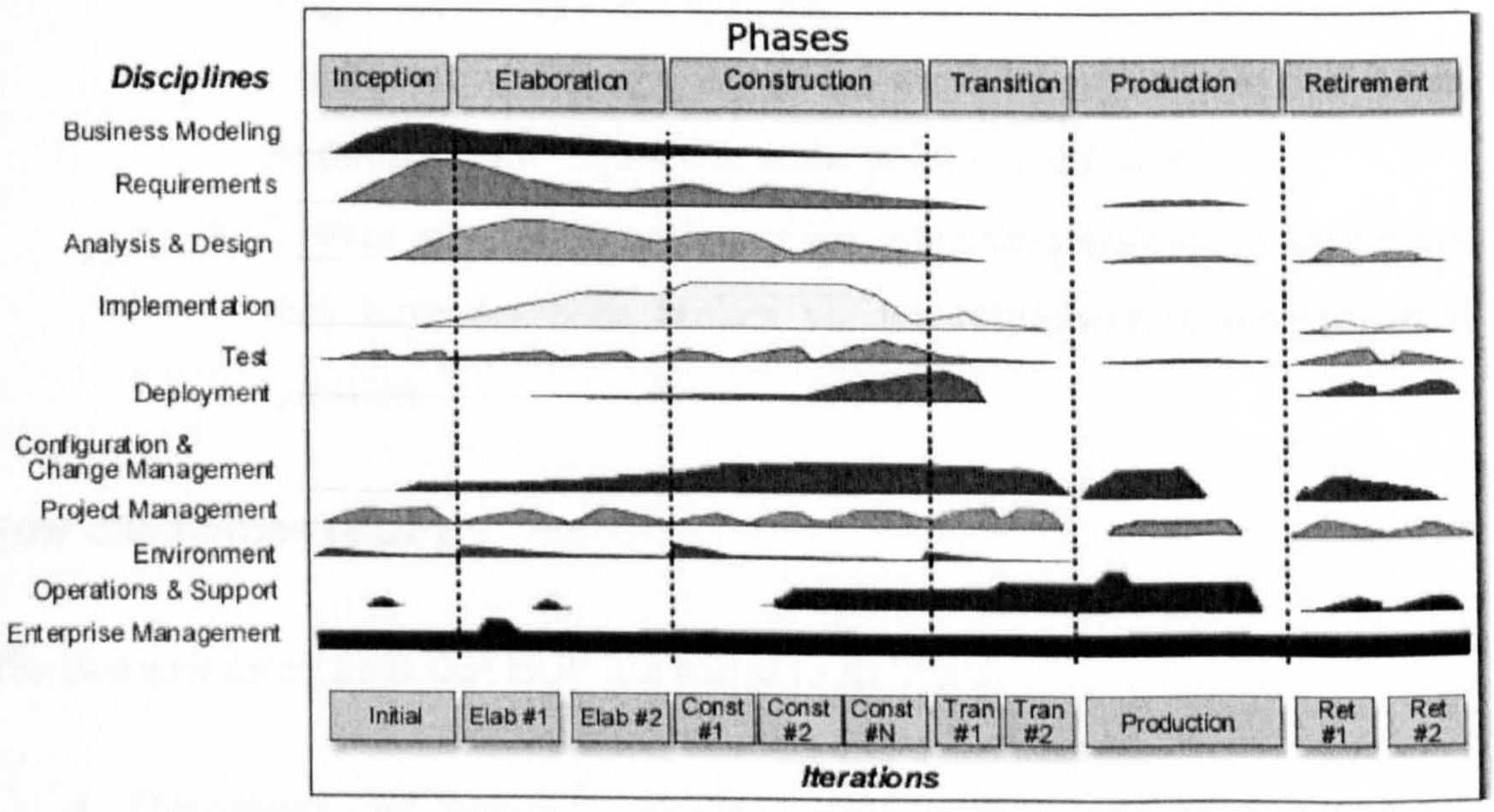

Figure 23. A typical EUP lifecycle model [Ambler and Constantine 2000a]

New phases (EUP)

The two new phases that EUP has added to RUP are:

- *Production*: added as the fifth phase, the focus of this phase is on keeping the software in production until it is either replaced with a new version (by executing the lifecycle all over again), or retired and removed. There are no iterations during this phase. This phase is somewhat similar to the maintenance phase in the generic software development lifecycle, in that it is mainly concerned with the operation and support of the system; but unlike classic maintenance, any need for changing the system (even a bug fix) will result in the reinitiation of the development cycle [Ambler and Constantine 2002].
- *Retirement*: added in 2002 as the sixth phase, the focus of this phase is on the careful removal of a system from production, either because it is no longer needed or is being replaced. This typically includes [Ambler 2005]:
  - Identification of the existing system's coupling to other systems.
  - Redesign and rework of other systems so that they no longer rely on the system being retired.
  - Transformation of existing legacy data.
  - Archival of data previously maintained by the system that is no longer needed by other systems.
  - Configuration management of the removed software so that it may be reinstalled if required at some point in the future.
  - System integration testing of the remaining systems to ensure that they have not been broken via the retirement of the system in question.

New disciplines (EUP)

The two new disciplines that EUP has added to RUP are:

- *Operations and Support*: concerned with issues related to operating and supporting the system, typically associated with the maintenance phase of the generic software development lifecycle. This discipline, however, spans several phases, not only the production phase. During the

construction phase, and perhaps as early as the elaboration phase, the development of operations and support plans, documents, and training manuals is initiated. These artefacts are enhanced and perfected during the transition phase, where the discipline will also include the training of the operations and support staff. During the production and retirement phases, the discipline covers classic maintenance activities: the operations staff will keep the software running, performing necessary backups and batch jobs, and the support staff will communicate with the users to help them work with the software [Ambler and Constantine 2000a,b,c].

- *Enterprise Management*: concerned with the activities required to create, evolve, and maintain the organization's cross-system artefacts such as the organization-wide models (requirements and architecture), software process, standards, guidelines, and the reusable artefacts [Ambler and Constantine 2000a,b,c]. The Enterprise Management discipline was broken down into seven disciplines in the 2005 version of the methodology [Ambler et al. 2005], namely: *Enterprise Business Modeling*, *Portfolio Management*, *People Management*, *Enterprise Architecture*, *Strategic Reuse*, *Enterprise Administration*, and *Software Process Improvement*.

Modified disciplines (EUP)

In EUP, several changes have been made to RUP disciplines, including:

- The *Test* discipline has been expanded to include requirements validation during the inception phase, using techniques such as walkthroughs, inspections, and scenario testing [Ambler and Constantine 2000a].
- The *Deployment* discipline in EUP has been augmented by deployment modeling activities (which in RUP are a part of the analysis-and-design discipline). The EUP also advocates starting deployment planning as early as possible in the lifecycle. As a result of these two changes, the deployment discipline in EUP has been extended into the inception and elaboration phases [Ambler and Constantine 2000a,b].
- The *Environment* discipline has been updated to include the work necessary to define the *Production* environment [Ambler and Constantine 2002].

- The *Configuration and Change Management* and *Project Management* disciplines are extended into the new production and retirement phases. Furthermore, new features have been added to the project management discipline, including metric management, subcontractor management and people management [Ambler and Constantine 2002, Ambler 2005].

3.3.3.6 FOOM (2001)

Introduced in 2001 by Shoval and Kabeli, FOOM (Functional and Object-Oriented Methodology) [Shoval and Kabeli 2001] is an object-oriented variant of Shoval's ADISSA methodology of 1988 [Shoval 1988]. ADISSA (Architectural Design of Information Systems based on Structured Analysis) was an attempt to ameliorate the shortcomings of the classical, process-oriented Structured-Analysis/Structured-Design (SA/SD) methodology through introduction of the *transaction* – a notion very similar to the use case – as the basis for the design process. FOOM, in turn, strives to combine the classical process-oriented approach (as prescribed by ADISSA) with the object-oriented paradigm, very much in the tradition of well-established "hybrid" methodologies such as OMT.

The FOOM process consists of the following phases:

1. *Analysis*: concerned with requirements elicitation and problem-domain modeling, this phase consists of two activities, performed in parallel or iteratively:
    1.1. *Data Modeling*: with the focus on identifying and modeling the class structure of the problem domain.
    1.2. *Functional Analysis*: with the focus on identifying and modeling the functional requirements of the system.
2. *Design*: concerned with designing implementation-specific classes and adding structural and behavioural detail to the models, this phase consists of the following stages:
    2.1. *Defining Basic Methods*: with the focus on specifying primitive operations for the classes.
    2.2. *Top-level Design of Application Transactions*: with the focus on identifying *transactions*, which are intra-system chains of

processes performed in response to stimuli from outside the system; as such, each transaction is in fact a unit of functionality performed by the system in realization of its functional requirements. Structured descriptions of the identified transactions are also generated, to be extensively used during later stages of the design phase.

2.3. *Interface Design*: with the focus on designing a menu-based user interface for the system. Suitable classes are then defined in order to implement these menus.

2.4. *Input/Output Design*: with the focus on designing the input forms/screens and the output reports/screens of the system, and defining classes for implementing them.

2.5. *Design of System Behaviour*: with the focus on providing detailed specifications for the transactions, and elaborating on object interactions and operations of the classes.

3. *Implementation*: with the focus on object-oriented coding and testing of the system.

FOOM is mainly targeted at data-intensive information systems. This explains its lack of provision for behavioural modeling during systems analysis. Targeting data-intensive systems has also resulted in a slack attitude towards behavioural design of the system; many of the activities prescribed in the Design-of-System-Behaviour stage are too simplistic to be of any practical use in developing process-intensive systems.

A short description of each of the first two phases (*Analysis* and *Design*) is given in the next sections. The developers of the methodology do not propose a specific procedure for the *Implementation* phase, merely stating that the system is implemented based on the models and specifications produced during the design phase (especially the behavioural specifications), using any common object-oriented programming language.

Analysis (FOOM)

The following activities are performed in this phase:

1. *Data Modeling* – Problem-domain classes are identified along with their attributes and relationships, with the results modeled in a *Class Diagram.* This initial class diagram does not include the operations (*methods*) of the classes, as these are to be added during the design phase. The classes identified, therefore, are in fact *data* classes representing the data content of the problem domain.

2. *Functional Analysis* – Functional requirements of the system are elicited and modeled in a hierarchy of *Object-Oriented Data Flow Diagrams* (OO-DFDs). What makes these diagrams different from traditional DFDs is that *classes* replace traditional *data stores*. Furthermore, the traditional notion of external entities has been expanded to include *time entities*, *real-time entities* and *communication entities* in addition to ordinary *user entities*. *Time entities* act as modeling proxies for clocks, generating time signals at specific points in time or during predetermined time-intervals, whereas *real-time entities* act as generators of asynchronous sensor events from the system environment, and *communication entities* represent other systems interacting with our system via communication channels.

The two activities complement each other: not only are their products bound together by common elements (data classes), but they also contribute to each other in the sense that each activity provides an insight into the problem domain that can then be used for enhancing the course of the other activity. Therefore, the methodology prescribes that these activities be performed either in parallel or iteratively (with the analysis team alternating between the two); more recently, it has been suggested that, although the two activities should overlap, starting with data modeling is preferable [Kabeli and Shoval 2003].

Design (FOOM)

The design phase consists of the following stages:

1. *Defining Basic Methods* – primitive methods are attached to each data class in the initial class diagram. These methods, which are fairly independent from the business logic of the system, are of two types:

1.1. *Elementary Methods*, which are the basic methods typically found in classes, namely: *construct-object* (instantiate), *destruct-object*, *get-attribute(s)*, and *change-attribute(s)*.

1.2. *Relationship/Integrity Methods*, which are derived from structural relationships between classes and are intended to manage the links between the objects at run-time and perform referential integrity checks. Integrity checks should take into account the relationship types that the classes are involved in, and the cardinality constraints of these relationships. There are five types of Relationship/Integrity methods generally defined for each relationship a class is involved in, namely: *initialize-connections* (on object construction), *break-all-connections* (on object destruction), *connect-to-object* (via relationship), *disconnect*, and *reconnect*.

2. *Top-level Design of Application Transactions* – Very much like the modern-day *use case*, a *transaction* is a unit of functionality performed by the system in direct support of an external entity (as categorized in OO-DFD semantics). A transaction is triggered (initiated) as a result of an *event*. Events in FOOM are of four types: *user events* (originating from user entities), *communication events* (originating from communication entities), *time events* (originating from time entities), and *real-time events* (originating from real-time entities). Top-level design of the transactions is performed in the following steps:

2.1. *Identification of transactions*: the transactions of the system are identified from the hierarchy of OO-DFDs constructed during the analysis phase. The OO-DFD hierarchy is traversed in order to isolate the transactions, each of which consists of one or more chained leaf processes, and the data classes and external entities connected to them. Generally each transaction has one or more external entities at one end and data classes and/or external entities at the other.

2.2. *Description of transactions*: a top-level transaction description is provided in a structured language referring to all the components of the transaction: every data-flow from or to an external entity is

translated to an "Input from..." or "Output to..." line; every data-flow from or to a data class is translated to a "Read from..." or "Write to..." line; every data flow between two processes translates to a "Move from... to..." line; and every process in the transaction translates into an "Execute function..." line. The process logic of the transaction is expressed by using standard structured programming constructs. The top-level descriptions thus produced will be extensively used during later stages of design as a basis for designing the application-specific features of the system.

2.3. *Definition of the "Transaction" class*: an abstract "Transaction" class is added to the class diagram. Acting as a *utility class*, the "Transaction" class will encapsulate operations for implementing the process logic of complex transactions; that is, transactions that are not deemed suitable to be assigned to ordinary classes due to their over-complexity are put in this class as operations. Operations of this class will be defined during the last stage of design.

3. *Interface Design* – In this stage, the OO-DFD hierarchy is traversed in a top-down fashion in order to produce the menu-based interface of the system: a main menu, initially empty, is defined for the system; for each process at the topmost level of the hierarchy that is connected to a user entity, a corresponding menu-item is defined and added to the main menu; at any level of the OO-DFD hierarchy, for every non-leaf process connected to a user entity, a corresponding submenu is defined and initialized as empty, and for every process (leaf or non-leaf) that is connected to a user entity a corresponding menu-item is defined and added to its parent-process's submenu. The menu tree thus derived is then refined into the user-interface of the system. The leaf items in this tree correspond to leaf processes connected to user entities, and will invoke a system transaction when selected at run-time. In order to realize this interface, a "Menu" class is defined and added to the class diagram of the system. Instances of this class will be the run-time menus, with their items saved as attribute values.

4. *Input/Output Design* – The top-level descriptions of the transactions are used for determining what input forms/screens and output reports/screens should be designed: an input form/screen will be designed for each "Input from" line appearing in the transaction descriptions, and an output report/screen will be designed for each "Output to" line. Two new classes, the "Form" class for the inputs and the "Report" class for the outputs, are then added to the class diagram. The actual screens, forms, and reports are instances of these classes, with the titles and data-fields stored as attribute values.
5. *Design of System Behaviour* – This stage of the design phase produces the main behavioural specifications of the system. The top-level descriptions of the transactions are used as a basis for identifying and detailing the main application-specific operations of the classes as well as the object interactions (message-passing chains) that implement the transactions of the system. This process typically involves the following activities:

    5.1. *Identification of operations*: the top-level descriptions are refined so as to include details on the operations in charge of implementing the expected functionality, as well as the classes/objects to which these operations belong. Transaction specifications thus refined show the full object-oriented process logic of the transactions in terms of run-time message interchange among objects. The following conversions and mappings are typically performed:

    - Each Input/Output line is converted into a message to a corresponding operation in the relevant Report/Form object.
    - Each Read/Write line is translated into a message to the corresponding basic function in the relevant data class.
    - Each Execute-Function line is converted to a message-passing chain consisting of one or more messages to specific operations of particular classes. These operations may be basic operations already defined, or new application-specific operations that should be assigned to appropriate classes. The design team decides on how to

realize the expected functionality of each Execute-Function line as a message passing chain, and in doing so identifies new operations for the classes involved. It should also make sure that the message passing logic is incorporated in each of the participating classes.

Detailed signatures are then defined for all the operations. The detailed descriptions of the transactions are ultimately translated into pseudo-code, in which the process logic of each transaction (the sequence of the message interchange, and the iterations/conditions involved) is expressed by using standard structured-programming constructs. In addition, for every transaction that involves chains of message-interchange among objects, a *Message Diagram* (identical to the UML collaboration diagram) is produced; these diagrams help further clarify the process logic of the transactions.

5.2. *Transaction assignment*: classes are put in charge of fully executing, or initiating/directing the execution of the transactions for which detailed specifications were produced in the previous substage. Depending on the complexity of its internal process logic (excluding Input/Output and Read/Write messages), each transaction undergoes one of the following:

- Transactions that have a processing scope confined to instances of a single class are assigned to that class as an operation. Triggering the transaction will result in the invocation of the corresponding operation, which will "execute" the transaction in its entirety.
- Transactions with moderate processing complexity involving a message-passing chain among instances of different classes are assigned to a participating class as an operation. Instances of this class will thus be able to act as chain "initiators"; the overall process logic is distributed among the participant classes, with the participating objects *knowing* to which object the next message should be directed.

- Transactions with complex process logics involving many classes are deemed not suitable to be assigned to any of their participant classes. Such transactions are assigned to the abstract "Transaction" class as an operation. The high-level process logic of the transaction is centralized in the operation, thus putting it in charge of orchestrating the processing through "directing" the invocations (analogous to a "main" module).

Every operation executing, initiating or directing a *user* transaction is linked to its corresponding menu item in the relevant Menu object, so that selection of the item by the user at run-time will activate the proper operation. Provision should also be made for operations that execute/initiate/direct other types of transactions (*time*, *real-time*, and *communication*) to be invoked upon occurrence of their pertinent trigger events.

5.3. *Detailed specification of operations*: pseudo-code descriptions are produced for all significant operations. The pseudo-code specifications of the methods (operation bodies) are intended to facilitate the actual coding of the system during the Implementation phase.

3.3.4 Methodologies: Agile

Enthusiasm over agile development has been such that the methodology war of the early 90s has been more or less repeated over agile methodologies. Not only have numerous variants of prominent agile methodologies emerged, but agile variants of older methodologies have also been proposed. The agile methodologies selected for inclusion in this analysis are the main contenders, widely recognized as the torchbearers of the agile movement.

3.3.4.1 DSDM (1995, 2003)

DSDM (Dynamic Systems Development Method) was first introduced in 1995 by a consortium of UK companies. Motivated by an ever-increasing need for a standard, generally-accepted RAD (Rapid Application Development) methodology, the

consortium produced DSDM as an iterative-incremental generic framework based on evolutionary prototyping and principles that are nowadays attributed to agile development [DSDM Consortium 2003]. Starting with 16 UK companies, the consortium now has more than 1000 members, including industry giants such as IBM, Microsoft and Siemens; it should not be surprising, then, that the framework proposed by DSDM is now considered the de facto standard for RAD.

The latest version of the DSDM process consists of seven phases [DSDM Consortium 2003]; the first and last ones, though, are not considered main phases, since they are not considered part of the project itself:

1. *Pre-project*: with the focus on providing the necessary resources for starting the project, along with a plan for the next immediate phase, i.e. the feasibility study.
2. *Project-proper*, during which the five main phases of the DSDM are applied; the first two sequentially at the start of the project, and the remaining three as interwoven cycles (Figure 24):
   - 2.1. *Sequential Phases*: primarily concerned with studying the business domain and performing a preliminary analysis of the system, these short phases set the stage for the actual development of the system:
     - 2.1.1. *Feasibility Study*: analogous to the classic feasibility analysis, albeit with a special focus on analyzing the suitability of DSDM for the project, and coming up with an outline plan for the subsequent phases.
     - 2.1.2. *Business Study*: with the focus on identifying system-relevant processes and information entities in the business domain, defining and prioritizing the high-level requirements of the system, developing the system architecture, and producing a development plan.
   - 2.2. *Iterative Phases (The Development Cycle)*: based on the high-level knowledge acquired during the business study phase, the three iterative phases iteratively and incrementally analyze,

design, code and deploy the system through evolutionary prototyping:

2.2.1. *Functional Model Iteration*: with the focus on selecting requirements according to their priority, and performing detailed analysis and modeling of the selected requirements through prototyping.

2.2.2. *Design-and-Build Iteration*: with the focus on evolving the prototypes into final deliverable increments of the system.

2.2.3. *Implementation*: with the focus on deploying the deliverable increments into the operational environment, and reviewing and validating the system built so far.

3. *Post-project*: with the focus on system maintenance, which as in most other iterative-incremental methods, is applied through further iterations of the main phases.

DSDM does not prescribe a specific order for the execution of the iterative phases in the overall process: it is true that prototypes should undergo the three phases in the order specified above, yet as shown in Figure 24, the three iterative phases themselves form an outer interwoven cycle (hence the name "Development Cycle"). The selection of the number of iterations in each cycle, and the way the iterations should interact, is completely dependent on the project and up to the development team to decide. Furthermore, the introduction of multiple development sub-teams working in parallel enables the phases to overlap, adding another configurable dimension to the process. Since all of this enables the developers to tailor the process to fit the project in hand, DSDM is referred to as a configurable *process framework*, rather than a *methodology*.

In customizing the process framework, the development team also has to set up a strict time-constrained plan for the development. In DSDM, stringent constraints are set on *time* and *resources*, leaving the requirements (*functionality*) as the only variable parameter of the project (DSDM is thus deemed especially suitable for projects with highly volatile requirements); this is in contrast to traditional

methods, in which time and resources are allowed to vary, while functionality is fixed.

In DSDM, time constraints are set up using time frames called *time-boxes*. A fixed completion date is set for the overall project, thereby defining the overall time-box in which the project is to be done. During the business study phase, shorter time-boxes of two to six weeks are nested inside this overall time-box, setting temporal boundaries for development cycles and/or iterations. Each time-box is assigned a fixed end-date and a prioritized set of requirements. End-dates are not movable, and lower priority requirements are to be sacrificed if the time-box does not allow work to be done on them, in which case they might be taken on in later time-boxes. Each time-box is to produce tangible artefacts, and is therefore the basic unit for project monitoring and control.

Like other agile development methods, DSDM is based on a number of principles, the most important of which are active user involvement, frequent deliveries, empowered development teams, reversibility of changes, and testing in all phases of the project.

The following sections contain brief descriptions of the tasks performed in each of the five main phases of DSDM.

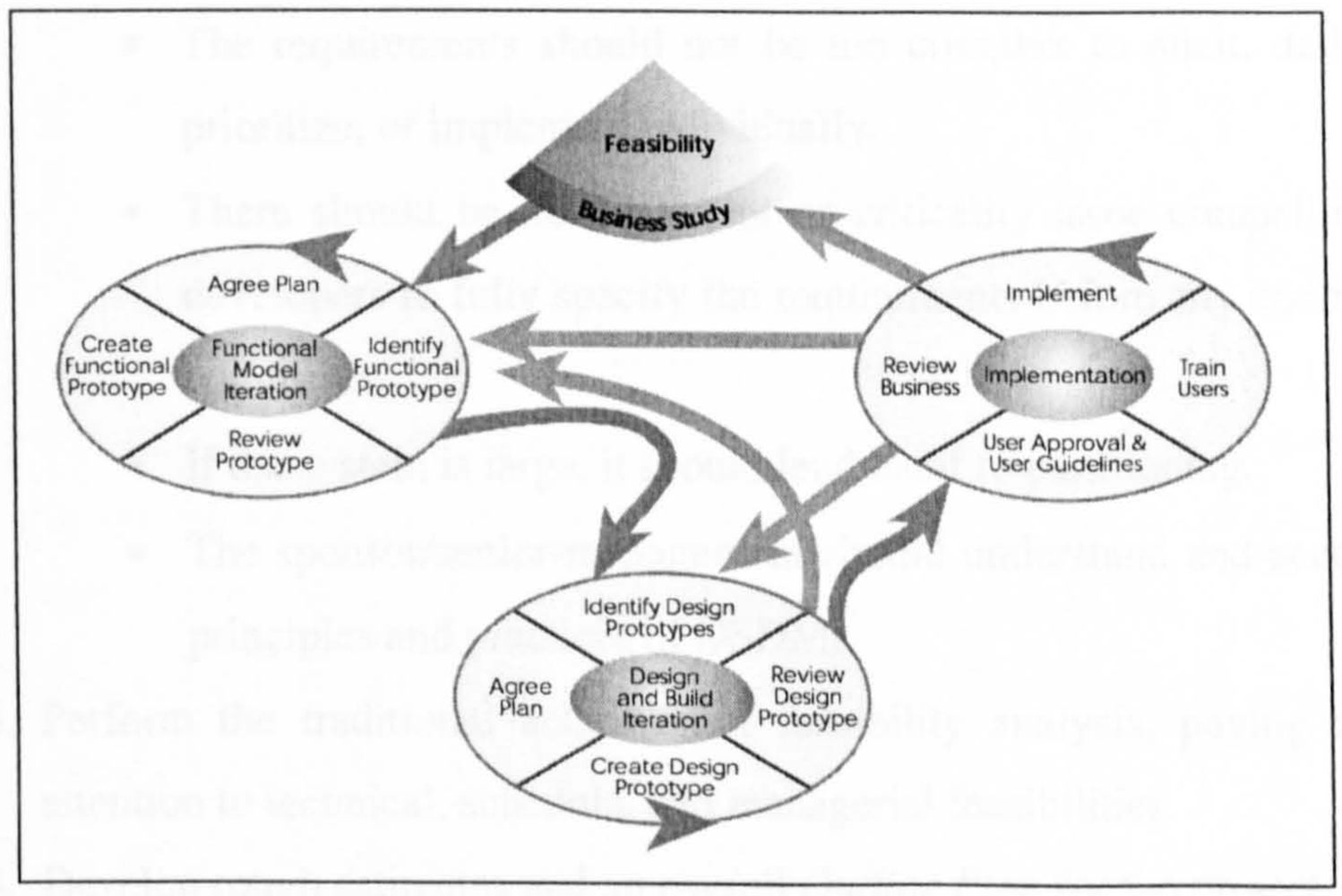

Figure 24. The DSDM process: the five main phases of the framework [DSDM Consortium 2003]

Feasibility Study (DSDM)

In this short phase, which typically takes no more than a few weeks, the following tasks are performed:

1. Acquire high-level knowledge as to the nature of the project, its scope, and the risks and constraints involved.
2. Check whether DSDM is the suitable approach for the project in hand. This is done by applying a list of project and organizational criteria (called the *Suitability Filter*) to the project. The suitability filter defines the characteristics that should be present in a project for DSDM to be properly applicable. The following non-exhaustive list includes a number of the more important characteristics, some of which are legacies from RAD:
    - The system to be developed should be interactive, with the functionality amply visible at the user interface level (screens, reports and controls), thus allowing prototyping to be effectively applied.
    - The system should have a clearly defined user group, so that well-informed representatives (called *Ambassador Users*) can be identified and involved as active participants in the project.
    - The system should not be computationally complex (more business-oriented rather than scientific).
    - The requirements should not be too complex to elicit, delineate, prioritize, or implement individually.
    - There should be no constraint or criticality issue compelling the developers to fully specify the requirements before any coding can commence.
    - If the system is large, it should lend itself to partitioning.
    - The sponsor/senior-management should understand and accept the principles and practices of DSDM.
3. Perform the traditional activities of feasibility analysis, paying special attention to technical, schedule, and managerial feasibilities.
4. Develop rough estimates and an overall *Outline Plan* for the project.

The results of the first three tasks are compiled in the *Feasibility Report*. The report may be complemented by a primitive prototype of the system (called the *Feasibility Prototype*), the main purpose of which is to demonstrate the scope and the technical feasibility of the project.

Business Study (DSDM)

The business study broadly encapsulates the following tasks, typically performed through a series of facilitated workshops involving the developers and the ambassador users:

1. Identify the processes and information entities in the business domain that are relevant to the system, as well as the types of users interacting with, or affected by, the system. The list of user-types will help identify ambassador users to participate in later tasks.
2. Define and prioritize the high-level functional and non-functional requirements of the system. The requirements are prioritized according to what DSDM calls the **MoSCoW** Rules, which is, in effect, categorizing each of the requirements as one of the following:
   - **M**ust-Haves: essential requirements on which the project's success relies.
   - **S**hould-Haves: important requirements, but not essential to the project's success.
   - **C**ould-Haves: requirements that can be excluded from the system functionality without having any serious effect on the project.
   - **W**on't-Haves: requirements that will not be part of the system functionality in the current project.

   The project must guarantee the implementation of the must-haves and should strive hard to deliver the should-haves. The could-haves will only be realized if time and resources allow their implementation.

   The results of the first two tasks are packaged as the *Business Area Definition* document.
3. Develop the *System Architecture Definition*, which highlights the architecture of the software solution, and specifies the development and operational platforms.

4. Produce the *Prototyping Plan*, outlining the order of activities during the iterative phases of the development.

Functional Model Iteration (DSDM)

In this iterative phase of the process, based on the high-level specifications outlined during the business study, detailed systems analysis is carried out through evolutionary prototyping. The following tasks are to be performed during the overall phase:

- A risk analysis is conducted in order to assess the risks involved in developing the requirements. The analysis will be refined during the iterations (based on the feedback and experience gained from the prototypes), ultimately resulting in the *Development Risk Analysis Report.*
- Requirements are selected according to their development risk (higher risk meaning higher priority), and functional prototypes are iteratively built in order to demonstrate the relevant functionality to the ambassador users, and refine the requirements based on the feedback. Testing is rigorously performed during the prototyping activities, and records are carefully logged. The prototypes produced in this phase not only constitute the embryo from which the final system will ultimately evolve, but as manifestations of the refined functional requirements, they also form the main part of the *Functional Model* of the system (thereby eradicating any need for the use of functional/behavioural modeling notations).
- Non-functional requirements are refined and listed. This list too is considered a constituent of the functional model.
- If necessary, static models (class diagrams) are used for modeling the structural aspects of the domain area being analyzed. These models are also appended to the functional model.

The above tasks are performed through iterations, with the following activities (similar to the activities in the traditional prototyping lifecycle) being carried out in each iteration:

1. Identify what is to be produced (the products).
2. Agree how and when to carry out the production (the plan).

3. Create the product(s).
4. Check that the products have been produced correctly (by reviewing documents, demonstrating a prototype or testing part of the system).

Design and Build Iteration (DSDM)

In this iterative phase of the process, the functional prototypes produced in the previous phase are completed and refined into a thoroughly tested and operational increment of the system. The prototypes from the functional model were merely meant for the purpose of requirements elicitation, refinement and modeling, and are therefore far from deployable: they are lacking in low-level functionality and structure, and do not adequately address non-functional requirements and implementation-specific issues. During the design-and-build phase, the prototypes are iteratively refined and gradually evolved into a working software subsystem, ready to be deployed as an increment into the operational environment, and integrated into the system built so far. The intermediate prototypes are called *Design Prototypes*, since they act as "live" executable blueprints for the final product.

As in the previous phase, the activities performed in each iteration are similar to those of the traditional prototyping lifecycle. Testing is performed on a continuous basis, with test cases and relevant results and decisions carefully logged. The intermediate prototypes are also kept on record as design documentation.

Implementation (DSDM)

During this phase of the project (which could have well been called Deployment, or Transition), the increment produced in the previous phase is deployed into the user environment, and integrated with the system built so far. Each iteration involves the following tasks:

1. Users and support personnel are trained, and manuals are prepared.
2. The increment is introduced into the operational environment. This naturally involves dealing with system integration and conversion issues, and the subsequent refactoring and testing activities.

3. A comprehensive validation review is performed on the system with feedback acquired from the users, results of which are compiled in the *Increment Review Document*. Based on the results of the review, alternative courses of action may be taken. There are four possible outcomes:
    - All requirements planned to be realized have been implemented to the users' satisfaction, in which case the project is declared as finished.
    - A major area of functionality was discovered during development that had to be abandoned because of time-box constraints, but should be developed; in this case a return to the business study phase is required.
    - An area of functionality had to be left out because of time-box constraints, but should be developed; in this case a return to the functional-model-iteration phase is required.
    - A non-functional requirement had to be ignored because of time-box constraints, yet should be realized; in this case a return to the design-and-build-iteration phase is required.

3.3.4.2 Scrum (1995, 2001)

The first mention of "Scrum" as a development method was made in 1986, when it was used to refer to a new fast and flexible product development process being practiced at that time in Japanese manufacturing companies. The name emphasizes the importance of teamwork in the methodology and is derived from the game of rugby. The variant of Scrum used for software development, jointly developed by Sutherland and Schwaber, was introduced in 1995 during a workshop at the annual ACM/OOPSLA conference [Schwaber 1995]. Originally intended as a general framework for systems development, Scrum is currently advertised as a comprehensive software development methodology [Schwaber and Beedle 2001, Schwaber 2004].

The Scrum process consists of three phases [Schwaber and Beedle 2001], as shown in Figure 25:

1. *Pre-game*: concerned with setting the stage for the iterative-incremental development effort; this phase consists of the following subphases:
    1.1. *Planning*: with the focus on producing an initial list of prioritized requirements for the system (called the *Product Backlog*), analyzing risks associated with the project, estimating the resources needed for implementing the requirements, obtaining the resources necessary for starting the development, and determining an overall schedule for the project.
    1.2. *Architecture/High-level Design*: with the focus on determining the overall architecture of the system in such a way as to accommodate the realization of the requirements identified so far.
2. *Development (Game)*: with the focus on iterative and incremental development of the system. Each iteration (called *Sprint*) is typically one month in duration and delivers an operational increment satisfying a predetermined subset of the product backlog.
3. *Post-game*: with the focus on integrating the increments produced and releasing the system into the user environment.

The following sections contain brief descriptions of the activities performed in each phase of the Scrum process.

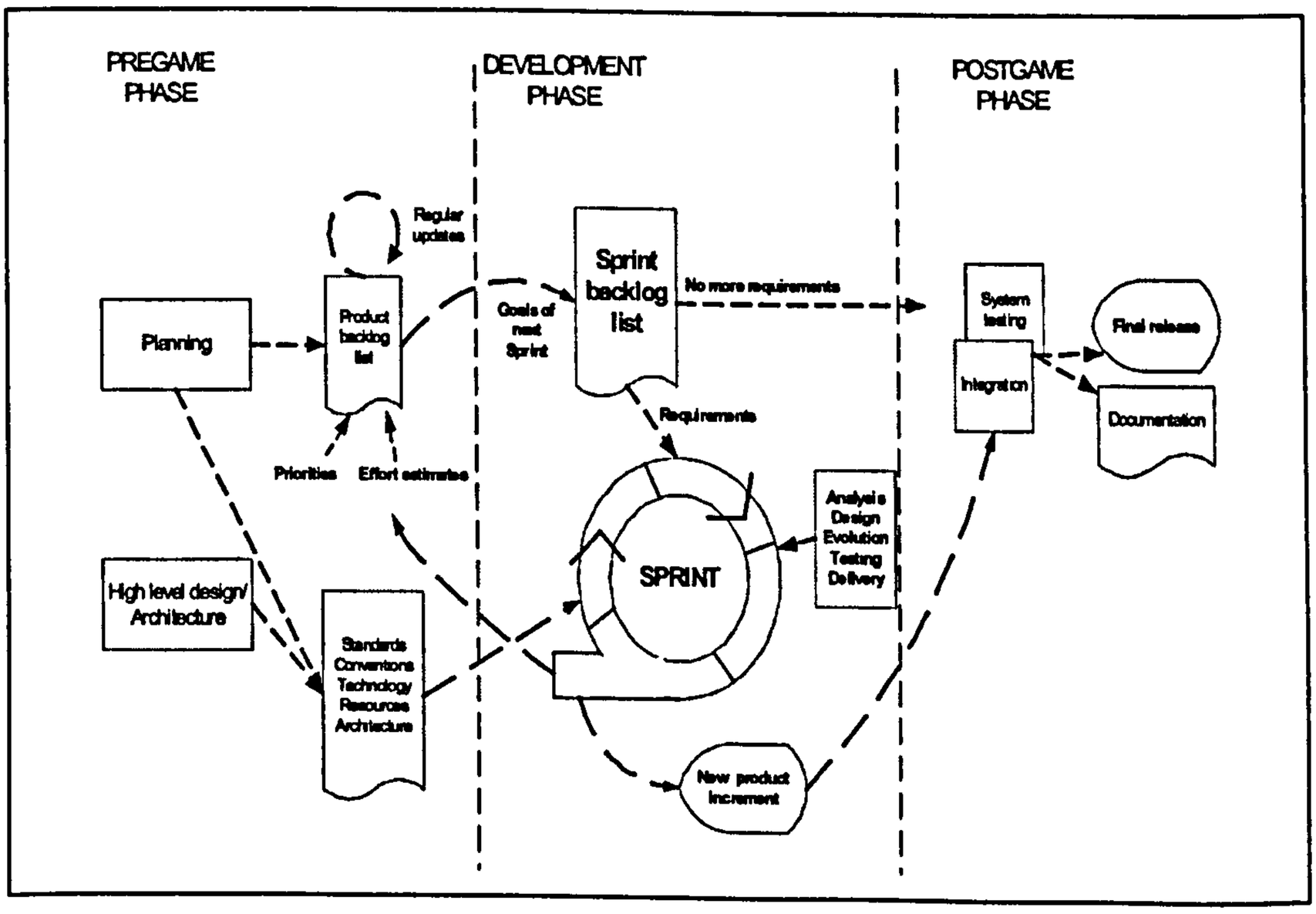

Figure 25. The Scrum process [Abrahamsson et al. 2002]

Pre-game (Scrum)

The two subphases comprising this phase usually overlap. The following activities are performed in the Planning subphase:

1. Development of an initial list of requirements (*Product Backlog*) for the system; the customer is fully involved in producing this initial version of the product backlog, but all other possible sources are also used for requirements elicitation. The product backlog will be completed and updated as the project moves on, always acting as the basis for the development effort. It will contain the functional and non-functional requirements of the system, as well as bug fixes and enhancements necessitated during the development process. Due to its utmost importance, a dedicated caretaker (typically a key user of the system), called the *Product Owner*, is put in charge of managing and controlling the product backlog.
2. Estimation of the effort and resources needed for developing the items on the product backlog and deploying the final system.
3. Assessment of the risk involved in developing the items on the product backlog.
4. Prioritization of the items on the product backlog.
5. Definition of a delivery date for the release of the system. If the system is too large, multiple releases might be deemed appropriate, and a delivery date specified for each.
6. Formation of development team(s); each team (called *Scrum Team*) typically has five to ten members with diverse specialties. The teams are supposed to be self-organizing, in that team-members collectively decide on issues of task assignment, team management and control. Nevertheless, a supervisor, or *Scrum Master*, is assigned to each team to act both as a facilitator in charge of removing the obstacles preventing the team's progress, and an enforcer of Scrum practices, making sure that the team does not digress from the course of action, values and guidelines laid out by Scrum.
7. Provision of tools and resources necessary for the actual development to commence.

The Architecture/High-level Design subphase consists of the following activities:

1. Problem domain analysis: based on the items in the product backlog, domain models reflecting the context and requirements of the system are built. Prototypes may also be built in order to gain better understanding of the problem domain.
2. Definition of the architecture of the system: this is done in such a way as to support the context and requirements of the system represented in the domain models.
3. Updating the product backlog: new backlog items are added and/or existing items are changed in order to accommodate the architecture designed.

Development (Scrum)

As the main development engine of the Scrum process, this phase is where the requirements listed in the product backlog are realized through iterative analysis, design, and implementation. This phase consists of a number of iterations, or *Sprints*, each of which produces an executable increment to the system. The following activities are performed in each sprint:

1. *Sprint Planning*: a *Sprint Planning Meeting* is held at the start of each sprint in which all parties concerned with the project – development team(s), users, customers, management, product owner and scrum master(s) – participate in order to define a goal for the sprint. The *Sprint Goal* defines the objective of the sprint in terms of the product backlog Items that it should implement. In defining the sprint goal, special attention is given to the priority of the items on the product backlog. The development team then sets out to determine a *Sprint Backlog*, which is a list of tasks to be performed during the sprint in order to meet the sprint goal. Thus the sprint backlog is a fine-grained, implementation-oriented, expanded subset of the product backlog. Items on the sprint backlog thus produced are assigned to the development team(s), and will be the basis for development activities performed during the rest of the sprint. If the sprint planning meeting concludes that no further sprints are necessary, the development phase is declared as finished, and the post-game phase is started.

2. *Sprint Development*: the development team analyzes, designs, and implements the requirements set in the sprint goal through performing the tasks detailed in the sprint backlog, all in the 30-day time frame set by the sprint. In order to effectively manage and control the activities of the sprint, 15-minute *Daily Scrum Meetings* are held during which the team-members discuss what they have achieved since the last meeting, their plans for the period leading to the next meeting, and the impediments they have encountered. The purpose of the meeting is to maintain and keep track of the progress of the team and resolve the problems that might adversely affect the team's pace. The management and the scrum master also attend the meetings and are to help overcome the problems faced by the team-members.
3. *Sprint Review*: a *Sprint Review Meeting* is held at the end of each sprint during which the increment produced is demonstrated to all the parties concerned. A comprehensive assessment is made of the achievements of the sprint in satisfying the sprint goal, and the product backlog is updated accordingly; i.e. fully realized requirements are marked as such, necessary bug fixes or enhancements are added, and appropriate changes are made to partially developed requirements. The sprint can also result in the identification of new requirements, or changes to already defined requirements, both of which are duly considered when updating the product backlog. As another objective of the sprint review meeting, issues impeding the progress of the development team are discussed and resolved. The meeting is also concerned with updating the system architecture according to the insight gained during the sprint.

Post-game (Scrum)

The following typical deployment activities are performed in this phase, introducing the release into the user environment:

1. Integration of the increments produced during the sprints.
2. System-wide testing.
3. Preparation of user documentation.
4. Preparation of training and marketing material.

5. Training the users and operators of the system.
6. System conversion/packaging.
7. Acceptance testing.

3.3.4.3 XP (1996, 2004)

XP (eXtreme Programming) was developed by Beck in 1996. Although the introductory material on the methodology was available on the Web almost from the start, it took three years for the first authentic XP book to appear [Beck 1999], with a revised and refined version appearing in 2004 [Beck and Andres 2004]. Although some of the methodologies that are nowadays dubbed as agile are older than XP, it was the advent of XP that sparked the agile movement.

XP considers itself a software engineering *discipline* rather than a *methodology*, yet it does incorporate a process. The XP lifecycle consists of six phases (Figure 26):

1. *Exploration*: with the focus on developing an initial list of high-level requirements, and determining the overall design of the system through prototyping.
2. *Planning*: also called *Release Planning*, this phase's focus is on estimating the time needed for the implementation of each requirement, prioritizing the requirements, and determining a schedule (as well as a minimal, select set of requirements to be implemented) for the first release of the system.
3. *Iterations to First Release*: with the focus on iterative development of the first release of the system, using the specific rules and practices prescribed by XP. Iterations are typically between 1 to 3 weeks in duration.
4. *Productionizing*: with the focus on system-wide verification and validation of the first release, and its deployment into the user production environment.
5. *Maintenance*: with the focus on implementing the remaining requirements (including any resulting from post-deployment maintenance needs) into the running system. Unlike many other methodologies, entering the maintenance phase of XP does not mean that the project is over; in fact, maintenance is the time for system evolution, and therefore is the time when the project is considered to be in its "normal" state.

6. *Death*: with the focus on closing the project and conducting post-mortem review and documentation.

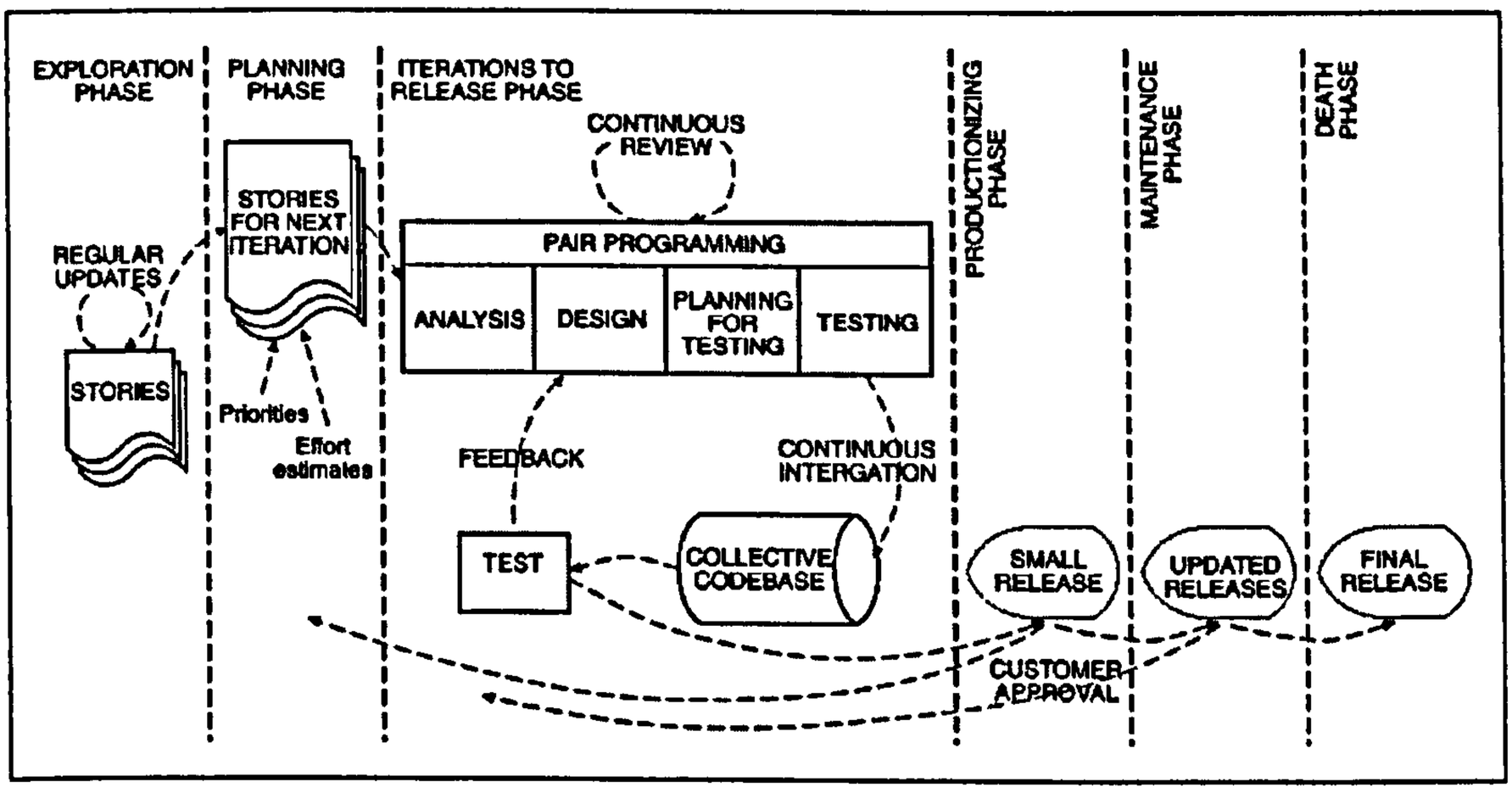

Figure 26. A general overview of the typical XP process [Abrahamsson et al. 2002]

The activities performed in the second, third and fourth phases of the process constitute the development "engine" of the XP methodology (Figure 27), in that each execution (run) of these phases produces a new release. According to the XP process, a first release of the system is initially produced and deployed, which is then incrementally improved and complemented during the maintenance phase through further iterations (runs) of the development engine.

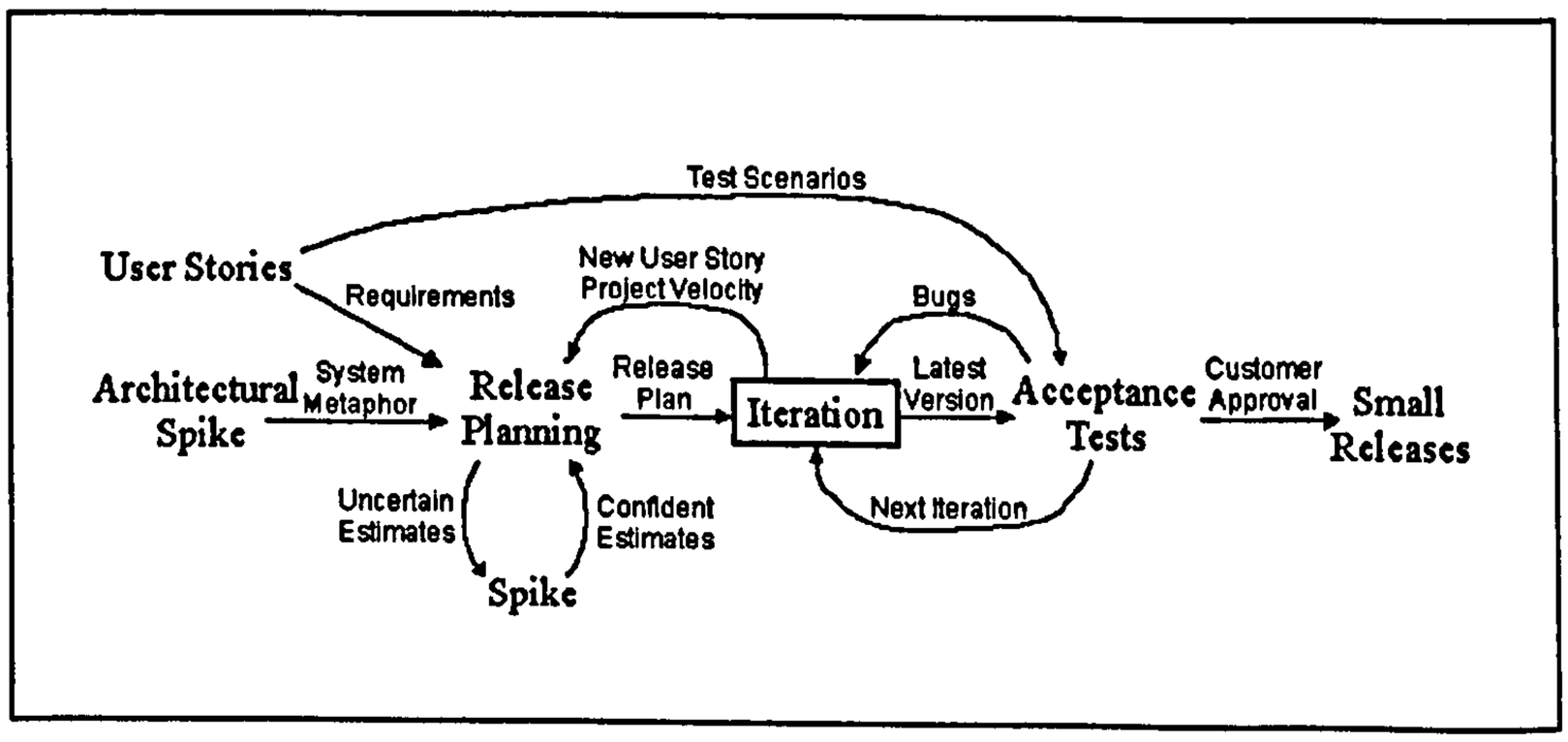

Figure 27. Top level activities in the XP development engine [Wells 2003]

The following sections contain brief descriptions of the activities performed in each phase of the XP process.

Exploration (XP)

The main activities performed in this phase of the XP process are as follows:

1. *Formation of the development team*: the team typically consists of a *coach* acting as monitor and facilitator, a number of *programmers*, and a business representative (*customer*) that should be always available to actively participate in project activities and supply the team with information and feedback. The team may also include a number of *analysts* to help elicit the requirements, a number of *testers* helping the customer define acceptance tests, and a *resource manager*.
2. *Development of the initial set of User Stories*: a *User Story* defines a feature of the system as seen from the customer's point of view. User stories are written by the customer in his own terminology on index cards, and are nothing but short descriptions (around three sentences) of a certain chunk of functionality needed to be delivered by the system. User stories are only detailed enough to allow relatively reliable estimation of the time needed for their implementation, and therefore only provide a high-level view of the requirements; yet they are the main drivers of the planning and development activities. The list of user stories is constantly updated during the process to reflect the changes and additions made.
3. *Creation of the system Metaphor*: a prototype (called *Spike* or *Spike Solution* in XP) is developed, exploring potential architectures for the system. The prototype helps the team define the system *Metaphor*, which is typically a very simple, high-level description of how the system works. It usually takes the form of a description-by-analogy in order to be easily understandable to all the team members. Though informal, the metaphor gives an extremely useful idea of the overall architecture of the system without setting too many constraints.

Planning (XP)

The main activities performed in this rather short phase of the XP process (typically taking no more than a couple of days), which is also called Release Planning, are as follows:

1. *Estimation of development time*: developers estimate the time needed to develop each of the user stories as conforming to the system metaphor, and write the estimates down on the user-story index cards. User stories that need more than 3 weeks to develop are broken down into smaller ones, and user stories taking less than 1 week are merged. In cases where estimates are not reliable enough, spike solutions (prototypes) are developed in order to help the developers mitigate schedule risks, and improve the estimates.
2. *Prioritization of user stories*: the customer prioritizes the user stories according to their business value.
3. *Planning the first release*: the team selects a minimal, most valuable set of user stories for implementation in the first release, and agrees on the release date. In doing so, the team also decides on the iteration duration (between 1 to 3 weeks), which once determined, will be the same for all iterations. The resultant release plan will be the framework according to which the iterative development effort in the next phase will proceed.

Iterations to First Release (XP)

This phase is the iterative development core of the XP process, with the ultimate objective of producing the first working release of the system according to the release plan. As a result of development activities, new user stories may be identified, and the existing ones may change. The following activities are performed in each of the iterations (Figure 28):

1. *Iteration planning*: at the start of each iteration, a planning meeting is held during which the development team performs the following activities:
    1.1. *Selection of user stories to implement, as well as failed acceptance tests of previous iterations that should be rectified*: based on the release plan, the customer selects user stories (according to their business value) for development in the coming

iteration. Failed acceptance tests encountered during previous iterations are also considered for inclusion in the list of jobs to be attended to. Special attention is given to the experience gained during previous iterations as to the development speed of the team (called *Project Velocity* in XP) in order to make sure that the selected jobs can indeed be completed by the end of the iteration.

1.2. *Identification of programming tasks*: the developers on the team break down the selected user stories and debugging jobs into programming tasks, which are then written down on the user-story index cards.

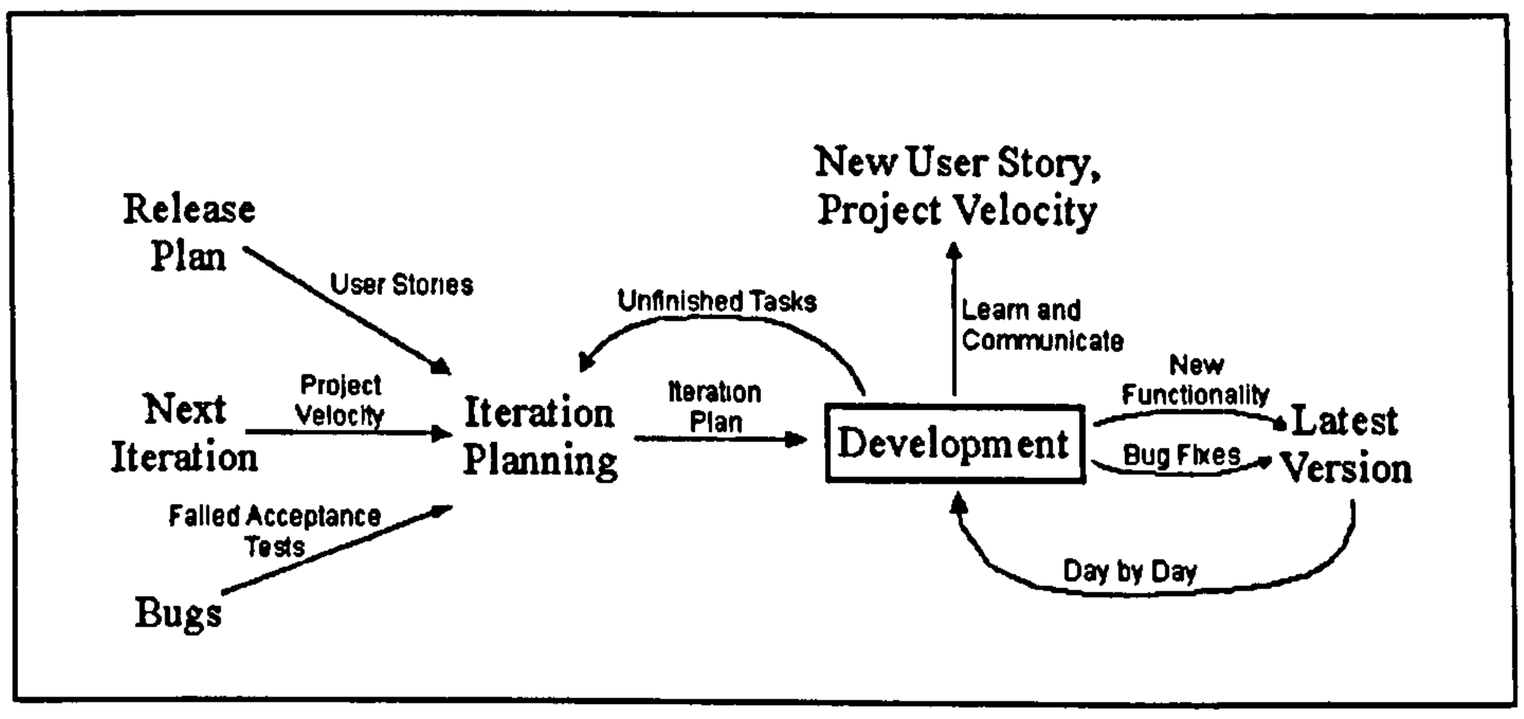

Figure 28. Activities in each iteration (XP) [Wells 2003]

1.3. *Task sign-up and estimation*: programmers sign-up to do the tasks. Each developer then estimates the time he needs for completion of each of the tasks he has undertaken, making sure that he can develop all of them in the time available. Each task should take between 1 to 3 days to complete.

2. *Development*: the development activity in each iteration is itself an iterative process with daily cycles. The main activities performed during development, as shown in Figure 29, are as follows:

    2.1. *Holding daily stand up meetings*: A short stand up meeting is held every morning in order to communicate problems and solutions, and help the team keep on track.

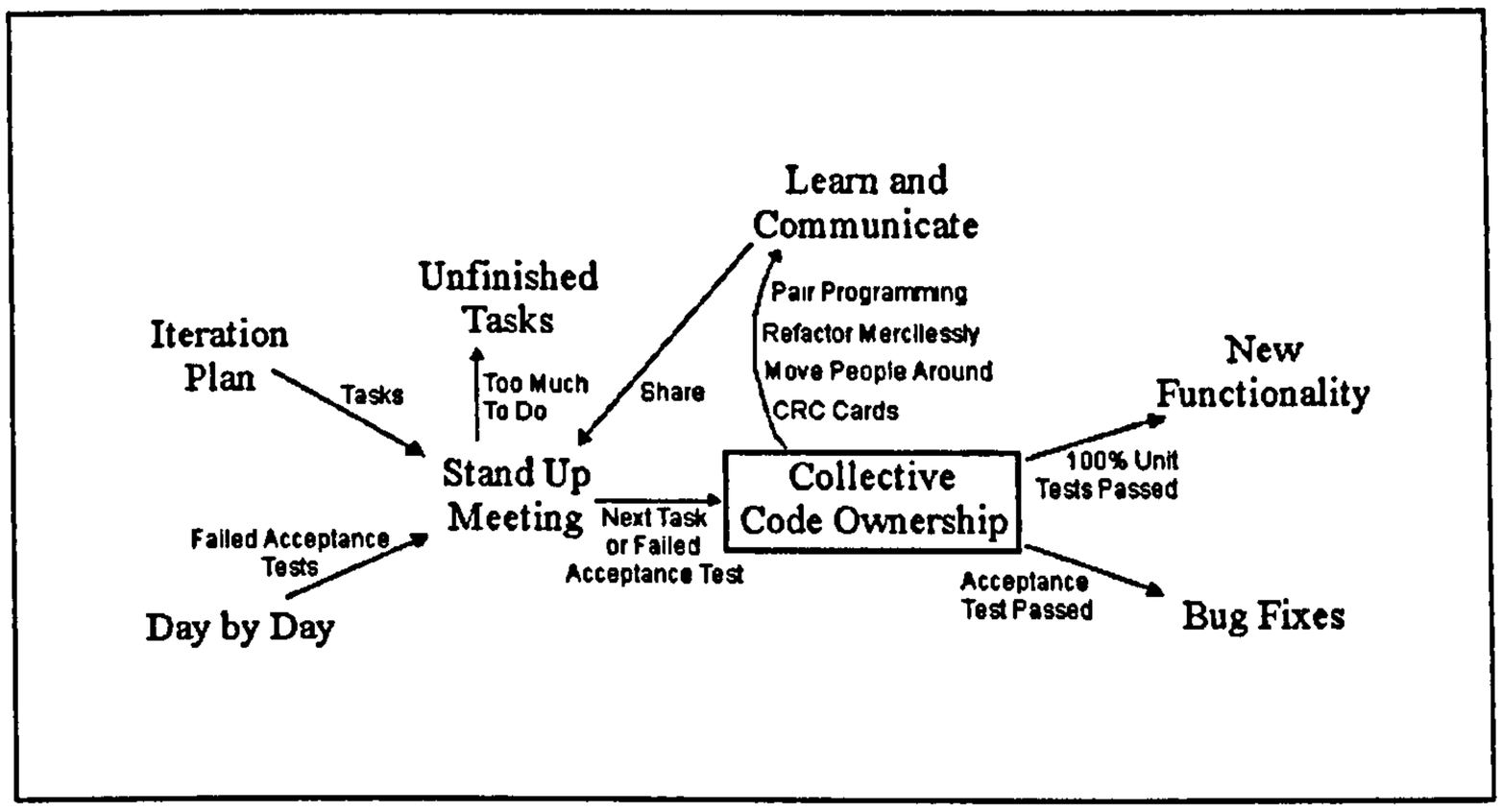

Figure 29. Development activities in each iteration (XP) [Wells 2003]

2.2. *Analysis, design, coding, testing and integration in a Collective-Code-Ownership environment* (Figure 30): *Collective Code Ownership* means that all the code developed is put in a shared code repository, and any developer can change his or others' code in order to add functionality, fix bugs, or refactor. In order to make collective code ownership possible, test-driven development is applied: the developers have to create unit tests for their code as they develop it. All the code in the code repository includes unit tests, forming a suite of tests that is automatically applied by test tools whenever code is added or changed. Builds are frequent in XP, and continuous integration is encouraged; yet, for code to be allowed integration into the repository, it must pass the entire test suite. The test suite thus safeguards the repository from malignant change.

In order to make sure that the user stories are indeed being implemented, black-box acceptance tests based on the user stories are defined by the customer and developed by the team during the iteration. Acceptance tests are frequently applied (by automated tools) to the code; the defects detected are relegated to the next iterations if time constraints do not allow their rectification in the present cycle.

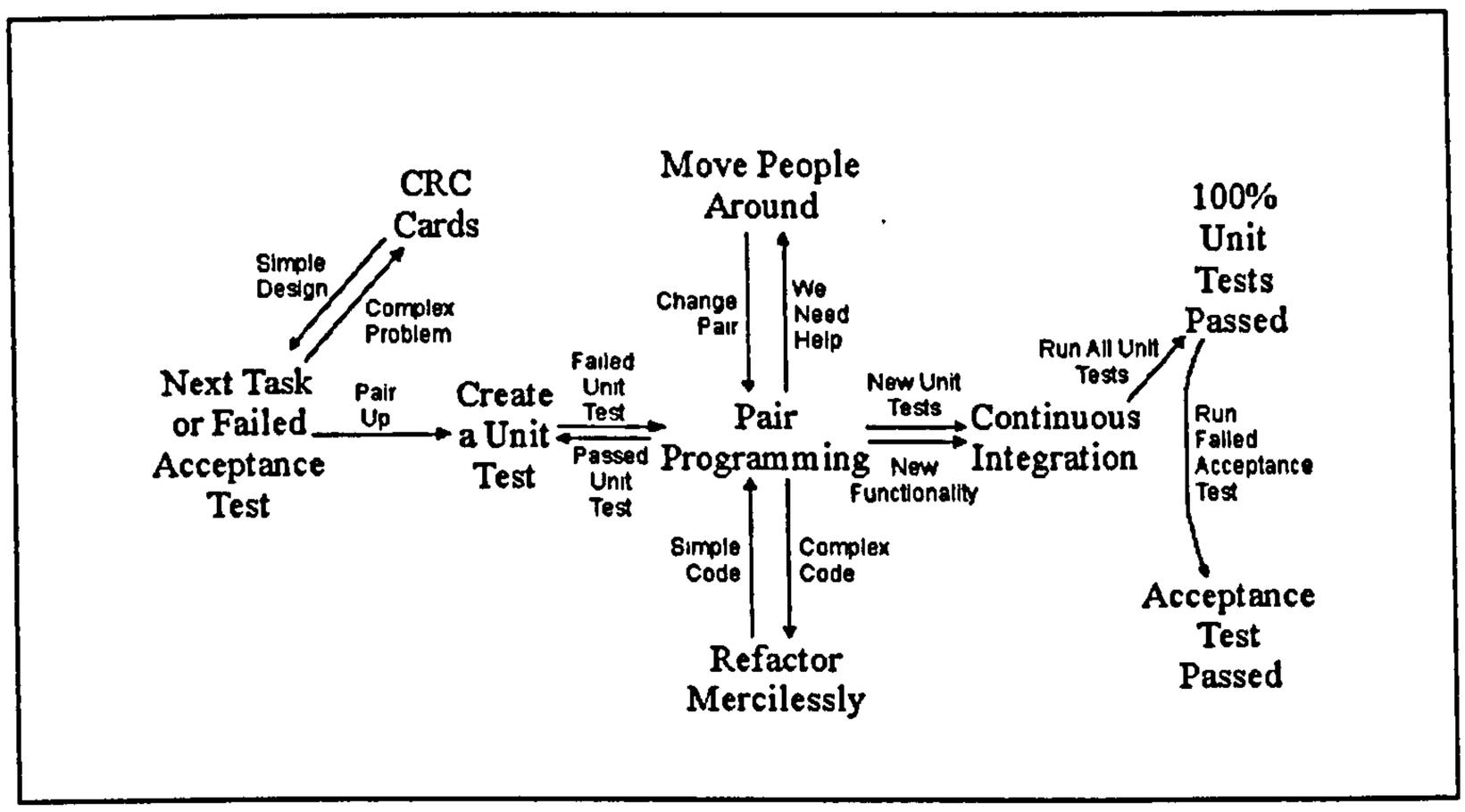

Figure 30. Activities in a Collective-Code-Ownership environment (XP) [Wells 2003]

Other rigorous development practices are also prescribed by XP, which will be briefly mentioned here. Although many of these development practices are much older than XP itself, XP was the first methodology to combine them into a synergistic development-practice core:

- Programmers work in pairs, each pair on one machine (a practice called *Pair Programming*).
- Programmers use CRC cards (explained in Section 3.3.2.3) in order to come up with the simplest design possible for the programming task in hand.
- Refactoring is constantly done in order to simplify the code and eliminate redundancy.
- A common coding standard is enforced in order to promote code legibility, which in turn enhances communication among developers.
- Developers are moved around so that they acquire knowledge about all parts of the system; this will reduce the cost of changes made to the team structure and will help relieve overloading and coding bottlenecks.

- Developers are to work at a sustainable pace, with forty hours a week as the norm; nobody is allowed to work overtime for two weeks in a row.

Productionizing (XP)

The main activities performed in this phase of the XP process are as follows:

1. *System-wide verification and validation*: the release is tested in order to make sure of the user's approval and the system's readiness for deployment. Acceptance tests, mostly developed during the iterations-to-first-release phase, are used here as regression tests. Defects found are resolved through iterations of the main development cycle.
2. *Deployment into the production environment*: the release is introduced into the user environment. This naturally involves the usual integration, conversion, tuning, training, and documentation activities typical of deployment efforts. Any tuning and stabilization action on the release itself is regarded as a development activity (analogous to user story development) and is conducted through short iterations (typically weekly) of the development cycle.

Maintenance (XP)

This post-deployment phase of the XP process encompasses the same activities as those in the previous three phases (the *development engine*); i.e. Planning, Iterations to First Release (the "First" will be dropped though), and Productionizing. It is still dependent on the evolving set of user stories and the system metaphor, and the activities in the constituent phases are performed in the same order as before. The important difference is that the small releases produced during maintenance are integrated into an already running and operational system. The maintenance phase is when the remaining user stories are implemented into the system (thereby evolving the operational first release into a complete system) and the system is maintained as such. As customary in iterative and incremental processes, requirements arising as a result of maintenance are treated as ordinary requirements (also expressed as user stories) and implemented through the same

iterative development process. Maintenance in this way employs a uniform process for both evolving and maintaining the system over its operational life.

The maintenance phase continues until either there are no more user-stories to develop and none are anticipated in the future (an improbable happy ending for the project effort), or the system in no way lends itself to necessary evolution any more.

Death (XP)

The project is declared dead when evolution is either unnecessary or impossible. The main activities performed in this final phase of the XP process are as follows:

1. *Declaring the project as closed*: this involves wrapping up the usual legal, financial and social loose ends.
2. *Post-mortem documentation and review*: this mainly involves preparing a short document (no longer than ten pages) providing a brief tour of the system, and writing a review report summarizing the lessons learned from the project.

3.3.4.4 ASD (1997, 2000)

ASD (Adaptive Software Development) was introduced by James Highsmith in 1997 [Highsmith 1997]. A refined and extended version was introduced in 2000 [Highsmith 2000a]. Evolved from a RAD process and based on the teachings of the complexity theory, ASD strives to present a change-tolerant, *adaptive* alternative to the *classical* Plan-Design-Build and the *iterative* Plan-Build-Revise lifecycles. The component-based development lifecycle prescribed by the ASD methodology assumes that all aspects and constituents of the development effort (business environment, people, requirements, resources, methods, etc.) are highly volatile, and that building complex systems is an evolutionary process extremely difficult to achieve unless special measures are taken to facilitate collaboration among the people who are somehow involved or affected by the development of the system.

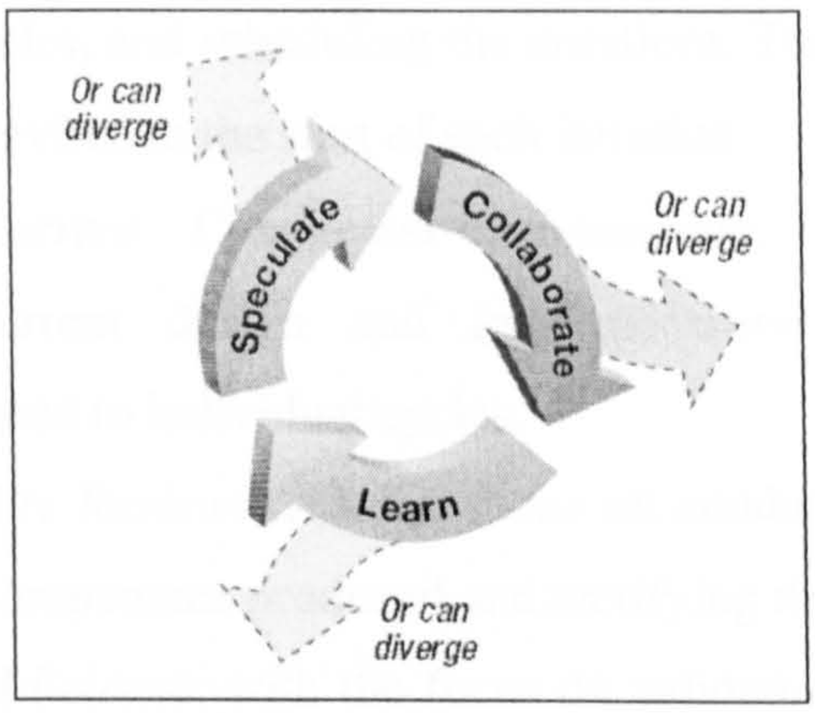

Figure 31. The basic Adaptive Software Development (ASD) lifecycle [Highsmith 2000b]

According to ASD, the uncertain and unpredictable nature of the development leaves developers no alternative but to use short iterations, or *cycles*. In order to bound the development effort and keep it focused, a specific mission, a set of components to develop, and a time box are defined for each cycle. Iterations should be planned, but plans are only risk-driven *speculations*, requiring revision after each iteration of the cycle; the actual design and implementation of the system components becomes a by-product of intense *collaboration*; and for the process to be adaptive, group reviews are performed at the end of each cycle, to enable the people involved to *learn* from the experience and implement the lessons learned in the process. The Speculate-Collaborate-Learn lifecycle thus formed (Figure 31) becomes the basic ASD framework for developing software systems.

ASD goes further than specifying just a framework: it also specifies the concrete phases comprising the lifecycle. The five phases constituting the ASD process, the three middle phases of which form the iterative development engine of the methodology, are as follows [Highsmith 2000a]:

1. *Project Initiation*: with the focus on understanding the project's objectives and estimating its size and scope, exploring the constraints and the risks involved, organizing the development teams, identifying high-level requirements, and specifying success criteria.
2. *Iterative Development Phases*:
    2.1. *Adaptive Cycle Planning*: with the focus on setting time frames for the project and the development cycles, defining the components that should be developed, assigning the components

to cycles, and scheduling the iterations. The plan will be revisited and revised at the start of each iteration.

2.2. *Concurrent Component Engineering*: with the focus on concurrent design and implementation of the components assigned to individual cycles.

2.3. *Quality Review*: with the focus on conducting group reviews of the components produced and rectifying the problems confronted.

3. *Final Q/A and Release*: with the focus on validating the produced system and deploying it into the working environment.

Figure 32 shows the order of the phases and their relative mapping to the basic Speculate-Collaborate-Learn lifecycle. The five phases of ASD and the activities performed in each are briefly described in the following sections.

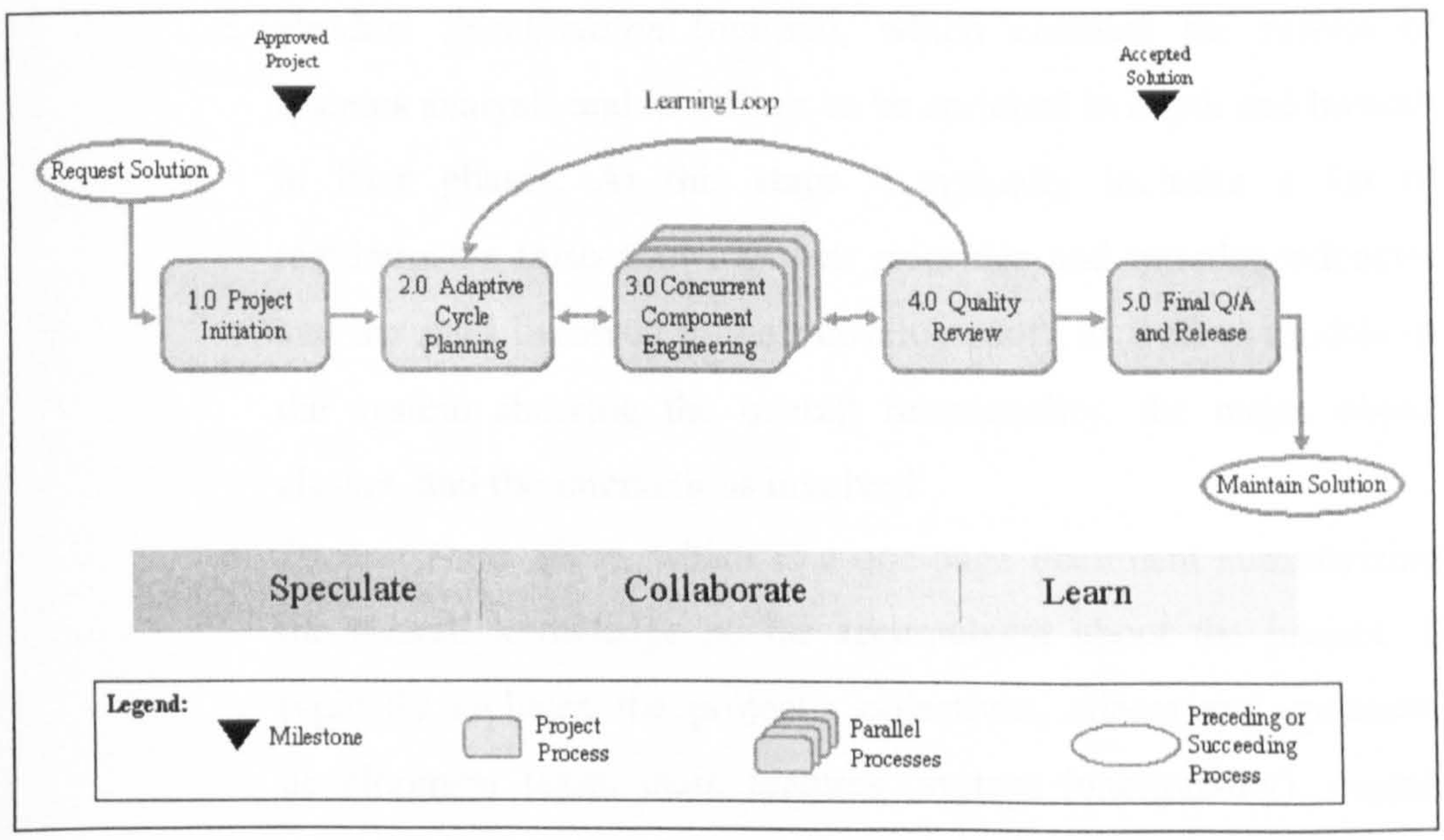

Figure 32. The ASD process [Crystal Methodologies Organization 2001]

Project Initiation (ASD)

The activities performed in this phase are as follows:

1. *Specify the Project Mission*, which defines the objectives to be achieved and broad requirements to be satisfied by the project.
2. *Identify the project team(s).*
3. *Create the Mission Artefacts*, consisting of the following:

a. *Project Vision* (*Charter*), which sets boundaries on the following:
    i. Scope, size, and context of the project.
    ii. Resources allocated to the project.
    iii. Project staff; defining the skills, knowledge, and authority required to successfully execute the project.
    iv. Communication among the people involved in or affected by the project, i.e. the *Project Community*.
b. *Product Mission Profile*, which identifies the primary factors governing the product's success. The main part of this profile is a matrix depicting the priority to be assigned to the four project variables of *scope*, *quality*, *schedule*, and *resources* in order to lead the project towards a successful product. The matrix also shows the target values to be achieved for each variable, and the degree of tradeoff allowed.
c. *Product Specification* (outline), which contains the results of systems analysis and modeling, to be enriched in depth and breadth in later phases. At this stage it typically includes a list of requirements (also showing their priorities and interdependencies and the risks involved in their development), as well as models of the system showing the overall functionality, the major object classes, and the interactions involved.
d. *Project Data Sheet*, which is a one-page document summarizing the overall knowledge so far accumulated about the project. It typically includes the project's objectives, clients and sponsors, development team, main features (system functionality), overall scope (in the shape of a *Context Diagram*), resources, benefits and implications, milestones, constraints, priorities, and the key risks involved.

The necessary information for producing the artefacts is usually obtained through JAD sessions.

4. *Obtain approval* of the clients/sponsors and the permission to go ahead with the project.
5. *Share mission values* among the project community, through discussing and agreeing on quality objectives and evaluation criteria.

Adaptive Cycle Planning (ASD)

The activities performed in this first phase of the iterative-development part of the ASD process are as follows:

1. *Determine time boxes* for the entire project and each of the development cycles. Before specifying time frames for the development cycles, the number of cycles necessary for developing the system should be estimated. Cycle time boxes in ASD are typically between two to eight weeks in duration.
2. *Write objective statements* for the development cycles. The objective statement will help the development team focus its efforts during the cycle.
3. *Define product components* through JAD sessions. The components form the ultimate system implementing the requirements, and are of three types: *feature components*, which are domain-specific, *analysis components* that enact the business logic of the system; and *technology components* and *support components*, which are domain-independent, *design* components that act as the technical infrastructure on which feature components rely for execution and perfect run-time operation.
4. *Assign components to cycles* according to the risks involved in their development and with careful consideration given to their interdependencies. The assignment should be such that each cycle delivers a tangible result.
5. *Plan the project*; an activity that typically involves developing buffered schedules for the development cycles (considering the risks involved in each and the resources they require), and setting up a suitable medium (methods, tools and procedures) for enabling and enhancing collaboration among members of the project community.
6. *Develop a Project Task List*, consisting of the tasks that should be performed during the remaining phases of the project. Naturally, most of the tasks are directly related to the development of components.

Due to the iterative nature of this phase, the speculative plans produced during the first iteration are revised and updated during later iterations to reflect the lessons learned.

Concurrent Component Engineering (ASD)

The activities performed in this phase, which is rightly considered the heart of the iterative-development part of the ASD process, are as follows:

1. *Develop the components* assigned to the cycle. Working components are typically developed concurrently by development teams working in parallel and are delivered as *builds* on a daily or weekly basis. The produced builds are immediately fed into an integration process. Testing and refactoring are ongoing processes during this activity.
2. *Manage the project* through continuous monitoring and control. Maintaining the inter- and intra-team collaboration and keeping the cycle on the right track are the main concerns.
3. *Prepare for final Q/A* by developing system-level test plans and test cases.
4. *Prepare for quality review* by planning the review meetings to take place in the Quality Review phase.

Quality Review (ASD)

The activities performed in this last of the iterative phases are as follows:

1. *Conduct cycle review* by holding facilitated customer focus group sessions. The result of the cycle is presented to the customers. The feedback and change requests are carefully documented in order to be considered in later iterations.
2. *Determine next step*; decision is made on whether another iteration cycle should be initiated, or the system should be prepared for release.
3. *Conduct cycle post-mortem*, which typically involves reviewing the performance of the teams and the effectiveness of the methods used. The problems are then rectified so as not to adversely affect the next iterations.

Final Q/A and Release (ASD)

The activities performed in this phase are as follows:

1. *Perform tests*, with the main purpose of system-level validation.
2. *Evaluate the test results.*

3. *Fix the problems.*
4. *Make a decision* based on the test results, whether to release the system or to start a new development cycle.
5. *Transition to production*; typically involving deployment activities including system conversion, training, and preparation of documents.
6. *Close the project*, which, in addition to the usual wrapping-up and termination procedures, also includes a project post-mortem summarizing the lessons learned from the execution of the project.

3.3.4.5 dX (1998)

The dX methodology was introduced by Martin in 1998 as an agile instance of RUP [Booch et al. 1998]. Although a RUP derivative, dX closely resembles XP and is based on the same principles; even the name is XP rotated (Martin has claimed, however, that the methodology is referred to as dX because it is "very small" [Booch et al. 1998]). The dX process consists of the same four phases as RUP, yet the tasks performed in each phase are much simpler, and there is no trace of the elaborate disciplines (workflows) prescribed by RUP. The dX versions of the four phases are:

1. *Inception*: with the focus on determining the major requirements (use cases), producing a preliminary version of the project schedule, and designing a basic architecture for the system.
2. *Elaboration*: with the focus on iterative and incremental design and coding of higher- priority (i.e. higher-risk) use cases until the architecture of the system and the project-schedule are stabilized to a point that a release schedule can be reliably worked out.
3. *Construction*: with the focus on designing and coding the remaining use cases. In dX, the construction phase is a seamless extension of the elaboration phase, with the release schedule being the only milestone signifying the transition between the two.
4. *Transition*: with the focus on gradual introduction of the implemented releases of the system into the user environment, and the subsequent maintenance activities.

The four phases of dX and the tasks preformed are briefly described in the following sections.

Inception (dX)

Tasks performed in the inception phase are similar to the generic, and already familiar, *analysis* tasks. A team, consisting of *developers* and a *customer representative* (analogous to XP), is formed and takes on the following tasks:

1. The customer representative, taking into account the developers' viewpoints, writes simple descriptions of the major use cases on index cards. These use case cards are the only intermediate artefacts the production of which is enforced by dX.
2. Simple throwaway prototypes of the major use cases are developed in order to measure the efficiency of the development team and to verify that the use cases are at the appropriate level of granularity and detail. If there are alternative architectures for the system, which is typically the case, alternative prototypes are developed in order to obtain better understanding of the implications of each alternative architecture.
3. Based on the results obtained from the prototypes, a preliminary project schedule is prepared, which will be revised and improved in the course of the project, especially during the elaboration phase.
4. The results obtained from the prototypes are also used as a basis for choosing one of the alternative architectures as the initial version of the system architecture, which will be revised and improved during the course of the project, especially during the elaboration phase.

Elaboration (dX)

The elaboration phase is where the team, having determined the use cases, designs and implements the higher-risk ones. Mitigating the major risks in this way allows the system architecture and the project schedule to be stabilized, which in turn makes it possible for a release schedule to be produced. The design and implementation is done in short iterations, and the implemented increments are constantly integrated. Other features prescribed in dX are even more suggestive of XP's influence; index-card based planning techniques, customer tests, small

releases, simple designs, pair programming, test-driven development, stringent coding standards, ongoing design improvement, and collective code ownership are XP principles explicitly adhered to in dX.

The main tasks performed during elaboration are as follows:

1. The customer representative continues writing new use case cards and completing the existing ones.
2. The amount of effort needed for developing each use case is estimated by the developers and is written on the corresponding index card.
3. The use cases are prioritized according to their risk by the customer representative.
4. The actual development is done in short iterations, each of which involves the following activities:
    4.1. *Iteration Planning*: bound by the iteration-duration selected (which is typically no longer than one week), the customer representative allocates the higher-priority use cases to the iteration.
    4.2. *Design*: the use cases selected for development are designed to fit the system architecture. This is done during design sessions, in which the team decides on how to implement the use cases. Modeling may be done by any means the team finds appropriate (e.g. UML diagrams), yet is usually limited to using CRC cards, or simply writing the design decisions on the use case cards.
    4.3. *Coding*: pair programming, test-driven development, and constant refactoring are meticulously exercised. Collective code ownership is the accepted rule, and integration is performed continuously.
    4.4. *Post-iteration Revision*: after each iteration, the project schedule and the system architecture are revised to reflect the lessons learned. As soon as the project schedule and the architecture are stable enough, a release schedule is produced, and the transition to the construction phase takes place.

Construction (dX)

The construction phase consists of the same activities as the elaboration phase, except that the development is now performed according to the release schedule. Construction goes on until all the use cases are implemented and released.

Transition (dX)

In dX, like XP, releases are frequent, with the first happening as early as possible in the project. Transition starts immediately after this first release, running in parallel with the construction phase. The purpose of the transition phase is to introduce the software produced so far into the user community. This involves beta testing the release, integrating the release with existing systems, converting legacy databases and systems to support the new release, training the users, and ultimately, deploying the new system. Since the early releases of the system are generally lacking in functionality, a parallel conversion from the existing system to the new one is preferable if these early releases are to be safely introduced into the user environment.

3.3.4.6 Crystal (1998, 2004)

Based on the belief that different projects call for different methodologies, Cockburn has proposed Crystal as a family of methodologies [Cockburn 2001]. In Crystal, projects are categorized according to their size and the criticality of the system being produced. Four levels of criticality have been defined, based on what might be lost because of a failure in the produced system: Comfort (C), Discretionary Money (D), Essential Money (E), or Life (L). The maximum number of people that might have to get involved in a project is regarded as a measure of the project's size; therefore, a category L40 project is a project involving up to 40 people developing a life-critical system.

Crystal methodologies put heavy emphasis on communication among people involved in the project. Therefore, projects with a larger size require heavier methodologies since they involve more people, and hence, need better coordination, whereas projects with higher criticality call for a more rigorous approach, which might be accommodated by tuning a methodology used for a less

critical project. Based on this philosophy, Crystal methodologies are categorized according to the project size that they address. Each member of the Crystal family has been assigned a colour showing its relative complexity: the heavier the methodology, the darker the colour assigned to it. Figure 33 shows a portion of the project-type grid as defined in Crystal. Moving upward in the grid corresponds to higher project criticality, while moving to the right means larger project size and therefore more complex methodologies. The figure also shows a number of Crystal methodologies assigned to different project sizes and the project categories that they cover; i.e. *Clear*, *Yellow*, *Orange*, and *Red*, in ascending order of complexity. Other more heavyweight members of the family - namely *Maroon*, *Blue*, and *Violet* - have also been mentioned in the literature (though not shown in this grid), and yet others can be added if a usage context arises.

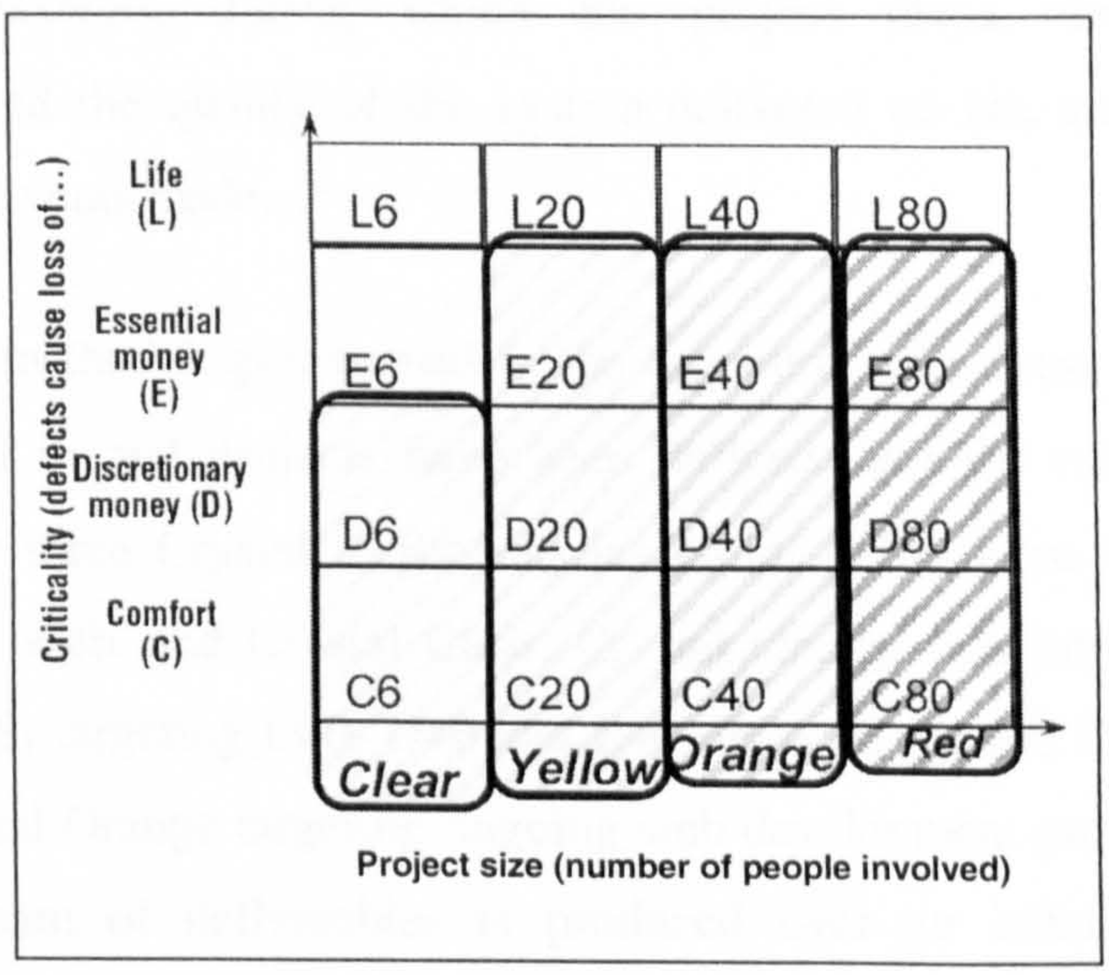

Figure 33. Project types in Crystal and the corresponding Crystal methodologies (partial grid) - adapted from [Cockburn 2001]

In addition to adhering to the principles of agile development [Beck et al. 2001], Crystal methodologies share several other common characteristics as well. Crystal methodologies do not support the development of life-critical systems, are iterative-incremental with each increment (delivery cycle) lasting no more than four months, do not support distributed teams and require the people involved to be collocated (e.g. in the same building), and depend on effective communication and information flow among team-members for successful enactment.

Every Crystal methodology enforces a development process framework and requires that a set of certain process elements (typically standard practices, strategies and techniques of a relatively general nature) be used, and certain work products be produced; yet, a large body of finer-grained detail is left to the development team to decide. In many cases, developers are even allowed to use techniques borrowed from other methodologies. Crystal methodologies thus provide means for tailoring the methodology to fit the project in hand: the development team(s) selects a base methodology at the start of the project (in the form of a minimal set of working conventions), and gradually refine and perfect it during development. This is Crystal's principal technique for making the development methodology adaptable to variable levels of project criticality and resilient to complications arising during development. In order to monitor and tune the development effort, Crystal methodologies make extensive and frequent use of *Reflection Workshops*, during which the project plans, the development methodology, and the quality of the system delivered so far, are reviewed and necessary adjustments made.

Of the Crystal methodologies named in the literature, only those that have been practically used in real projects have been defined, and the rest remain to be developed. The three Crystal methodologies so far defined are Crystal Orange, Crystal Orange Web, and Crystal Clear. Crystal Orange was introduced in 1998 [Cockburn 1998] targeting C40, D40 and E40 projects; Crystal Orange Web is a variant of Crystal Orange targeting ongoing web development projects in which a continuous stream of deliverables is produced over an indefinite time span [Cockburn 2001]. Crystal Clear, the lightest and most widely used member of the family, will be briefly described hereinafter.

Crystal Clear is primarily targeted at C6 and D6 projects [Cockburn 2004]. There is only one development team, with members working in close proximity to each other. Usable software is delivered at least once every three months, though delivery is typically expected to be much more frequent.

The project lifecycle in Crystal Clear consists of the following three sequential phases:

1. *Chartering*: taking a few days to a few weeks, this phase involves forming the development team, performing a preliminary feasibility analysis, shaping and fine-tuning the development methodology, and developing an initial plan for the project.
2. *Cyclic Delivery*: this is the main development engine of the process and consists of two or more *Delivery Cycles*. Each delivery cycle takes from one week to three months, during which the team updates and refines the release plan, implements a subset of the requirements through one or more program-test-integrate iterations, delivers the integrated product to real users, and reviews the development methodology adopted and the project plans. The iteration(s) in a delivery cycle are themselves composed of *daily* and *integration* cycles.
3. *Wrap-up*: during this last phase of the lifecycle, post-implementation activities are carried out, the software product is deployed into the user environment, and post-deployment reviews and reflections are performed.

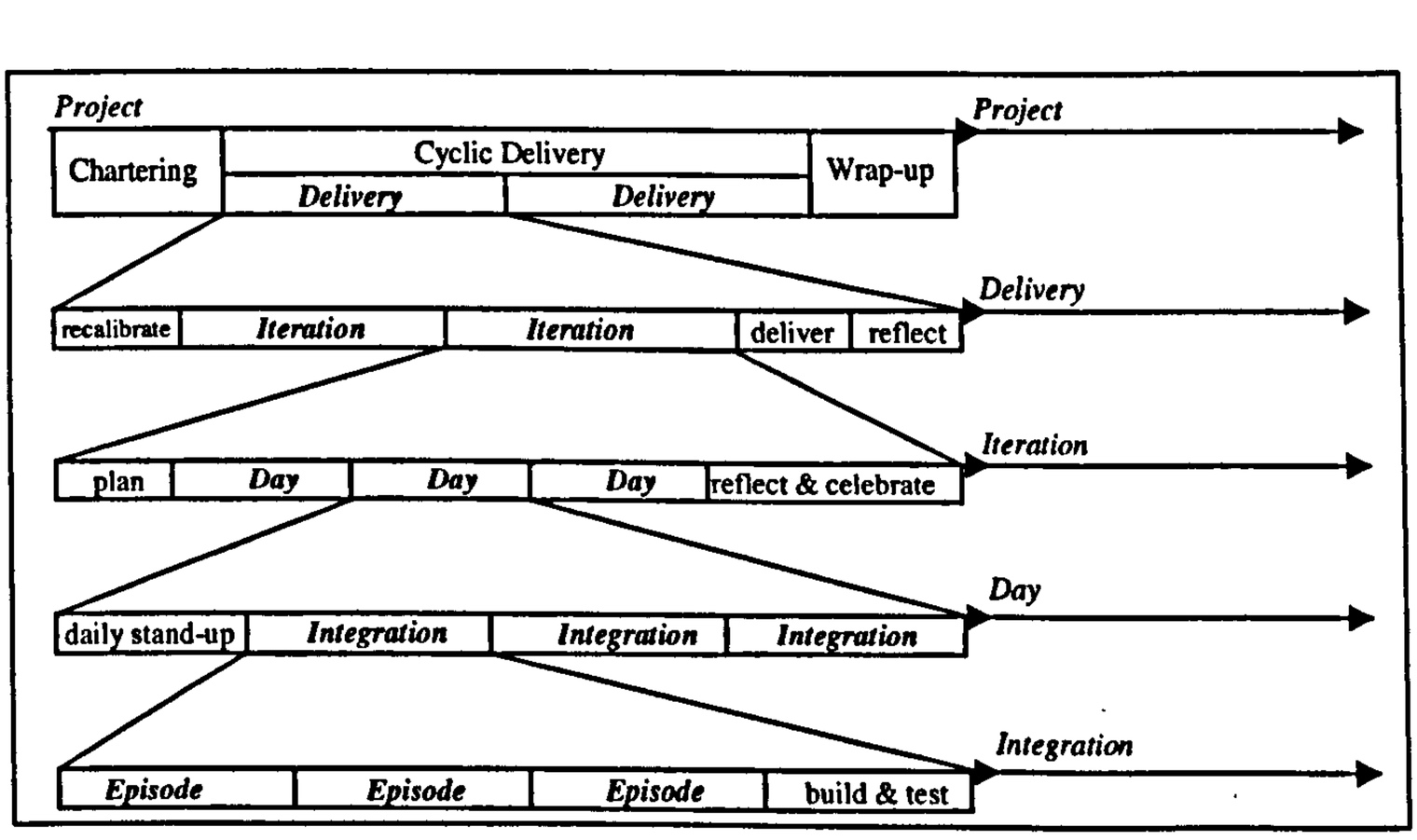

Figure 34. Example of phases, cycles and activities in Crystal Clear - adapted from [Cockburn 2004]

Figure 34 shows an example of the phases, cycles and activities in a typical project developed using Crystal Clear. The three phases of Crystal Clear and the cycles and activities performed in each are briefly described in the following sections.

Chartering (Crystal Clear)

This phase consists of the following four steps:

1. *Build the core of the team*, typically consisting of:
    a. An *Executive Sponsor*, who provides monetary and logistical support to the project and essential direction to the team, and may also act as the domain expert.
    b. A *Lead Designer*, who also acts as project manager, coordinator, and technical expert and trainer.
    c. An *Ambassador User*, who acts as the expert on system usage. Direct and active user involvement is essential to the methodology's success.
    d. A number of *Systems Analysts*, *Designer-Programmers*, *Business Experts*, *Testers*, *Text-Writers*, *Coordinators*, and others, as deemed necessary by the team (especially the above three main members).
2. *Perform the **Exploratory 360°***, which is a preliminary feasibility study providing a high-level project-wide review of the key issues governing the development of the project. These issues include: expected business value of the system and its high-level requirements, domain models, technology alternatives, overall project plans and constraints, necessary resources, and the development methodology. This step typically results in a decision to either go on with the project or terminate the effort due to infeasibility.
3. *Shape and fine-tune the methodology conventions*; a minimal set of rules is agreed upon by the team as the skeleton of the methodology to be used in developing the system. This initial set will be iteratively revised and perfected during cyclic delivery, gradually evolving into a methodology tailored to fit the project in hand.
4. *Build the initial project plan*, which typically includes a *Project Map* showing the development tasks and their dependencies, and a *Release Plan* showing the projected completion dates for delivery cycles and iterations. Tasks are identified, prioritized and estimated using a technique called *Blitz Planning*, which is a close variant of XP's card-based planning technique. The plans will be revisited and updated during cyclic delivery.

Cyclic Delivery (Crystal Clear)

This phase consists of two or more *Delivery Cycles*. Each delivery cycle involves the following four activities, collectively aimed at implementing, testing and delivering working software to the user:

1. *Recalibrate the release plan*: the requirements and the project plans are reviewed and updated according to the experience gained in the delivery cycles performed so far. Refinements are also made to the plans and fine-grained detail is added in order to accommodate the iterations to be performed in the current cycle.
2. *Develop in iterations*: one or more iterations are performed in every delivery cycle. Each iteration lasts from one week to three months, and consists of the following three activities
    2.1. *Iteration planning*: a fine-grained plan is produced involving the tasks that should be performed in the iteration.
    2.2. *Cyclic program-test-integrate*: an iteration consists of cyclic daily activities (Figure 34). The team's *Daily Cycle* includes a stand-up meeting (similar to that in Scrum), during which the team-members exchange information and ideas about their achievements, plans and problems. The rest of the day typically consists of several *Integration Cycles*. During each integration cycle, designer-programmers perform design-implementation *Episodes*; that is, they start development tasks, and carry out designing-programming (considered as one activity in Crystal) and unit testing. At the end of an integration cycle, the code produced by designer-programmers during the episodes of the integration cycle is integrated into the system built so far, and appropriate integration tests are performed. Developed code is thus continually integrated into the system, typically several times a day.
    2.3. *Iteration completion ritual*: a Reflection Workshop is held for reflecting on the quality of the code produced, the effectiveness of the development methodology and the reliability of the plans.

Necessary changes are made to the working conventions and the plans in order to resolve the problem issues.

3. *Deliver to real users*: the integrated system produced during the previous activity is delivered to a small number of users (preferably only one), and feedback is used for improving the system built so far and revising the plans and/or the requirements. As in most agile processes, delivery in Crystal Clear is frequent, necessitating frequent acceptance testing. Therefore, the number of users to which the system is delivered should be kept small in order to avoid excessive training and deployment costs.
4. *Reflect on the delivery*: through a workshop, the team reflect on the quality of the delivered product, the development methodology and the plans. The goal is to identify strengths and weaknesses and decide on ways for resolving the shortcomings.

Wrap-up (Crystal Clear)

The main purpose of this phase is to perform acceptance testing, prepare the final product and the user environment for final deployment, and ultimately carry out the system conversion. As expected, this phase also includes a final reflection activity aimed at compiling and recording the lessons learned from the project, in order to use them in future projects.

3.3.4.7 FDD (1999, 2002)

De Luca and Coad introduced FDD (Feature-Driven Development) in 1999, originally as a tailored complement to the "Object Modeling in Color" technique [Coad et al. 1999]. A revised version of the methodology was published in 2002 [Palmer and Felsing 2002]. This latter version had been completely decoupled from "Modeling in Color", and was general enough to be considered an independent methodology.

As the name implies, FDD is based on expressing and realizing the requirements in terms of small user-valued pieces of functionality called *Features*. Each feature is a relatively fine-grained function of the system expressed in client-valued terms, conforming to the general template: <*action*> <*result*> <*object*>; for example, "*calculate* the *total value* of a *shipment*" or "*check* the *availability of seats* on a

flight". The granularity of each feature should be such that it would take no more than two weeks to develop; otherwise it will be broken down into smaller features. Each feature is identified as a *Step* in one or more *Activities* (also called *Feature Sets*), and *activities* in turn belong to *Areas* (or *Major Feature Sets*). This three-layered structure allows the developers to adequately manage the complexity of the requirements. Furthermore, features can also be partitioned according to the architectural layer to which they belong: FDD prescribes a layered architecture for software systems (as explained later in this section), providing a further means for managing the complexity of requirements through architectural partitioning of features.

The FDD process consists of five subprocesses, during the course of which several deliverables are produced (Figure 35). The first three subprocesses are concerned with requirements analysis and development planning and are performed sequentially at the start of the process, whereas the remaining two are design and implementation activities, done in iterations of no longer than two weeks.

The subprocesses of the FDD process, as shown in Figure 35, are:

1. *Sequential Subprocesses*: during this primary sequential phase, the problem domain is modeled, requirements are identified as hierarchical lists of features, and development planning is performed. Although not explicitly included in any of the subprocesses, the sequential phase may also include the production of an architecture for the system, typically conforming to the general layered architecture proposed by FDD (Figure 36). The subprocesses, in the order they are performed, are as follows:

    1.1. *Develop an Overall Model*: with the focus on building a mainly structural model of the problem domain called the *Object Model*. This model mainly consists of full-featured class diagrams, yet it may also include sequence diagrams (if deemed necessary) for capturing important behavioural patterns of interaction in the problem domain. The object model will be extensively used, and refined, during the design-by-feature subprocess.

    1.2. *Build a Features List*: with the focus on identifying the required functionality of the system. This is done by first identifying the

areas of functionality in the system, and the *activities* performed in each area. *Features* are then identified as *steps* in the activities, and a three-layered pyramid of functionality, taking the form of a hierarchy of lists, is thus produced.

1.3. *Plan by Feature*: with the focus on scheduling the features for development, and then assigning the feature sets (*activities*), and the classes in the object model, to developers. During the iterative subprocesses, feature-set-developers (called *Chief Programmers*) will develop the feature sets assigned to them by commissioning class-developers (called *Class Owners*) to cooperate in order to design and implement the features.

2. *Iterative Subprocesses*: during this iterative development phase, strands of design-and-build iterations start off as each chief programmer selects the set of features (called the *Work Package*) that should be developed in each of the iterations performed under his supervision, and forms a team of class owners to do the job. A chief programmer selects features and schedules his iterations according to the overall development plan, taking care that each iteration takes no longer than two weeks to complete. Typically, at any point during this development period, several iterations are being performed concurrently, some of them supervised by the same chief programmer, with each of the class owners taking part in several iteration-teams simultaneously. The subprocesses, in the order they are performed in each iteration, are as follows:

2.1. *Design by Feature*: with the focus on determining how the features in the work package should be realized at run-time by interactions among objects. Sequence diagrams are drawn for each of the features, resulting in additions and modifications being made to the object model, and refined class and method descriptions being produced.

2.2. *Build by Feature*: with the focus on coding and unit-testing the necessary items for realization of the features in the work package. The implemented items that pass the tests are then promoted to the main build.

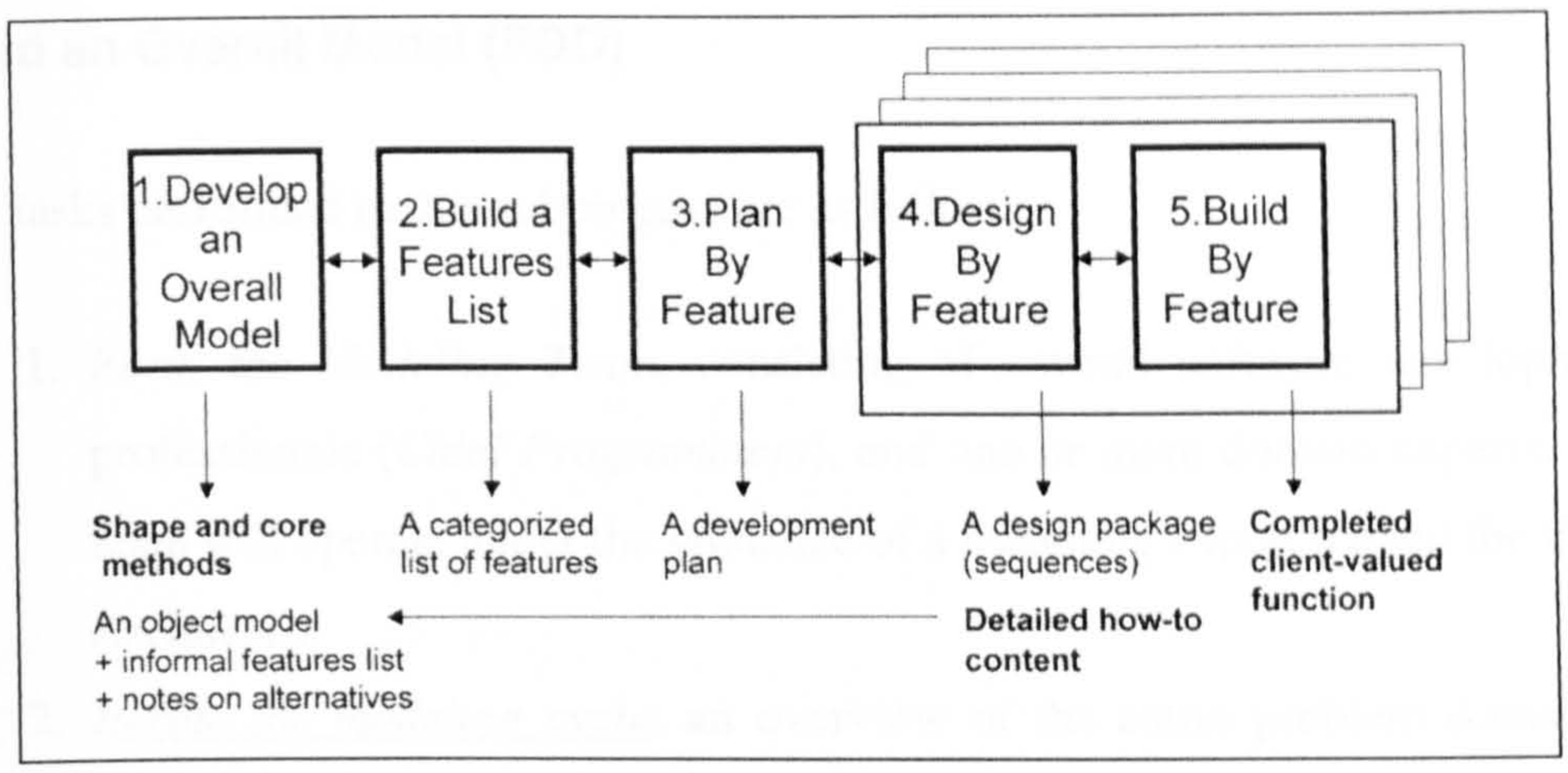

Figure 35. The FDD process and its deliverables [Palmer and Felsing 2002]

The FDD methodology cannot be considered an all-inclusive software development methodology, in that it starts when the feasibility study and overall project planning have already been done, a business case has been established, and permission has been granted by the sponsors to go on with the development. It also excludes post-implementation activities such as system-wide verification and validation, and the ultimate system deployment and maintenance. Before a project is started, a *Project Manager* is assigned who coordinates all development activities, making sure that project activities, with the FDD process embedded as the core, are performed coherently. The project manager's responsibilities include, among other usual project management duties, the forming of the various teams that should perform the FDD tasks.

The five subprocesses of FDD and the tasks performed in each are briefly described in the following sections.

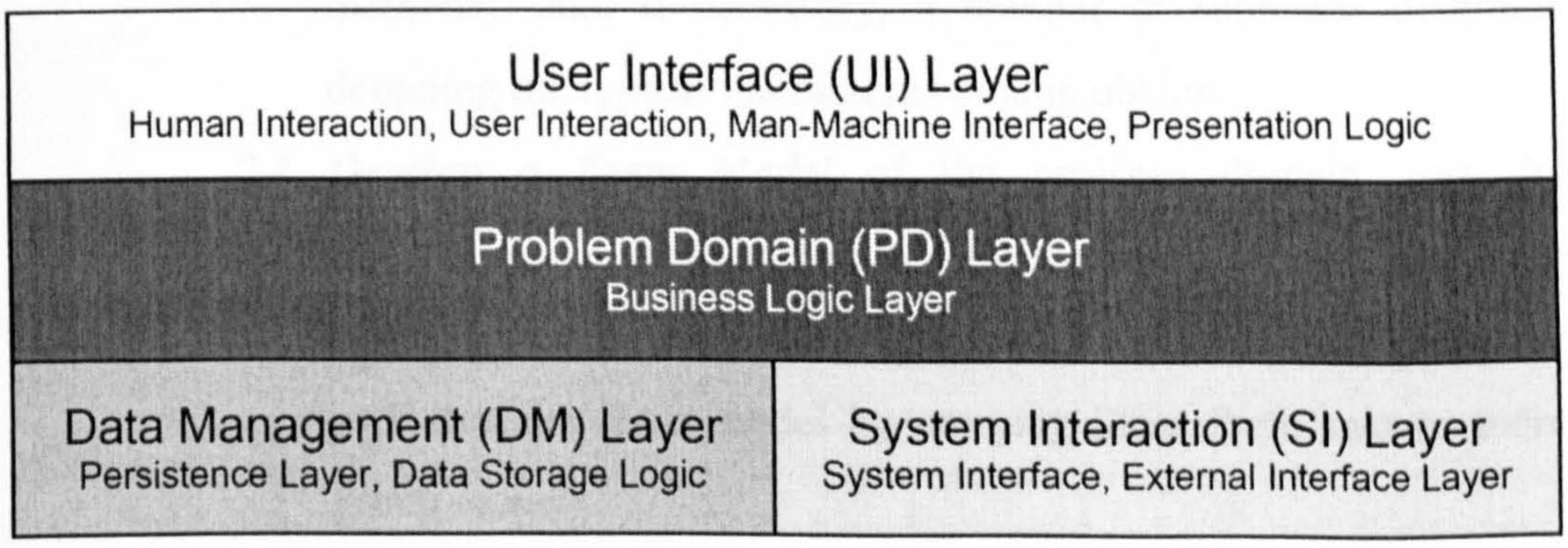

Figure 36. The general layered architecture of software systems as proposed by FDD [Palmer and Felsing 2002]

Build an Overall Model (FDD)

The tasks performed in this subprocess are as follows:

1. *Form the Modeling Team*, consisting of several software development professionals (*Chief Programmers*), and one or more domain experts. The team will operate under the guidance of a modeling expert (called the *Chief Architect*).
2. *Iterate the modeling cycle*: an overview of the entire problem domain is first presented by the domain experts. The problem domain is then partitioned into *areas*, and the modeling is performed iteratively: each problem-domain area is separately analysed and modeled through tasks 2.1 to 2.4 (below); the resulting sub-model is then integrated into the overall model through task 2.5, and model notes are added in task 2.6. This cycle is repeated until all problem domain areas are adequately covered and modeled to the satisfaction of the chief architect. The tasks performed in each iteration of the cycle are:
   2.1. *Conduct a domain-area walkthrough*, which is also presented by the domain experts.
   2.2. *Study documents* of the problem domain area (if available).
   2.3. *Develop small Group Models* of the problem domain area by breaking the modeling team into small groups (of no more than three members), and commissioning each group to develop its own version of the object model for the problem domain area. Each model will consist of full-featured class diagrams (showing classes, their inter-relationships, and their attributes and methods), and, if necessary, a number of sequence diagrams depicting the typical interactions among objects.
   2.4. *Develop a Team Model* of the problem domain area, by examining the models produced by the small groups. The team either approves one of the proposed models as the team model, or produces the team model by merging ideas from two or more group models.
   2.5. *Refine the overall Object Model* by integrating the model of the problem domain area into the overall problem-domain object

model produced so far. This naturally requires a certain degree of refactoring to be done.

2.6. *Write model notes*, which describe specific aspects of the model that are not explicitly addressed by the model itself, especially including accounts of the alternatives explored by the modeling team during the modeling process.

Build a Features List (FDD)

The tasks performed in this subprocess are as follows:

1. *Form the Features-List Team*, which consists of the chief programmers participating in the modeling team from the previous subprocess.
2. *Build the features list*, which is a three-layered hierarchical list with the following structure:
    - A list of *areas* (major feature sets).
    - For each area, a list of *activities* (feature sets) within that area.
    - For each activity, a list of *features* representing the steps in the activity.

The features-list is built in a top-down fashion: the features-list team first identifies the *areas* (high-level feature sets) by carefully investigating the knowledge acquired about the problem domain, particularly the problem-domain areas (partitions) identified while building the overall object model in the previous subprocess; the *activities* (low-level feature-sets) in each area, and the *features* (steps) in each activity are then identified by applying functional decomposition.

Plan By Feature (FDD)

The tasks performed in this subprocess are as follows:

1. *Form the Planning Team*, consisting of the project manager, the chief programmers, and a *Development Manager* (which is put in charge of the development effort, and as such, supervises the chief programmers).
2. *Determine the development sequence* by scheduling the development of the feature sets (*activities*), specifying a date (month and year) for the

completion of each. This requires taking into account the inter-dependencies among the feature sets, the workload distribution across the development team, and the risks associated with the feature-sets. A completion date is then determined for each *area* (major feature set) as the last completion date assigned to its constituent feature sets.

3. *Assign feature sets to Chief Programmers*, thereby declaring them as the *owners* of the feature-sets assigned to them.
4. *Assign classes to developers*, thereby declaring the developers as class owners.

Design By Feature (FDD)

The tasks performed in this subprocess are as follows:

1. *Form a Features Team*, which will design and build the feature(s) selected for development in the current iteration under the supervision of the chief programmer who owns the features. After identifying the set of classes that might be involved in the realization of the features, the chief programmer brings together the owners of these classes and thereby forms the features team.
2. *Conduct a domain walkthrough* (if at all necessary), by inviting domain expert(s) to help the features team grasp all the relevant particulars of the features. This task is usually undertaken for high-risk features, the development of which usually requires a deeper understanding of the data, algorithms, and constraints involved.
3. *Study the referenced documents* (if at all existent), in order to obtain a better understanding of the features. As with the previous task, this task is usually performed for high-risk features for which descriptive documentation already exists.
4. *Develop the sequence diagram(s)*, which as the pivotal part of the design models, are required to show how objects should interact at run-time in order to implement each of the features. The features team also meticulously logs the alternative design models it has explored, as well as the constraints and assumptions that apply.

5. *Refine the Object Model* (class diagrams) so that it supports the sequence diagrams produced in the previous task. This usually means that new elements are added to the model, some of the existing elements are changed, and refactoring is necessitated as a consequence.
6. *Write Class- and method-Prologues* for the elements of the object model. These relatively low-level design details are produced by the class owners as the last design artefacts needed before the coding can commence.
7. *Design inspection* is performed by the features team (possibly in consultation with other people involved in the project) in order to verify the integrity of the design artefacts produced.

The products of this subprocess are transferred to the next subprocess as a package. This *Design Package* consists of the sequence diagrams produced, the refinements made to the object model, the prologues, and the notes on the design alternatives explored, constraints, and assumptions.

Build By Feature (FDD)

The tasks performed in this subprocess are as follows:

1. *Implement classes and methods* according to the specifications given in the design package. Each of the class owners implements the necessary items (including the unit-testing code) in the classes he or she owns.
2. *Conduct a code inspection*, either before or after the unit-test, during which the features team examines the code to make sure of its integrity and conformance to coding standards.
3. *Unit-test* the code to ensure that all classes satisfy the functionality required. Class owners perform class-level unit-tests, as well as feature-level unit-tests prescribed by the chief programmer.
4. *Promote to the build*, if the implemented classes are successfully inspected and unit-tested. As the leader of the features team, it is the chief programmer who makes sure that all the classes necessary to realize the features are ultimately integrated into the main build.

3.3.5 Process Patterns

Process patterns are the results of applying abstraction to recurring processes and process components, thereby creating means for developing methodologies through composition of appropriate pattern instances. They are an invaluable source of insight for researchers, since they typically reflect the state of the practice and are based on well-established, refined concepts.

3.3.5.1 Introduction

The first recorded reference to the term "Process Pattern" was made by Coplien in his landmark paper in 1994 [Coplien 1994]. Coplien defined process patterns as "the patterns of activity within an organization (and hence within its project)", and almost all his patterns are relatively fine-grained techniques for exercising better organizational and management practices, which although quite useful, do not constitute a comprehensive, coherent whole for defining a software development process. A number of them, however, such as "Prototype" and "Decouple Stages", are indispensable in any process.

Ambler, who is the author of the only books so far written on object-oriented process patterns, defines a process pattern as "a pattern which describes a proven, successful approach and/or series of actions for developing software" [Ambler 1998a], and an object-oriented process pattern as "a collection of general techniques, actions, and/or tasks (activities) for developing object-oriented software" [Ambler 1998b].

A brief overview of Ambler's process patterns is presented in the following sections.

3.3.5.2 Types of Process Patterns (Ambler)

According to Ambler, process patterns are of three types [Ambler 1998a]. These types, in the ascending order of abstraction level, are as follows:

1. *Task Process Pattern*: depicting the detailed steps to execute a specific *task* of the process.

2. *Stage Process Pattern*: depicting the steps that need to be done in order to perform a *stage* of the process. A *stage* process pattern is usually made up of several *task* process patterns.
3. *Phase Process Pattern*: depicting the interaction of two or more *stage* process patterns in order to execute the *phase* to which they belong. Ambler believes that in any process (even object oriented ones), *phases* are performed in serial order, whereas the *stage* patterns inside them can be executed iteratively.

Ambler proposes many patterns of each type in his books, complete with detailed steps and guidelines for integrating and shaping the patterns into a comprehensive process [Ambler 1998a, Ambler 1999].

3.3.5.3 Object Oriented Software Process (Ambler)

Using his library of patterns, Ambler has proposed a general software development process, which he has called the Object Oriented Software Process (OOSP).

As shown in Figure 37, OOSP consists of four serial *phases*, each of which is made up of a number of *stages*. Each stage in turn consists of a number of *tasks*. All the phases, stages and tasks have been instantiated from Ambler's library of patterns according to guidelines provided in his method.

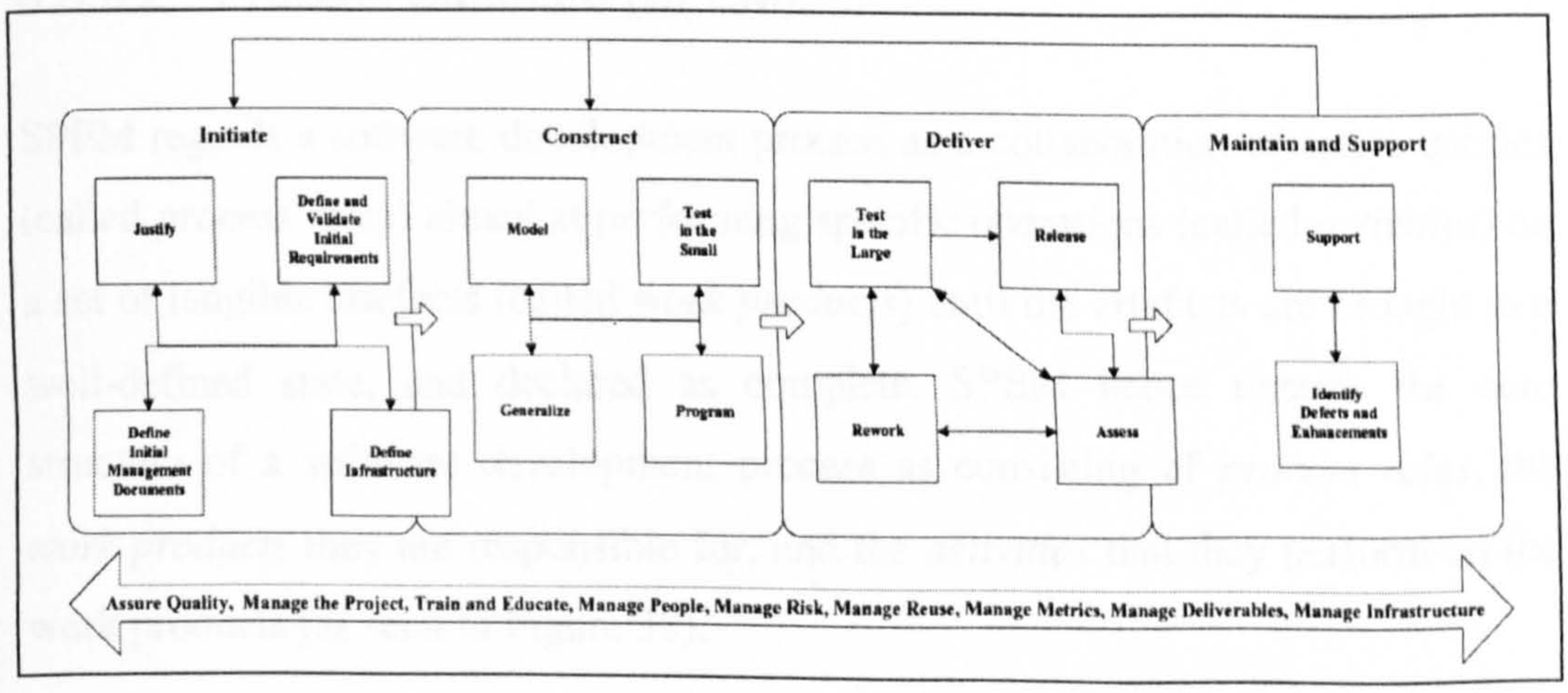

Figure 37. Ambler's Object Oriented Software Process (OOSP) [Ambler 1998a]

3.3.6 Process Metamodels

In a bid to highlight the high-level features of a process or family of processes, efforts have been made to apply abstraction to software development processes; *process metamodels* thus produced can be instantiated in order to produce concrete processes.

The two most well-known object-oriented process metamodels are the Open Consortium's *OPEN Process Framework (OPF)* [Firesmith and Henderson-Sellers 2001], and the OMG's *Software Process Engineering Metamodel (SPEM)* [OMG 2002]. OPF was briefly explained when describing the OPEN methodology. A brief overview of SPEM is given in the following sections.

3.3.6.1 The Software Process Engineering Metamodel (SPEM)

Similar in essence to OPF yet much simpler, SPEM is primarily based on Rational Corporation's *Unified Software Process Metamodel (USPM)* [Kruchten 2001]. USPM was chiefly intended as a metamodel for the RUP process; consequently, SPEM mainly supports the modeling of UML-based processes similar to RUP. Unlike OPF, SPEM does not include a process component library, nor does it offer a specific procedure for instantiating a software development process using the metamodel.

3.3.6.2 Process Structure (SPEM)

SPEM regards a software development process as a collaboration of active entities (called *process roles*) aimed at performing specific operations (called *activities*) on a set of tangible artefacts (called *work products*) until the artefacts are brought to a well-defined state, and declared as complete. SPEM hence regards the core structure of a software development process as consisting of *process roles*, the *work products* they are responsible for, and the *activities* that they perform on the work products (as seen in Figure 38).

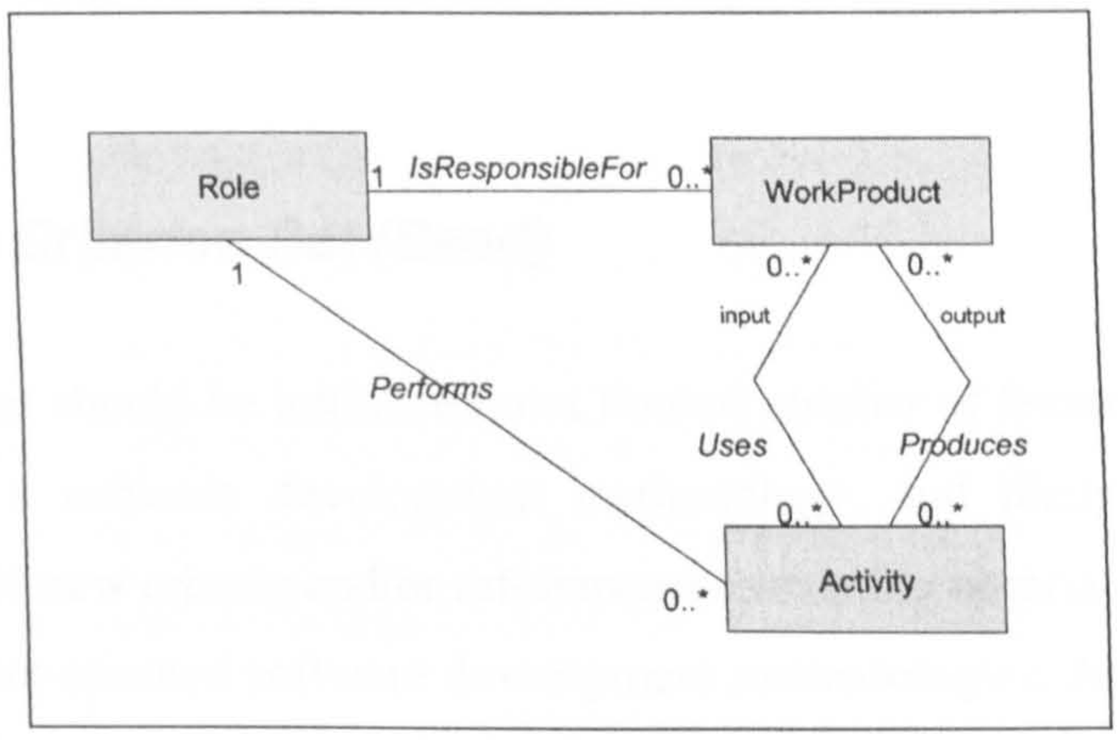

Figure 38. Core structure of a software development process, as defined by SPEM [OMG 2002]

The complete structure of a process in SPEM is actually much more complex than the core structure mentioned above. A *work product* may be composed of other work products, and can be associated with a state machine showing the states the work product can be in, and the permissible transitions between these states. *Activities* can be partitioned into *disciplines* based on the structural and functional themes that they have in common, and each activity may consist of atomic sub-activities called *steps*. An activity can have a *precondition* and a *goal* as constraints on its enactment, and may be associated with an *activity graph,* which shows the flow of steps in the activity.

In order to constrain the order in which the activities are performed, and to define the lifecycle structure of the process, SPEM incorporates definitions for *iteration*, *phase* and *lifecycle*, which are very similar to their corresponding definitions in RUP. The process structure proposed by SPEM also includes several abstract classes encapsulating the structural and behavioural commonalities of the various types of process elements. It also includes well-formedness rules, to be observed when instantiating processes.

3.4 Criteria-Based Evaluation

This section contains a description of the initial set of analysis criteria and the results of its application to the object-oriented methodologies, process patterns and process metamodels described in the previous section. The analysis results are

reported, along with the final criterion set, which is then used for defining the requirements.

3.4.1 Basic Criterion Set (Seed)

The criterion set should be initialized to a limited number of features known to be significant in a software development methodology, and likely to trigger the identification of new criteria and/or refinements to existing criteria when iteratively applied to object-oriented software development methodologies. As well as general traits and characteristics found in Software Engineering textbooks [Pressman 2004], features found desirable in object-oriented methodologies are also good candidates [Graham 2001]. The important point to have in mind is that the initial criterion set is to act as a "detonator": since the criteria are used as focus-pointers guiding the analysis process in exposing the processes' strengths and weaknesses, the initial criterion set should be expansive and incisive in order to trigger a large-scale fan-out effect, ever increasing the breadth and depth of the analysis, and thereby uncovering new criteria and refining the existing ones. The initial criterion set should therefore be expected to undergo dramatic changes – both in structure and content – during the analysis process.

The following were selected as initial criteria:

1. Coverage of standard software development activities: covering activities constituting or supporting the generic software development lifecycle [Pressman 2004].
2. Compactness of process: referring to lightness and simplicity of process, and its being free of nonessential, excess features; hefty and complex processes are hard to understand and master, and difficult to use.
3. Extensibility of process: the degree to which the process can be extended to support software development efforts of different sizes, complexities and criticalities.
4. Traceability of artefacts to requirements: the degree to which artefacts can be shown to have stemmed from the requirements.
5. Consistency of artefacts: mutual agreement and logical coherence of the artefacts.

6. Testability of artefacts: the degree to which artefacts lend themselves to establishment of test criteria and performance of tests to determine whether the test criteria have been met.
7. Tangibility and understandability of artefacts to users and developers: the level of consideration given to the balance between abstraction and concreteness in producing the artefacts – removal or reduction of low-level detail when appropriate and developing physical manifestations when necessary (e.g. prototyping) – with the ultimate objective of enhancing the perceptibility of the underlying notions.
8. Rationality of process and artefacts: evident rationality behind every task and the order in which the tasks are performed, and undeniable use for every artefact produced.

The seven characteristics listed in Chapter 1 – representing the core areas where OOSDMs need improvement – have been included in order to reveal what the existing processes lack or provide in this regard, thus unearthing features to exploit and pitfalls to avoid. The first criterion has been added in order to broaden the scope of the analysis to cover the whole lifecycle of the processes, and also to prompt scrutiny into the details of the activities performed.

3.4.2 Evaluation Results

As a result of iterative-incremental criteria-based analysis of the selected methodologies, process patterns and process metamodels according to the dynamic criterion set, significant strengths and weaknesses were identified, the final list of which is presented in the following subsections. Unlike many criteria-based analyses, the results are not represented as ratings denoting the degree of support each methodology provides for each of the criteria. As the criteria are used as focus-pointers, guiding the analyst towards potential areas of significant strength or weakness in the processes, there is no one-to-one relationship between the criteria and the results: a process might possess several significant strengths/weaknesses as pertaining to one criterion, while having nothing significant to offer in relevance to another criterion. Whereas a simple rating procedure would add nothing new to the criteria, the focus-pointing approach makes it possible to gradually increase the

span and depth of exploration and identify potential areas of improvement, thus facilitating the evolution of the criteria.

3.4.2.1 Seminal Methodologies

Shlaer-Mellor

Strengths

- ☑ Partitioning the system into domains, providing distinct layers from logical to physical
- ☑ Overall process generally governed by the domain structure; the base domain model is used as a focal point and as a high-level roadmap for development
- ☑ Intra-object behaviour accurately captured

Weaknesses

- ☒ Excessive number of models
- ☒ Complex behavioural modeling
- ☒ Bottom-up modeling: models are not based on system-wide behaviour or functionality; modeling starts from object and intra-object structure and then builds upward to inter-object and system-wide behaviour
- ☒ The modeling language used during architectural design lacks behavioural features, wrongly suggesting that architectural mechanisms have no distributed behaviour to add to the models.
- ☒ No modeling of physical configuration, i.e. processes (modules/components) and processors

Coad-Yourdon

Strengths

- ☑ Simple and well-defined process
- ☑ Seamless development phases based on layered construction of analysis models and tiered architecture of design

- ☑ Partitioned design corresponding to the three-tiered architecture
- ☑ Single notation for class structure and inter-object behaviour
- ☑ Layered construction of class diagrams
- ☑ Rich structural modeling

Weaknesses

- ☒ No basis in behavioural or functional requirements of the system (scenarios, use cases, responsibilities, etc.), resulting in poor requirement traceability
- ☒ Poor behavioural modeling
- ☒ Lack of formal features
- ☒ No modeling of physical configuration

RDD

Strengths

- ☑ Strong basis in system-level functionality (captured as *responsibilities*)
- ☑ Seamless development, although mostly limited to structural models
- ☑ Support for interfaces (called *contracts*); classes may have multiple interfaces
- ☑ One of the first instances of fractal (recursive) modeling: subsystems and classes are both treated as having interfaces

Weaknesses

- ☒ Process coverage limited to detailed analysis and design
- ☒ Poor behavioural modeling (almost nonexistent)
- ☒ Lacking in structural modeling features
- ☒ No formal features
- ☒ No modeling of physical configuration

Booch

Strengths

- ☑ Iterative-incremental (*Micro* in *Macro*)
- ☑ Rich structural and behavioural modeling features (*static/dynamic*)
- ☑ The Micro process puts precedence on identification of behaviour over identification of structure, avoiding unwanted/unneeded relationships
- ☑ Modeling support for physical structure (configuration)

Weaknesses

- ☒ Comparatively complex process
- ☒ Traceability to requirements is not straightforward
- ☒ Poor behavioural modeling at the problem-domain and system levels
- ☒ Inadequate formal features

OMT

Strengths

- ☑ Identification of physical architecture prior to detailed design of classes (*system design* before *object design*)
- ☑ Behavioural modeling at the system level (*scenarios* and *event-traces*)
- ☑ Functional modeling at the system level (DFDs)
- ☑ Rich structural and behavioural modeling features

Weaknesses

- ☒ OMT was a political solution intended to introduce OO into SA/SD communities, and as such, was a temporary remedy bound to be pushed aside upon widespread adoption of the object-oriented approach

- ☒ Inadequacy of the DFD as the functional modeling element (which does not exactly integrate well with other models; this ultimately led to the advent of OMT-2 in 1994 [Rumbaugh 1994]): orthogonal models need to converge somewhere along the process, or be oriented around or based on a common notion.
- ☒ Inter-object behaviour not modeled
- ☒ Lack of formal features
- ☒ No modeling of physical configuration

OSA

Strengths

- ☑ Rich structural and behavioural modeling features
- ☑ Explicit inter-diagram links (linking object interactions to object states and events)

Weaknesses

- ☒ No process
- ☒ Limited to analysis modeling
- ☒ Traceability and seamlessness not addressed (and is not applicable)
- ☒ Inadequate modeling of object interactions
- ☒ Inter-class details not adequately captured in the models
- ☒ Limited support for formality

OOSE

Strengths

- ☑ Strong basis in problem-domain modeling and functional modeling of the system (via use cases and domain object models)
- ☑ Traceability to requirements (via use cases)
- ☑ Seamless use case oriented development (despite a slight hiccup in the *Robustness Analysis* phase)
- ☑ Use-case oriented testing

- ☑ Rich functional and behavioural modeling (at the system-, inter-object-, and intra-object levels)
- ☑ Rich inter-object structural modeling

Weaknesses

- ☒ Typing of objects in the *robustness analysis* phase is somewhat premature, especially the introduction of *control* objects
- ☒ Poor intra-object structural modeling
- ☒ Modeling intra-object behaviour is deferred to late design, where events typically correspond to method invocations; doing the modeling earlier can be more helpful in understanding the requirements
- ☒ Lack of formal features
- ☒ No modeling of physical configuration

BON

Strengths

- ☑ Based on system-level behavioural modeling and requirements thereby identified
- ☑ Seamless development
- ☑ Customizable process through deliverable-based development: changing the order of the tasks is permissible as long as all deliverables are eventually produced.
- ☑ Ongoing refinement
- ☑ Rich structural and inter-object behavioural modeling
- ☑ Formal features (especially contracts)
- ☑ Good complexity management in structural models (via the notion of *cluster*)

Weaknesses

- ☒ Many deliverables, resulting in complexity
- ☒ Very limited functional modeling (use cases are casually described and tabulated)

- ☒ Intra-object behaviour not modeled
- ☒ Behavioural modeling starts late in the process: system-level behavioural deliverables, especially *event-charts*, which can be useful in identifying classes, are not produced until after class definition
- ☒ No modeling of physical configuration

Hodge-Mock

Strengths

- ☑ Based on structural and behavioural modeling at the system-level
- ☑ Seamless top-down development based on structure and behaviour of the system (gradual well-defined transition from system-level models to intra-object models)
- ☑ Traceability to requirements via evaluation scenarios based on scenarios of typical system usage
- ☑ Continual verification based on evaluation scenarios
- ☑ Using behavioural modeling in order to verify and refine the set of classes identified during structural modeling
- ☑ Rich functional modeling at the inter-object level
- ☑ Rich intra-object behavioural and functional modeling
- ☑ Rich structural modeling

Weaknesses

- ☒ Relatively complex process
- ☒ Poor inter-object behavioural modeling
- ☒ Inadequate functional modeling at the system level (limited to scenarios for evaluation)
- ☒ Lack of formal features
- ☒ Prohibitive number of diagrams and tables
- ☒ No modeling of physical configuration

Syntropy

Strengths

- ☑ Substantial formal features derived from Z
- ☑ Based on models of the problem-domain
- ☑ Overall simplicity as to diagram types
- ☑ Rich structural and behavioural modeling throughout
- ☑ Smooth seamless transition from logical (problem-domain level) to physical; through using *type-views* and *state-views* at all levels and gradually refining them throughout the process
- ☑ Well-defined rules for linking different models to each other
- ☑ Well-defined rules for transition from a logical view to its physical counterpart

Weaknesses

- ☒ Process coverage limited to analysis and design
- ☒ Traceability suffers from inadequate attention to functional modeling at the system level; especially lack of attention to usage scenarios
- ☒ Poor functional modeling
- ☒ Inter-object communication not modeled until the last stage of design
- ☒ No modeling of physical configuration

Fusion

Strengths

- ☑ Based on functional, behavioural and structural modeling of the problem-domain and the system
- ☑ Smooth transition from task to task and from phase to phase
- ☑ Traceability to requirements via scenarios of system usage
- ☑ Rich models (structural, functional, and behavioural)
- ☑ Support for formalism

- ☑ Strong functional and behavioural modeling at the system level through identifying detailed scenarios of interaction with the system at the system boundary
- ☑ Extra attention to details of inter-object visibility and the references that objects need to make to each other
- ☑ Detailed inter-object/inter-class models produced during design

Weaknesses

- ☒ Partial coverage of the generic analysis phase: the process starts when a preliminary informal requirements document is already available.
- ☒ Structural model identified during analysis is discontinued in the design phase, with its information broken down and then perfected, thus damaging seamlessness
- ☒ The number of diagrams and other deliverables is prohibitive
- ☒ No modeling of intra-class behaviour
- ☒ No intra-system behavioural and functional modeling during the analysis stage; this has been done intentionally, but nevertheless damages the comprehensiveness of analysis
- ☒ No modeling of physical configuration

3.4.2.2 Integrated Methodologies (Third Generation)

OPM

Strengths

- ☑ Simplicity of process
- ☑ Some degree of seamless development and traceability to requirements due to the singularity of the model type used (disrupted, though, because of OPD's limited modeling capacity)
- ☑ Innovative structural and functional modeling in a single type of diagram (OPD)
- ☑ Strong structural modeling at the inter-object level

Weaknesses

- ☒ Process is defined at a shallow level, with ambiguities and inadequate attention to detail
- ☒ Seamlessness and traceability are disrupted due to lack of behavioural models (especially at the inter-object and intra-object levels, directly affecting the identification and design of class operations)
- ☒ No basis in system-level behaviour and usage scenarios
- ☒ Poor behavioural modeling
- ☒ No formalism
- ☒ Poor intra-object structural modeling
- ☒ Models are prone to over-complexity
- ☒ No modeling of physical configuration

Catalysis

Strengths

- ☑ Based on requirements identified and modeled as system functionality and behaviour in the context of the problem domain: the system is modeled as a class – *type* – among other classes in the problem domain
- ☑ Seamless development through uniform approach to modeling at different levels
- ☑ Traceability to requirements via usage scenarios and use-case-based testing
- ☑ Gradual refinement from problem domain to the system boundary, then to the component architecture of the system, and finally to the class architecture of the components
- ☑ Process patterns identified for different kinds of projects
- ☑ Special attention to non-functional requirements
- ☑ Adequate complexity management
- ☑ Special attention to physical configuration of the system early in the process
- ☑ Smooth transition from logical to physical aspects

- ☑ Component based approach
- ☑ Fractal modeling
- ☑ Rich structural and behavioural modeling at all levels. Functional modeling limited to UML's capabilities

Weaknesses

- ☒ Heavy process; fractal modeling and process patterns help, but are not enough
- ☒ Focus mostly confined to business systems, more or less limiting the applicability of the process

OPEN

Strengths

- ☑ Flexibility and configurability due to the framework definition of the process
- ☑ Well-defined framework (generally and in detail) for instantiating tailored-to-fit processes
- ☑ Accommodates seamless process configurations
- ☑ Accommodates process configurations supporting traceability
- ☑ Accommodates various lifecycles, including iterative-incremental
- ☑ Covers enterprise-level activities and business-process-reengineering
- ☑ Incorporates a rich library of process components
- ☑ Provides guidelines as to how customized processes should be built (especially how stages should be structured and organized)
- ☑ Accommodates comprehensive modeling at all levels (problem domain to objects; logical to physical)
- ☑ Rich modeling-language support (UML and OML)

Weaknesses

- ☒ As a result of merging various methodologies, OPEN is not a specific methodology, but rather a process framework; in trying

to remain noncommittal to any single process, it has lost concreteness.

☒ OPEN is huge and complex; many developers tend to use typical instances introduced by the authors rather than instantiate their own.

☒ The developer is responsible for constructing the methodology, and even though OPEN prescribes the framework, components, and guidelines as to how to construct the process, bad instances *can* be built (very much like a Lego game).

RUP/USDP

Strengths

☑ Iterative-incremental process

☑ Well-documented process

☑ Based on functional, behavioural, and structural modeling of the problem domain and the system

☑ Traceability supported through use cases

☑ Seamlessness (though with hiccups, e.g. transforming use cases to sequence diagrams)

☑ Architecture-centric process (which necessitates early specification of an architectural blueprint)

☑ Customizability addressed

☑ Risk-based development, aimed at mitigating the risks before undertaking the tasks

☑ Support for structural, behavioural and functional modeling at all levels (problem domain to objects; logical to physical)

☑ Rich modeling language (UML), especially in structural and behavioural modeling features

☑ Support for formalism (through UML/OCL)

Weaknesses

☒ Very complex process

- ☒ The process is confusing to those involved: it is hard to understand the logic behind some of the deliverables and tasks performed. The iterative-incremental nature of the process further complicates the issue.
- ☒ Although advertised as customizable, configuring the process is a formidable task in itself. Trying to tailor down the process often has the opposite effect.
- ☒ Since the process is very complex, not having a maintenance phase, on the grounds that it can be performed by iterating the whole process as a cycle, is not convincing.
- ☒ Prohibitive number of models
- ☒ Strict adherence to UML, which is not necessarily constructive, especially since UML is not perfect and can exacerbate the model inconsistency problem.
- ☒ Substantial potential for inconsistency of models

EUP

Strengths

- ☑ Same benefits as RUP
- ☑ Addresses enterprise-level issues
- ☑ Maintenance is a phase in its own right.
- ☑ Attention is given to post-mortem activities when retiring the project (in the form of a new *Retirement* phase).
- ☑ Not strictly adherent to UML; other modeling languages such as DFDs are also used.

Weaknesses

- ☒ Like RUP, EUP is
  - very complex
  - encumbered with a prohibitive number of models
  - suffering high potential for model inconsistency
  - confusing as to the process used
  - hard to customize

- ☒ EUP has added further complexity to RUP by adding two new phases and two new disciplines.
- ☒ Adding the maintenance phase is not sufficient, since any change needed will result in a restart of the development process.

FOOM

Strengths

- ☑ Based on functional and structural modeling of the problem domain and the system
- ☑ Traceability to requirements (via *transactions*)
- ☑ Appealing to domain experts and the SA/SD community (due to the popularity of DFDs)
- ☑ Attention to interface design and I/O design based on the *transactions* identified and the OO-DFDs

Weaknesses

- ☒ No implementation, deployment and maintenance phases
- ☒ Only suitable for data-intensive information systems
- ☒ Seamlessness suffers because OO-DFDs are not exactly object-oriented.
- ☒ The process is vague in how *operations* and *transactions* extracted from the OO-DFDs are assigned to classes; this is the same problem that triggered the demise of DFDs in OO methodologies, after transformative methodologies and OMT failed to resolve the issue. Using DFDs in an OO context without solving the problem of mapping (data-stores to classes and processes to operations) and assignment (operations to classes) will most probably result in failure.
- ☒ No modeling of logical architecture and physical configuration
- ☒ Poor behavioural modeling (performed only in later stages of design at the inter-object level)
- ☒ Lack of formalism

☒ The issue of design-level refinements to the *Data Model* (class diagram) is not properly addressed (only "Form", "Menu", "Report", and "Transaction" classes are added).

3.4.2.3 Agile Methodologies

DSDM

Strengths

☑ Iterative-incremental process
☑ Based on functional and structural modeling performed on the problem domain and the system
☑ Early specification of the physical architecture
☑ Flexible and configurable process (through defining the main development cycle as consisting of interwoven Analyze-Design-Implement cycles)
☑ Carefully worked-out process
☑ Especially suitable for projects with highly volatile requirements, since it is easily adaptable
☑ Seamless development through using prototypes
☑ Incorporating a *Suitability Filter* to make sure that the project can be carried out with DSDM
☑ Based on careful planning
☑ Test-based development
☑ Active user involvement
☑ Reversibility of changes
☑ Early and frequent releases
☑ Smooth transition from stage to stage
☑ Traceability to requirements achieved through constant testing and via the prototype produced (the prototype is the manifestation of the requirements and will ultimately evolve into the final system)
☑ Based on prioritization of requirements by categorizing them into specific types
☑ Design-based development

Weaknesses

- ☒ Not scalable
- ☒ Limited applicability scope: the project should lend itself to RAD through evolutionary prototyping.
- ☒ Stringent constraints on time and resources
- ☒ Severe model-phobia: text reports are abundant but visual models are avoided unless absolutely essential. The prototype is considered the main model.
- ☒ Lack of formalism

Scrum

Strengths

- ☑ Iterative-incremental process
- ☑ Based on modeling the problem domain and the system
- ☑ Requirements are allowed to evolve over time.
- ☑ Traceability to requirements through the *Product Backlog*: the repertoire of requirements which all the stages are based upon
- ☑ Architecture of the system drafted before the development engine is started
- ☑ Iterative development engine governed by careful planning and reviewing
- ☑ Active user involvement
- ☑ Simple and straightforward process
- ☑ Early and frequent releases, demonstrating functionality at the end of each iteration (*sprint*) of the development cycle

Weaknesses

- ☒ Integration is done after all increments are built
- ☒ Lack of scalability
- ☒ Based on the assumption that human communication is sufficient for running projects of any size and keeping them focused
- ☒ Not necessarily seamless (details of tasks are not prescribed)

- ☒ No clear-cut design effort
- ☒ Model-phobic
- ☒ Models are not prescribed, leaving it to the developer to decide what model can be useful.
- ☒ Lack of formalism

XP

Strengths

- ☑ Iterative-incremental process
- ☑ Based on system functionality captured in *User Stories*
- ☑ The process is tuned according to feedback during its execution
- ☑ Traceability to requirements through the use of *user stories* throughout the process as the basis for tasks and tests
- ☑ Based on system architecture (*Metaphor*) identified through prototyping
- ☑ Active user involvement
- ☑ Test-based development
- ☑ Stringent standards enforced on coding
- ☑ Early and frequent releases
- ☑ Requirements are allowed to evolve over time
- ☑ Iterative development engine governed by careful planning and reviewing
- ☑ Explicit coverage of maintenance and project retirement ("*Death*") phases; maintenance in fact comprises the bulk of the development effort
- ☑ Continuous validation
- ☑ Continuous integration
- ☑ Refactoring exercised in order to acquire the simplest code possible

Weaknesses

- ☒ Process is rather vague: the process commonly introduced as the "XP Process" is just a typical example.

- ☒ More intended as a set of principles and practices rather than a methodology
- ☒ Limited evidence of scalability
- ☒ Seamlessness is not addressed: development is more or less a jump from *user stories* to code, and the little design that is done (if at all) does not have to conform to any standard.
- ☒ Requires the use of automated tools and enforcement of discipline for "*Collective-Code-Ownership*" to be practicable.
- ☒ No clear-cut design effort
- ☒ Model-phobic
- ☒ Except for CRC cards, models are not prescribed, leaving it to the individual developer to decide what model is useful to him.
- ☒ Lack of formalism

ASD

Strengths

- ☑ Iterative-incremental process
- ☑ Based on structural, functional and behavioural modeling of the problem domain and the system
- ☑ Well-worked-out process
- ☑ Special attention to quality assessment and control (Q/A is performed at all levels: per-project and per-iteration)
- ☑ Component-based development
- ☑ Adaptive (tuneable) process; through risk-driven planning, conducting reviews, and revising the plans and the development process according to what has been learnt during the iterations
- ☑ Extensive use of JAD sessions for information gathering and decision making
- ☑ Stress on the importance of a collaborative environment for the development to be successful: a *User Community* is established and a suitable medium of collaboration (methods, tools and procedures) is set up.
- ☑ Test-based development

- ☑ Refactoring for simplifying the code
- ☑ Continuous integration
- ☑ Stress on parallel development of components by collaborating teams of developers, thus speeding up the process
- ☑ Traceability to requirements through ongoing validation and quality review

Weaknesses

- ☒ Not scalable
- ☒ Over-dependence on inter-human communication
- ☒ Need for intensive project monitoring and control in order to maintain inter-team and intra-team collaboration during component development
- ☒ Seamlessness not addressed
- ☒ No clear-cut design effort
- ☒ Model-phobic
- ☒ No specific models prescribed
- ☒ Physical configuration modeling is ignored (even though necessary in component-based development).
- ☒ Lack of formalism

dX

Strengths

- ☑ Iterative-incremental process
- ☑ Based on system architecture identified through prototyping
- ☑ Prototyping results used in planning and scheduling
- ☑ Prototypes compensate for lack of analysis modeling
- ☑ Based on system functionality captured in use cases
- ☑ Traceability to requirements through the use of use cases throughout the process as the basis for tasks and tests
- ☑ Advantageous development practices borrowed from XP (test-based development, early and frequent releases, active user

involvement, refactoring for achieving code simplicity, continuous integration, and stringent coding standards)

- ☑ Design-based development; *design sessions* are held in order to decide on how the use cases should be implemented to fit the system architecture.
- ☑ Iterative development engine governed by planning and reviewing
- ☑ Seamlessness observed (though limited) due to use-case based activities throughout the process, and design-based development
- ☑ Risk-based process
- ☑ Not particularly model-phobic
- ☑ Formal features can be added via UML/OCL

Weaknesses

- ☒ Lack of detailed descriptive documentation on the methodology
- ☒ Not scalable
- ☒ *Transition* is defined as a phase solely in order to remain compliant with RUP; whereas it is, for the large part, a per-iteration activity.
- ☒ Poor analysis modeling is likely to have an adverse effect on the design activity.

Crystal

Strengths

- ☑ Iterative-incremental process
- ☑ Continuous integration
- ☑ Iterative development engine governed by planning and reviewing
- ☑ Flexible and configurable process (in each methodology): methodologies are tuned through gradual perfection and revision based on cyclic reflection workshops
- ☑ Methodologies used for a low-criticality project can typically be tuned to fit a higher-criticality project (if the criticality level is

supported by the methodology), provided that the project size is not increased dramatically

- ☑ Active user involvement
- ☑ Early and frequent releases
- ☑ Scalability (though limited) through using different methodologies for different project sizes
- ☑ Continuous validation
- ☑ Specific work-products prescribed, though details and templates are left to the developers to decide
- ☑ [Crystal Clear] Traceability to requirements (though limited) through continuous validation and quality reviews
- ☑ [Crystal Clear] Requirements are allowed to evolve over time
- ☑ [Crystal Clear] Preliminary feasibility analysis conducted as a risk mitigation mechanism
- ☑ [Crystal Clear] Based on system functionality, typically captured in use cases
- ☑ [Crystal Clear] Based on structural modeling of the problem domain
- ☑ [Crystal Clear] Based on a system architecture identified and refined during the process
- ☑ [Crystal Clear] Test-based development
- ☑ [Crystal Clear] Design activities encouraged, with results documented as *Design Notes*

Weaknesses

- ☒ Only limited scalability
- ☒ Lack of an unambiguous common process
- ☒ Limited applicability: not suitable for developing highly critical systems
- ☒ Over-dependence on inter-human communication
- ☒ [Crystal Clear] Seamlessness not addressed
- ☒ [Crystal Clear] traceability to requirements suffers because planning and development activities are not necessarily

requirements-based (e.g. *Blitz Planning* is task-based rather than requirements-based).

☒ [Crystal Clear] Design activities are carried out by individual developers in the manner they choose; design is not performed as a team effort with globally available results based on which implementation can be carried out uniformly.

☒ [Crystal Clear] Since the detailed nature of many work-products is left to the individual developers to decide, behavioural and functional modeling can be poor throughout the process.

☒ [Crystal Clear] No formalism

FDD

Strengths

☑ Iterative-incremental process

☑ Based on a general layered architecture for systems

☑ Based on structural and behavioural modeling of the problem domain

☑ Based on system requirements captured as *Features*

☑ Traceability implemented through using *features* as a basis throughout the process

☑ Simple and straightforward process, yet well thought-out

☑ Continuous integration

☑ Seamlessness observed throughout the process via *feature*-based modeling activities

☑ Design-based development

☑ Continuous validation

☑ Frequent deliveries once the iterations start

☑ Complexity management at the *features* level through layering

☑ Only mild model-phobia

☑ Modeling at the problem-domain-, system-, inter-object-, and intra-object levels

☑ Group modeling used as a technique for putting all involved in the overall picture

☑ Iterative modeling in order to enhance the accuracy, completeness and consistency of the models

Weaknesses

☒ Does not cover post-implementation activities and preliminary analysis.

☒ Lacks adaptability due to inexistence of iteration-level planning, reviewing and revision.

☒ Intensive project supervision is essential

☒ No formalism

3.4.2.4 Process Patterns

Ambler

Strengths

☑ Comprehensive and detailed specification document

☑ Full coverage of generic development lifecycle activities

☑ Iterative-incremental process

☑ Full support for umbrella activities

☑ Requirements-based development

☑ Based on functional, behavioural, and structural modeling of the problem domain and the system.

☑ Accommodates comprehensive modeling at all levels (enterprise to problem domain to system objects; logical to physical).

☑ Rich modeling-language support (UML), especially in structural and behavioural modeling features

☑ Support for formalism (through UML/OCL)

☑ Traceability supported through use cases

Weaknesses

☒ Process patterns are not defined as individual patterns, but as components of a specific object oriented methodology (OOSP);

this enhances the tangibility of the patterns but damages their generality and applicability.

- ☒ Very complex process (OOSP)
- ☒ Configurability not addressed
- ☒ Seamlessness damaged due to hitches in model mapping
- ☒ Prohibitive number of models
- ☒ Substantial potential for inconsistency of models

3.4.2.5 Process Metamodels

SPEM

Strengths

- ☑ Flexibility and configurability due to the generality of the metamodel (albeit limited, because of dependence on RUP as a metamodel basis)
- ☑ Well-defined general framework
- ☑ Provision of well-formedness rules to be observed when instantiating processes

Weaknesses

- ☒ Lack of a specific instantiation procedure
- ☒ Lack of a detailed specification document: the specification document adopted by the OMG is a very general description of the metamodel.
- ☒ Lack of subtyping for important process components (let alone a component library), which makes the metamodel of very little practical use. Consequently:
  - Poor coverage of lifecycle activities
  - Lack of explicit support for umbrella activities
  - Modeling and artefact production issues not explicitly addressed

- ☒ Mainly targets the modeling of processes similar to RUP, hence limiting applicability and generality (even the terminology is that used in RUP).
- ☒ The developer is responsible for constructing the methodology, and well-formedness rules are not enough to prevent bad instantiations.

3.5 Final Criterion Set

The final, stabilized version of the criterion set, refined as the result of iterative-incremental application to the selected methodologies, process patterns and process metamodels, is as follows:

1. *Process*
    - 1.1. Clarity, rationality, accuracy, and consistency of definition
    - 1.2. Coverage of the generic development lifecycle activities (Analysis, Design, Implementation, Test, Maintenance)
    - 1.3. Support for umbrella activities, especially including:
        - 1.3.1. Risk management
        - 1.3.2. Project management
        - 1.3.3. Quality assurance
    - 1.4. Seamlessness and smoothness of transition between phases, stages and activities
    - 1.5. Basis in the requirements (functional and non-functional)
    - 1.6. Testability and Tangibility of artefacts, and traceability to requirements
    - 1.7. Encouragement of active user involvement
    - 1.8. Practicability and practicality
    - 1.9. Manageability of complexity
    - 1.10. Extensibility/Configurability/Flexibility/Scalability
    - 1.11. Application scope
2. *Modeling Language*
    - 2.1. Support for consistent, accurate and unambiguous object-oriented modeling:

2.2. Provision of strategies and techniques for tackling model inconsistency and managing model complexity

The final criteria satisfy the validity meta-criteria of [Karam and Casselman 1993], in that they are:

- *general* enough to be used for evaluating all object-oriented software development methodologies,
- *precise* enough to help discern and highlight the similarities and differences among object-oriented software development methodologies,
- *comprehensive* enough to cover all significant features of object-oriented software development methodologies, and
- *balanced*: adequate attention has been given to all three major types of features in a methodology: *technical*, *managerial* and *usage* [Karam and Casselman 1993].

3.6 Requirements

The final criterion set and the analysis results can be used for defining a set of requirements for object-oriented software development methodologies, as suggested by the following observations:

- Since the analysis criteria can be regarded as a framework defining the general features desirable in an object-oriented methodology, requirements for such a methodology can be built by detailing and enriching these features with information on the degree of support expected in the target methodology. Consider risk-management as an example of an analysis criterion: in order to evolve it into a requirement, the degree of risk management support that the target methodology is expected to provide should be defined.
- Development processes offer alternative ways for implementing desirable features; analysis results, when enriched with information as to how criteria are met or contradicted, provide a toolkit of methods and techniques for implementing features, as well as a list of potential pitfalls. The repertoire of ideas thus built (containing lessons learnt from existing

software development processes, i.e. features to use and pitfalls to avoid) can guide the developers in defining and refining the requirements.

Thus, using the final criterion set as the basis, and applying the lessons learnt from the results of the criteria-based analysis of software processes, the following requirements have been identified for the target object-oriented software development methodology:

1. ***Process***
    1.1. **Definition:** the methodology should be well-documented (comprehensive, clear, rational, accurate, detailed and consistent description should be provided):
        1.1.1. What should be captured? Lifecycle and work-units, producers, modeling language, work-products, techniques and rules, and issues pertaining to umbrella activities. Metamodels suggested by SPEM and OPF provide useful information as to what should be captured in the definition.
        1.1.2. How? Mainly *process-centred*: the structure of the documentation should closely resemble that of the lifecycle, and everything should be described as secondary to the work-units (phases, stages and activities) of the lifecycle. Gradual refinement (hierarchical layering) should be used in describing the process. Since object-oriented process metamodels – such as SPEM and OPF – regard processes as mainly consisting of *work-units*, *roles* (producers), and *products* (artefacts), the definition of the methodology should also provide a view focusing on the producers involved in the methodology (describing the work-units they participate in and the artefacts they produce) as well as a view focusing on the artefacts produced (describing the work-units where they are produced and the producers involved).

1.2. **Coverage:** the generic software development lifecycle activities (*Definition*, *Development*, and *Maintenance*) should be covered. Fusion, RUP, EUP and Catalysis are examples of methodologies providing extensive coverage. Close examination of the generic software development lifecycle [Pressman 2004, Sommerville 2004], Ambler process patterns [Ambler 1998a, Ambler 1999], and the OPEN Process Framework (OPF) [Firesmith and Henderson-Sellers 2001] shows that the following activities should be covered as a minimum:

1.2.1. Definition

1.2.1.1. Problem domain exploration and modeling

1.2.1.2. Requirements elicitation

1.2.1.3. Feasibility analysis

1.2.2. Development

1.2.2.1. Architectural Design

1.2.2.2. Detailed Design

1.2.2.3. Programming

1.2.2.4. Test

1.2.2.5. Deployment

1.2.3. Maintenance

1.3. **Support for umbrella activities:** especially including:

1.3.1. **Risk management:** through risk assessment and risk mitigation activities incorporated into the lifecycle. Of special importance are techniques proven effective in other methodologies: e.g. preliminary feasibility analysis (as seen in OPEN, Crystal Clear, and DSDM), prototyping (e.g. RUP, DSDM, XP, and dX), risk-based planning (e.g. RUP, DSDM and Scrum), iterative-incremental development (e.g. RUP and agile methodologies), active user involvement (e.g. Scrum and FDD), continuous verification and validation (e.g. Hodge-

Mock, XP and ASD), iterative process/product/plan reviews (e.g. ASD, Scrum and Crystal), early releases (e.g. XP and Scrum), and continuous integration (e.g. XP and FDD).

1.3.2. **Project management:** through planning, scheduling and control techniques incorporated into the process (as in RUP and EUP; DSDM and Scrum are good agile examples). Provision should be made for the plans and schedules to be iteratively revisited and revised based on experience gained through the development (as in EUP, ASD and Scrum). Special attention should be given to team management aimed at enhancing intra-team and inter-team communication and collaboration (as seen in RUP, EUP, Scrum and FDD).

1.3.3. **Quality assurance:** through quality assessment and enhancement techniques incorporated into the process. Of special importance are techniques proven effective in other methodologies: e.g. iterative technical reviews (as seen in agile methodologies; e.g. Scrum and Crystal), design by contract (e.g. BON), continuous verification and validation (e.g. Hodge-Mock, XP and ASD), and strategies/techniques enhancing requirements traceability (e.g. use-case-driven methodologies such as OOSE and RUP, scenario-based methodologies such as Hodge-Mock, and agile methodologies such as XP and FDD).

1.4. **Seamlessness and smoothness of transition between phases, stages and activities:** Although seamlessness can be incorporated via basing all tasks and artefacts on a common concept (e.g. classes in BON, the Domain Model in Shlaer-Mellor, and use cases in RUP), the transition between phases, stages and activities is not necessarily smooth, since it might

involve the production of brand new artefacts; even though not violating seamlessness, the effort that is typically required damages smoothness of transition. An alternative seamless strategy is continuous refinement of a specific set of models, around which the development tasks are oriented, which provides both seamlessness and smoothness of transition (as used in Coad-Yourdon, Syntropy and Catalysis). Fractal modeling (as in Catalysis) is an example of a technique that is particularly successful in this context. It should be noted that all methodologies providing smooth transition are not necessarily seamless; many agile methodologies provide smooth transition because of the iterative-incremental nature of their development strategy and the short cycles they usually have, yet they cannot always be considered seamless, since there can be a huge gap between analysis and implementation.

1.5. **Basis in the requirements (functional and non-functional):** functional and non-functional requirements should be captured early in the process, modeled in their own right, and used as a basis for design and implementation (Coad-Yourdon is an example of a methodology that neglects this seemingly obvious requirement); use-case-driven methodologies such as Catalysis and RUP, and agile methodologies such as FDD and Scrum are good examples of successful methodologies in this regard. Requirements should be allowed to evolve during the process, as is the case in many agile methodologies.

1.6. **Testability and tangibility of artefacts, and traceability to requirements:** artefacts should be few, simple, and understandable, with dependencies that are minimal and clearly defined (Catalysis is a good example, as are many seminal methodologies, e.g. BON and Fusion). Artefacts should complement each other in the context of the process, not decorate each other with clutter. Tangibility of the artefacts to the users and the developers should be maximized: executable artefacts and artefacts with syntax and semantics

understandable to the user are tangible to the user, while developers find those artefacts tangible that are visibly useful in the process (otherwise they will be ignored or botched, and quality may suffer as a result). Artefacts should be traceable to the requirements (e.g., as direct or indirect realizations of the requirements – as in RUP, or via the use of requirements-based evaluation scenarios – as in Hodge-Mock).

1.7. **Encouragement of active user involvement:** which is vital for risk management and quality assurance. Ambassador users, and planning and review sessions with user participants are proven techniques [Highsmith 2002]. Agile methodologies have a great deal to offer in this regard.

1.8. **Practicability and practicality:** the methodology should be employable; and effectively, efficiently and usefully at that. Over-complex methodologies are not practicable; configurability does not solve the problem since it typically involves complex procedures (as is the case with RUP), and neither do instantiation frameworks (like OPEN), for the same reason. Practicability can also depend on the project in hand; performing a feasibility analysis task early in the process (possibly involving the deployment of suitability filters) may prove essential. There are numerous factors, other than complexity, that affect practicality (some adversely), and should therefore be taken into account. Tasks that distract the developers from mainstream activities or encumber them with impertinent or unnecessary details should be deleted; techniques and strategies for focusing the development, such as requirements-based models (such as those seen in Fusion, Catalysis and FDD), system architecture/metaphor (such as those seen in RUP and XP), and team management sessions (such as those seen in Scrum and FDD) seem to be promising techniques in this context. Dependence on error-prone techniques and strategies can damage practicality (such is the over-dependence of some agile methods on the efficacy of

human communication, and dogged adherence of some integrated methodologies to UML). Dependency on special tools and technologies can also be detrimental to practicality. A very important factor affecting practicality is the project management strategy; lack of adequate management measures can render the methodology impractical or even impracticable, especially in large projects with stringent constraints on time and resources.

1.9. **Manageability of complexity:** the complexity of work-units should be manageable, e.g. via partitioning and layering. Catalysis is a particularly successful example.

1.10. **Extensibility/Configurability/Scalability/Flexibility:** the process should be an extensible core, with extension points and mechanisms explicitly specified. It is desirable to be able to configure the extensions or even the core itself in order to fit it to the project in hand (process patterns can be useful in this context). The methodology should be applicable to projects of different sizes and criticalities (as seen in integrated methodologies such as RUP and Catalysis, as well as some agile ones such as FDD). It should also be dynamically flexible: it should be possible to tune the methodology according to the experience gained during the development; useful techniques are iterative process review sessions, and feedback-based revisions (as seen in ASD and Crystal Clear); it should be noted, however, that tuning is a project-wide decision, and individual teams and developers should not be allowed to make alterations with possible project-wide implications.

1.11. **Application scope:** the application scope depends on the intended usage context, yet targeting information systems as a general usage context seems to be a logical minimum requirement, as this is likely to address the minimum modeling needs of a general methodology. The application scope in the context of this thesis is initially limited to information systems,

but can later be expanded depending on the outcome of the design.

2. *Modeling Language*

2.1. **Support for consistent, accurate and unambiguous object-oriented modeling:** specifically covering:

2.1.1. Diverse modeling viewpoints: Structural – Functional – Behavioural (as seen in UML, and the modeling languages of OMT and OSA)

2.1.2. Logical to Physical modeling: Business-Process/Problem Domain to Solution Domain to Implementation Domain (as seen in UML and OPEN/OML)

2.1.3. Diverse levels of abstraction and granularity: Enterprise level – System level – Subsystem/Package level – Inter-object level – Intra-object level (as seen in UML, and the modeling languages of Hodge-Mock and Fusion)

2.1.4. Formal and Non-formal specifications (as seen in UML/OCL, and the modeling languages of BON and Syntropy)

Although UML is rich and extensible enough to provide ample support, strict adherence to UML should not be enforced. The use of data-flow diagramming for functional problem-domain modeling – as seen in EUP and FOOM – is a successful example of complementing UML with other modeling languages.

2.2. **Provision of strategies and techniques for tackling model inconsistency and managing model complexity:** tackling model inconsistency is usually up to the process component of the methodology rather than the modeling language; yet modeling languages can facilitate consistency-checking through providing semantics which define model dependencies and constraints. UML lacks such semantics [OMG 2004], leaving it to the methodology process to define them; Catalysis is an

example of a successful process in this regard. However, modeling languages proposed by many seminal methodologies offer such semantics (examples include BON and Fusion). Another noteworthy contribution in this regard is OPM's single-model approach, which facilitates consistency-checking through eliminating model multiplicity. Modeling languages should also include constructs facilitating complexity management; UML's *package* and *component* elements are apt examples.

3.7 Summary

The analysis phase produces the requirements of the target OOSDM through analyzing existing object-oriented methodologies, process patterns and process metamodels. The analysis process adopted in this thesis starts with process-centred review of the methodologies, patterns and metamodels, resulting in descriptions which highlight the processes and prepare them for critical examination.

The processes are then scrutinized according to a set of criteria. The criteria-based analysis approach adopted in this thesis is based on iterative review of the processes, thereby incrementally identifying the strengths and weaknesses of the processes, perfecting the set of criteria along the way. The products of this analysis process are the analysis results (a list of strengths and weaknesses for the processes), and the refined criterion set.

The requirements are produced through specifying the degree of support expected to be provided by the target methodology for each criterion in the final criterion set. The list of strengths and weaknesses is also used in the definition of the requirements: the strengths and weaknesses identified in existing processes show how processes meet or fail the requirements, and can therefore be used for providing a more detailed definition of the requirements through supplying instances from existing processes.

The set of requirements produced in the Analysis phase is fed into the Design-Implementation-Test cycle of the methodology development lifecycle (introduced

in Chapter 1). This iterative development engine designs, implements and validates the target methodology based on the set of requirements.

Chapter 4

Design

The design phase focuses on determining a blueprint for the methodology based on the requirements defined during analysis (Section 3.6). The process-centred descriptions (Section 3.3) and the criteria-based analysis results (Section 3.4.2) provide a rich repertoire of ideas and techniques to be used in the design. The first task of this phase, however, is to determine an appropriate design method, which is then applied for producing the blueprint of the methodology.

4.1 Alternative Design Methods

The following methods were identified as alternative ways of designing the target methodology:

1. *Instantiation approach*: instantiating an already available process metamodel (reviewed in Section 3.3.6)
2. *Artefact-oriented approach*: devising a seamless complementary chain of artefacts and building the process around it
3. *Composition approach*: using one of the already available libraries of process patterns (reviewed in Section 3.3.5)
4. *Integration approach*: integrating features, ideas and techniques from existing methodologies (merits of which were discussed in Section 2.4)

As pointed out in Section 2.4, the *Instantiation* and *Composition* approaches are correspondingly analogous to the *Paradigm-based* and *Assembly-based* approaches of Method Engineering, but the *Integration* and *Artefact-oriented* approaches are relatively novel in this context. Any of these approaches can be used for designing the methodology, but since the approach undertaken should be flexible and versatile enough to make use of all of the merits that different approaches have to

offer, a *Hybrid approach* has been devised, using different alternatives from among the above-mentioned for different parts of the process and/or at different levels of abstraction.

4.2 The Hybrid Design Process

The hybrid design process has been devised as a top-down iterative-incremental process. The iterative-incremental engine at the core of the design process generates the methodology in a top-down fashion – from general lifecycle to finer grained detail of process phases and activities – using the requirements, methodology descriptions and methodology analysis results as a basis. The design approaches used in each iteration are determined according to the scope and abstraction level of the design activity undertaken in the iteration.

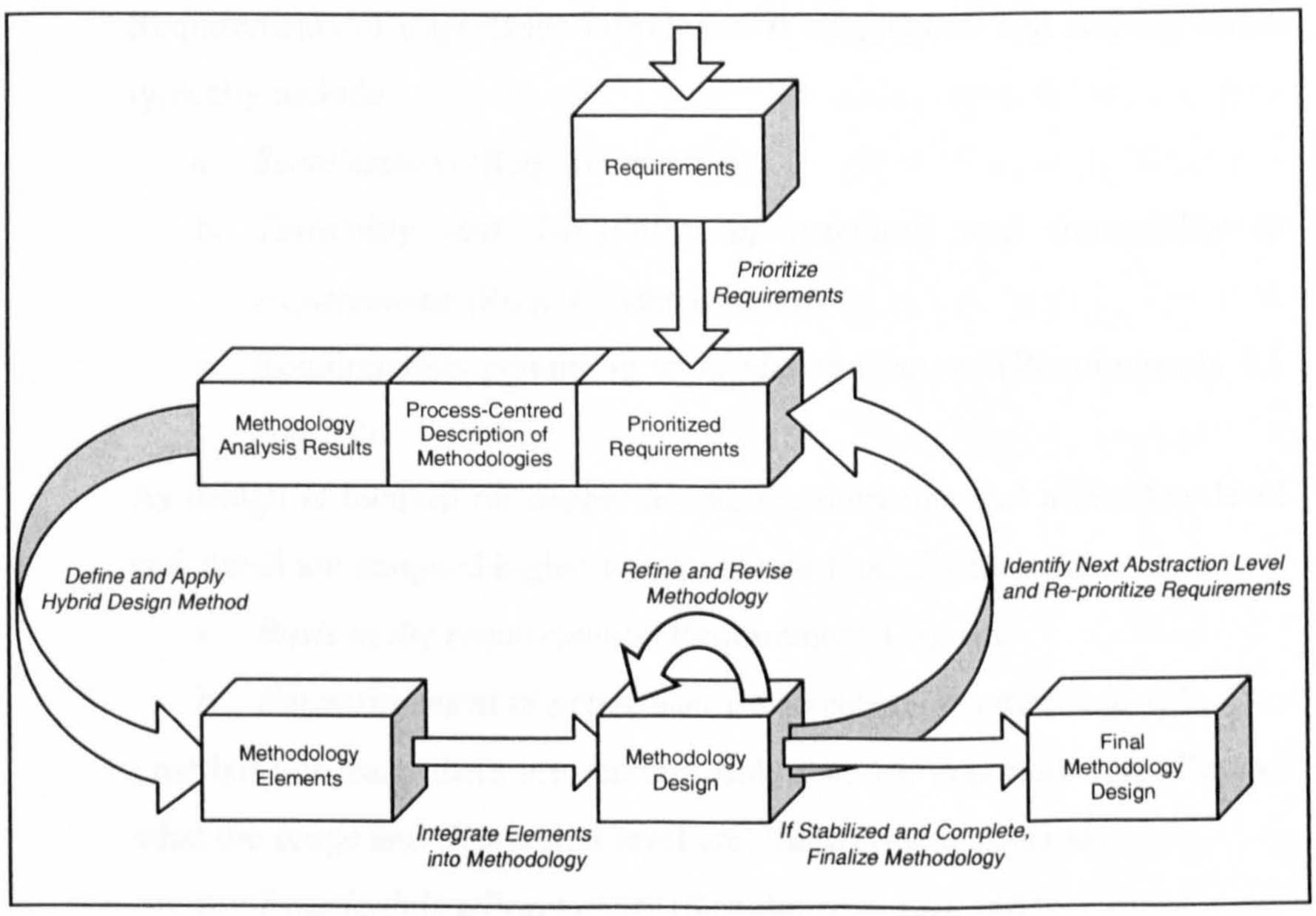

Figure 39. The hybrid design process

The hybrid design process consists of the following tasks (Figure 39):

1. *Prioritization of the requirements*: performed at the start of the process and repeated at the end of each iteration, prioritization orders the requirements according to their relevance to the current scope and level of abstraction,

thus focusing the design process on satisfying requirements of higher significance. At the start of the process, abstraction is at its highest level and the scope encompasses the whole lifecycle, therefore requirements with lifecycle-level impacts are given precedence over others; these typically include:

a. *Coverage of generic software development lifecycle activities* (Requirement 1.2),
b. *Support for umbrella activities* (Requirement 1.3),
c. *Extensibility/Configurability/Flexibility/Scalability* (Requirement 1.10), and
d. *Application scope* (Requirement 1.11).

As iterative design progresses to lower levels of abstraction and the scope is focused on individual subprocesses and their internal activities, priority is gradually shifted to requirements affecting finer-grained aspects. Requirements of importance introduced at subprocess- and activity levels typically include:

a. *Seamlessness* (Requirement 1.4),
b. *Testability and Tangibility of artefacts, and traceability to requirements* (Requirement 1.6), and
c. Requirements pertaining to *Modeling features* (Requirements 2.1 and 2.2).

As design is focused on deeper levels, requirements that affect low-level task detail are assigned higher priority; these typically include:

a. *Basis in the requirements* (Requirement 1.5), and
b. *Encouragement of active user involvement* (Requirement 1.7).

Last but not least, there are requirements that are important regardless of what the scope and abstraction level are; these typically include:

a. *Practicability/Practicality* (Requirement 1.8), and
b. *Manageability of complexity* (Requirement 1.9).

Prioritization of requirements is mainly performed as a complexity management measure, since having to focus on a large repertoire of requirements can result in inadequate attention to satisfying the important ones. However, it also gives a degree of flexibility to the design process,

enabling the designer to assign higher priorities to those requirements which he/she considers essential.

2. *Design engine*: The following tasks are performed in each iteration:
    a. Selection of the design approaches to be used in the current iteration: While all the four approaches listed in the previous section can be used regardless of the scope and the level of abstraction of the design activity, they have different uses depending on the scope and abstraction level of the design activity undertaken in the current iteration: *Instantiation* is more useful when designing high-level aspects of the methodology, *Integration* and *Composition* are more suited to the design needs of low-level aspects, and the *Artefact-oriented* approach comes in between, i.e. while less useful at the general lifecycle level, it is indispensable when addressing seamlessness issues at the inter-subprocess and intra-subprocess levels. Figure 40 provides an idea of the relative emphasis typically put on the four design approaches, depicting how emphasis can be expected to change as focus shifts from high-level to low-level design. Furthermore, although the approaches are not totally disjoint, they require focus on different aspects of the methodology, and rely on distinct sets of tools and techniques: *Instantiation* relies on metamodels, *Integration* is dependent on existing methodologies (with special attention to analysis results, and methodology descriptions), *Composition* requires libraries of reusable process components, and the *Artefact-oriented* approach needs concentration on designing artefact chains using modeling languages. Using all approaches in the same iteration is not impossible, yet can cause unwarranted complexity. Therefore, in order to keep design activities duly focused, the first task in each iteration is to decide which design approaches are most suitable to the needs of the current iteration. The design tasks in the iteration can then be commenced according to the process dictated by the design approaches selected.
    b. Application of the selected design approaches aimed at defining the methodology at the current scope and level of abstraction:

Special attention should be given to the analysis results and methodology descriptions, thus implementing features of strength and avoiding common pitfalls. The prioritized set of requirements focuses the design effort on satisfying requirements of importance. The methodology elements designed are then integrated into the methodology blueprint.

c. Revision, refinement and restructuring of the methodology built so far in order to accommodate the changes made in the current iteration.
d. Specification of the level of abstraction for the next iteration, and definition of the scope and intended level of detail.
e. Revision and refinement of the requirements, including their prioritization according to the scope and level of abstraction intended for the next iteration.

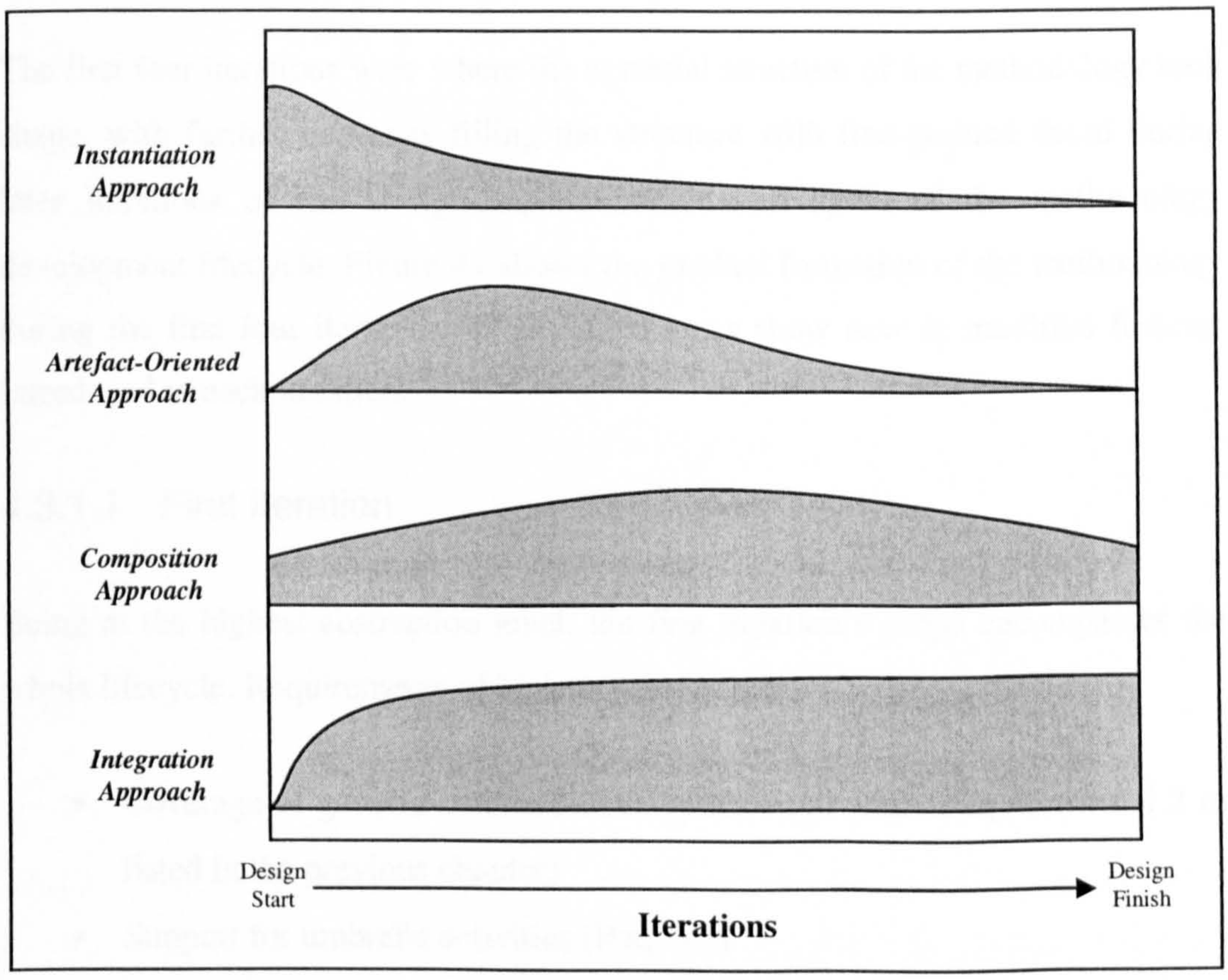

Figure 40. Emphasis put on different design approaches during iterations of the hybrid design process

Transition to the implementation phase occurs when the design process has produced a detailed enough blueprint of the methodology. Since the Design,

Implementation and Test phases of the methodology development process are carried out iteratively, results of design iterations later undergo implementation and test; the design is thus perfected gradually as implementation and test activities resolve ambiguities and mitigate risks.

4.3 Design Results

The design process described in the previous section was applied using the requirements, analysis results and methodology descriptions introduced in the previous chapter. The first four iterations of the process are briefly described in this section along with the resulting methodology design. Iterations are described in order to demonstrate the inner workings of the design process, and thereby clarify the rationale behind the design decisions.

4.3.1 Iterations

The first four iterations were where the essential structure of the methodology took shape, with further iterations filling the structure with fine-grained detail during later iterations of the Design-Implementation-Test cycle of the methodology development lifecycle. Figure 41 shows the gradual formation of the methodology during the first four iterations. Highlighted areas show new or modified features introduced in each iteration.

4.3.1.1 First Iteration

Being at the highest abstraction level, the first iteration's scope encompasses the whole lifecycle. Requirements of highest priority are:

- Coverage of generic software development activities (Requirement 1.2 as listed in the previous chapter)
- Support for umbrella activities (Req. 1.3)
- Practicability and practicality (Req. 1.8)
- Manageability of complexity (Req. 1.9)
- Extensibility/Configurability/Flexibility/Scalability (Req. 1.10)
- Application scope (Req. 1.11)

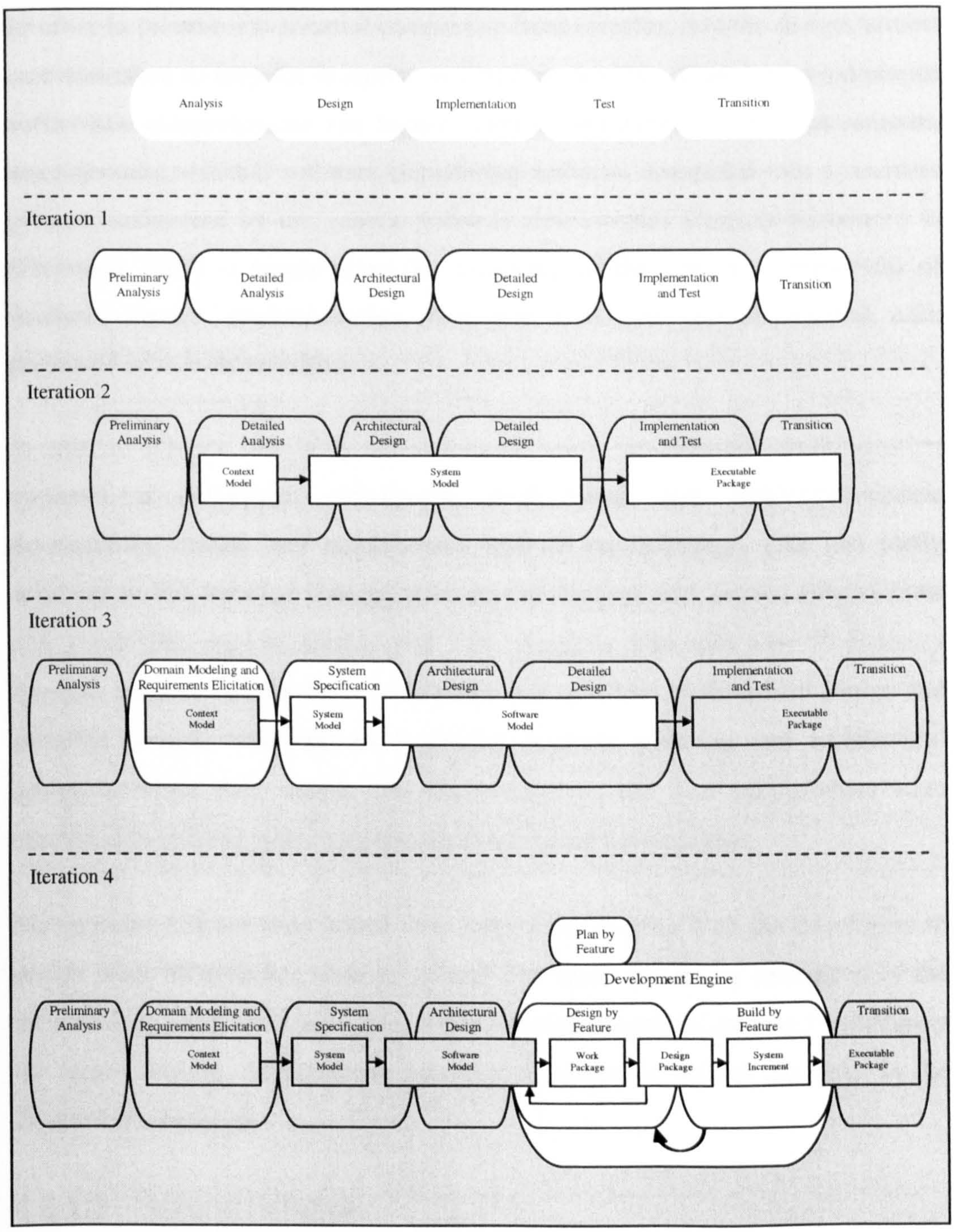

Figure 41. Gradual refinement of the methodology blueprint during the first four iterations of the design process

The design approach in this iteration was mainly *Instantiation*, using metamodels – including SPEM [OMG 2002] and OPF [Firesmith and Henderson-Sellers 2001] – and general object-oriented development lifecycles such as OOSP [Ambler 1998a]. Also used was the *Composition* approach, which was utilized for populating subprocesses with basic activities; OPF's process components (especially *stages*) [Firesmith and Henderson-Sellers 2001] and Ambler's process patterns [Ambler 1998a] were the main components used.

In order to prevent unwarranted complexity from creeping into the design, utmost care was taken to keep the blueprint as simple as possible. Therefore, a simple yet sufficiently comprehensive and highly cohesive methodology *core* was targeted, encompassing essential software engineering activities assembled into a seamless process. Influenced by the generic software development lifecycle introduced in [Pressman 2004], a simple lifecycle consisting of the generic subprocesses of *Analysis*, *Design*, *Implementation*, *Test*, and *Transition* was constructed, each populated with basic activities.

In order to enhance scalability and risk-management (as applicable to information systems), a risk-based, plan-driven, model-based, and architecture-centric development attitude was implemented into the methodology. This was partly achieved in this iteration through splitting the *Analysis* and *Design* subprocesses and combining *Implementation* and *Test*. *Analysis* was split into *Preliminary Analysis* and *Detailed Analysis*, and *Design* was split into *Architectural Design* and *Detailed Design*, with relevant feasibility analysis, planning and architectural design activities duly added. The *Implementation* and *Test* subprocesses were combined in order to better accommodate test-based development.

Maintenance has not been added as a subprocess, leaving it to the developers to decide what maintenance strategy to use. For small systems, a reiteration of the methodology lifecycle is advisable, whereas a separate procedure may be necessary for larger systems. Maintenance *planning*, however, is one of the tasks in the *Transition* subprocess.

4.3.1.2 Second Iteration

The second iteration focuses on adding modeling features to the relevant subprocesses of the methodology. Requirements of highest priority are:

- Seamlessness and smoothness of transition between phases, stages and activities (Req. 1.4)
- Testability and tangibility of artefacts, and traceability to requirements (Req. 1.6)
- Practicability and practicality (Req. 1.8)
- Manageability of complexity (Req. 1.9)

- Extensibility/Configurability/Flexibility/Scalability (Req. 1.10)
- Support for consistent, accurate and unambiguous object-oriented modeling (Req. 2.1)

The design approach in this iteration was mainly an *Integration* approach, using modeling features from existing methodologies and implementing them into the process framework designed in the previous iteration. Catalysis [D'Souza and Wills 1998] and OPEN [Graham et al. 1997] had the most influence on the design produced in this iteration.

A UML-based modeling approach similar to that used in the Catalysis methodology [D'Souza and Wills 1998] was chosen because of its fractal modeling approach, which is relatively seamless and highly tangible to developers and end-users (the modeling approach has been described in the section on Catalysis in Chapter 3). The first in the chain of artefacts thus devised is the *Context Model*, which captures the structural, functional and behavioural aspects of the problem domain (with the system as an element therein), and is built during the *Detailed Analysis* subprocess. System requirements are captured in use cases. Focus is then shifted to the internals of the system, and an initial version of the *System Model* is built during the *Detailed Analysis* subprocess, based on the information captured in the Context Model. The System Model is perfected with architectural and detailed design particulars (including the user interface and the database) during the design subprocesses, and ultimately used as a basis for implementing the system and producing the *Executable Package*.

The artefact chain introduced was perfected and elaborated during this iteration. Constituent diagrams were determined and dependencies were identified, and corresponding modeling activities were added to the relevant subprocesses.

4.3.1.3 Third Iteration

The third iteration focuses on refining and perfecting the model chain, especially targeting seamlessness and smoothness of transition. Requirements of highest priority are:

- Seamlessness and smoothness of transition between phases, stages and activities (Req. 1.4)
- Testability and tangibility of artefacts, and traceability to requirements (Req. 1.6)
- Practicability and practicality (Req. 1.8)
- Manageability of complexity (Req. 1.9)
- Extensibility/Configurability/Flexibility/Scalability (Req. 1.10)
- Application scope (Req. 1.11)
- Support for consistent, accurate and unambiguous object-oriented modeling (Req. 2.1)
- Provision of strategies and techniques for tackling model inconsistency and managing model complexity (Req. 2.2)

The design approach in this iteration was mainly an *Artefact-Oriented* approach, focusing on improving the artefact chain, and reshaping the process around it accordingly. The design was deeply influenced by FDD [Palmer and Felsing 2002], as well as the data-flow-oriented modeling approaches seen in FOOM [Shoval and Kabeli 2001] and EUP [Ambler and Constantine 2000a].

The artefact chain introduced during the previous iteration suffers from disruptions in seamlessness due to the fact that mapping problem-domain (context) models to corresponding system models is typically not a smooth process [Isoda 2001]. Realizing the need for a smoother transition from problem domain models to system models, a novel method was devised resulting in a modified chain of artefacts, a brief description of which will be given below. Detailed description of the model chain is given in Chapter 5, with examples presented in Chapter 6.

The fractal modeling approach inspired by Catalysis [D'Souza and Wills 1998] (introduced in the second iteration) is preserved: the problem domain is modeled as consisting of objects, and the target system is added to this model as a problem domain object, which in turn consists of system objects. The *Context Model* in this chain is built through direct object-oriented real-world modeling of the problem domain, with human workers, systems and data-stores modeled as collaborating objects. A data-flow-oriented approach has been adopted for structural and functional modeling – due to the high tangibility of the models produced and their

close correspondence to the real-world problem domain – using a notation similar to that of UML collaboration diagrams (Version 1.5) [OMG 2003] for modeling problem domain objects and the data flowing among them; Figure 56 in Chapter 6 (page 314) is an example of this diagram. The diagram in fact resembles a data flow diagram, and intentionally so, as DFDs used in this context – i.e. with actors regarded as objects and modeled as DFD processes – offer a close correspondence to problem domains, and are therefore widely used as workflow diagrams (alongside UML activity diagrams) for business process modeling [Ambler and Constantine 2000a]. Job descriptions and functionalities of problem domain objects are identified and expressed as FDD-style *Features* [Palmer and Felsing 2002]. Features have been preferred to use cases in this context, since they are intrinsically object-oriented, and the set of conventions governing their definition ensures a high-level of expressiveness and provides apt complexity management mechanisms. The system is then added to the models as an object of the problem domain. Requirements of the system are identified through redistributing features among problem domain objects, which results in the assignment of features to the system object. New features are then added to the system as additional requirements, if deemed necessary.

Focus is then shifted to the internals of the system, and the *System Model* is produced from the Context Model. Objects sharing features with the system are either moved inside the system boundary or assigned system counterparts. The system is then designed as a homogeneous extension to the problem domain; this means that the same types of entities as seen in the problem domain are used for designing the system. In a business system, for instance, this means adding a new section or department consisting of staff performing predefined jobs using equipment and tools made available to them. In this extension, data-store and flowing-data objects are assigned to *Custodians*, which are proxy objects enabling and controlling access to the objects (example in Figure 68, page 327). This ensures that all passive objects are coupled with corresponding active objects. Features are assigned to objects based on the features assigned to the system. Designing the System Model as an extension to the problem domain keeps the models tangible to both domain experts and developers, and smoothes the transition to software objects.

The System Model is then transformed into the *Software Model*, which depicts the internal structure and behaviour of the *computer-based* system. This is achieved through applying specifically adapted patterns by which features are redistributed among objects and necessary architectures are introduced. The real-world domain modeling approach used for producing the Context Model is a hazardous practice, in that it can introduce actors or irrelevant objects among system objects, or allow redundant associations and interactions to be entered into the System Model [Isoda 2001]. The pattern-based transformation approach adopted, however, resolves these issues during the transformation of the System Model into the Software Model by gradually pruning the models of redundancies. Bearing in mind that the target methodology's application scope is confined to information systems, a set of patterns specifically targeting business systems (as ubiquitous examples of information systems) was compiled and adapted for this purpose. The transformation process is similar in essence to an approach advocated by MDA [OMG 2001], in which transformation patterns are proposed for transforming Platform Independent Models into Platform Specific Models; here, though, models of the system designed as an extension to the problem domain are transformed into software domain models.

The pattern-based transformation approach was inspired by observations made by the author, indicating that object-oriented *reengineering patterns* [Demeyer et al. 2003] and *refactoring patterns* [Fowler 1999] can be used for redistributing features among objects so that anomalies in objects, relationships and interactions – introduced as a result of real-world modeling – are rectified. The applicability of *design patterns* [Gamma et al. 1995, Buschmann et al. 1996] for introducing object-oriented structure and behaviour in models of business systems is based on the author's personal experience, according to which many job definition and task assignment techniques in organizational design and personnel management are noticeably similar in effect to the transformations seen in design patterns. Applying design patterns not only results in improved structures familiar to domain experts, but also facilitates the translation of these structures into solution-domain and implementation-domain class structures.

The transformation procedure results in highly cohesive objects and reduced inter-object coupling. The procedure culminates in passive objects being combined with

their custodians. The Software Model thus built represents the actual model of the computer system, to be refined and perfected during the design subprocesses and ultimately used as a basis for producing the *Executable Package*.

Introduction of the System Model as a transitional model – bridging the gap between real-world domain modeling and system modeling – has resulted in the introduction of a new *System Specification* subprocess. The *Detailed Design* subprocess defined in previous iterations is split into a *Real-World Domain Modeling and Requirements Elicitation* subprocess and the new *System Specification* subprocess, where the former is where the context model is produced, and the latter is where the System Model is constructed and transformed into the Software Model. Activities of the two subprocesses were defined in this iteration, as well as the structure of the models produced.

4.3.1.4 Fourth Iteration

The fourth iteration's scope encompasses the *Detailed Design* and *Implementation and Test* subprocesses of the methodology, and focuses on harmonizing the design and implementation activities with the feature-driven basis of the models. Requirements of highest priority are:

- Support for umbrella activities (Req. 1.3)
- Seamlessness and smoothness of transition between phases, stages and activities (Req. 1.4)
- Basis in the requirements (Req. 1.5)
- Testability and tangibility of artefacts, and traceability to requirements (Req. 1.6)
- Encouragement of active user involvement (Req. 1.7)
- Practicability and practicality (Req. 1.8)
- Manageability of complexity (Req. 1.9)
- Support for consistent, accurate and unambiguous object-oriented modeling:

The design approach in this iteration was mainly an *Integration* approach, aiming to use existing methodologies for transforming the *Detailed Design* and

Implementation and Test subprocesses into feature-driven subprocesses. FDD [Palmer and Felsing 2002] had the most influence on this iteration.

The iteration resulted in the introduction of the FDD methodology's feature-driven iterative-incremental development engine into the methodology [Palmer and Felsing 2002]. Adapting other methodologies to the purpose was also considered (especially Catalysis and Scrum), yet FDD was deemed the logical choice, since it suitably addresses the above requirements, and already incorporates a cohesive feature-driven development engine. As a result, the *Detailed Design* and *Implementation and Test* subprocesses were replaced by the cyclic *Design by Feature* and *Build by Feature* subprocesses respectively. A *Plan by Feature* subprocess was also added, during which the activities of the development engine are planned and scheduled.

In each iteration of the development engine, features are selected from the Software Model for design and implementation. The selected features comprise the *Work Package*, based on which detailed design is performed and results are duly reflected back to the Software Model. The design results, comprising the *Design Package*, are then implemented and tested, with the resulting *System Increment* ultimately integrated into the *Executable Package*. Activities of the development engine were defined in this iteration, as well as the structure of the artefacts produced.

4.3.2 The Designed Methodology

The general approach of the designed methodology is based on smooth and seamless transition from real-world domain models to system models, and ultimately to software design models. In business systems, this can be achieved via the use of patterns for iterative transformation of the models through redistribution of functionalities and introduction of object structures. The process consists of the following subprocesses:

1. Feasibility analysis and preliminary planning.
2. Real-world domain modeling and requirements elicitation
3. System specification
4. Architectural design

5. Planning by feature
6. Feature-driven iterative-incremental development
    6.1. Design by feature
    6.2. Build by feature
7. Transition

UML (Version 1.5) [OMG 2003] is the main modeling language used for diagramming in the designed methodology. UML *activity diagrams* and *sequence diagrams* are used for behavioural modeling (in Interaction Models), with UML *component diagrams* used for modeling subsystems. UML *class diagrams* are used later in the modeling process to depict software classes and their relationships (in Class Models). However, Object Models – which capture functional and structural aspects throughout the modeling stages – use a notation similar to UML *collaboration diagrams*, yet in a data-flow-oriented context analogous to Data Flow Diagrams (DFD); i.e. Object Models use the same notation that is used for denoting message passing in UML collaboration diagrams (without the sequencing), yet the notation denotes data/control flow rather than message/signal flow.

The following sections present a more detailed description of the designed subprocesses, specifically targeting business systems as an example. The finer-grained detail will be added during the implementation phase, the results of which are reported in Chapter 5. Examples of the models mentioned in these sections are given in Chapter 6.

4.3.2.1 Feasibility Analysis and Preliminary Planning

The main tasks performed in this subprocess are as follows:

1. Acquire high-level knowledge as to the nature of the project, its scope, and the risks and constraints involved.
2. Perform the traditional activities of feasibility analysis.
3. Develop rough estimates and an overall *Outline Plan* for the project.

4.3.2.2 Domain Modeling and Requirements Elicitation

The main tasks performed in this subprocess are as follows:

1. Real-world modeling of the problem domain, starting with modeling the high-level view and gradually moving inside organizational sections, focusing on lower-level elements of the problem domain. The tasks performed are as follows:

    1.1. Human workers, systems and data-stores of the problem domain are modeled as collaborating objects in a Context Object Model. A notation similar to UML collaboration diagrams is used for representing the model, with links adorned with data/control flows instead of messages (without sequence numbers); Figure 56 in Chapter 6 (page 314) is an example of this diagram. Organizational boundaries are preserved, modeled through using packages and component diagrams. The resulting functional models comprise the main bulk of the Context Model.

    1.2. Typical transaction scenarios are modeled in activity diagrams (with swimlanes depicting the participating objects) and/or sequence diagrams (example in Figure 58, page 315). The resulting Context Interaction Models comprise the behavioural part of the Context Model.

    1.3. Job descriptions and functionalities are expressed as *areas* (major feature sets), *activities* (feature-sets), and *features* [Coad et al. 1999, Palmer and Felsing 2002]. Feature lists are compiled and added to the Context Model.

    1.4. A glossary of terms from the problem domain is compiled.

2. Introduction of the system into the problem domain: The system is added as an object to the Context Model (in Context Object Models) and feature sets are assigned to the system through redistribution and/or duplication (example in Figure 57, page 314). New feature sets are added as deemed necessary by the Modeling Team, and the feature lists in the Context Model are duly updated. Typical scenarios of interaction with the system are also modeled and the Context Interaction Models are updated accordingly (example in Figure 60, page 317).
3. Non-functional requirements and constraints are identified and added to the Context Model.
4. The results, the project plan and the requirements are reviewed.

4.3.2.3 System Specification

The main tasks performed in this subprocess:

1. The Context Model built during the previous subprocess is converted to the System Model. The system is designed as an extension to the organizational structure into which the system is to be ultimately introduced, using the same types of elements already present in the problem domain. The major tasks performed are as follows:
    1.1. Human elements, systems and data stores sharing features with the system are moved inside system boundaries or assigned system counterparts. The system is then designed as a homogeneous extension of the problem domain.
    1.2. Each data-store and each flowing data object is assigned to a *custodian*; any access to any such object should be made via the custodian. It should be noted that there is no limit on the number of staff assigned to the system. The resulting System Object Models comprise the functional component of the System Model (example in Figure 68, page 327).
    1.3. Typical interaction scenarios are identified, and relevant behavioural models (typically activity diagrams and interaction diagrams) are produced for each of the system's feature sets. The resulting System Interaction Models comprise the behavioural component of the System Model (example in Figure 69, page 328).
    1.4. Feature sets and features are assigned to the active elements (staff) based on the functionality assigned to the system as a whole and the interaction models produced in the previous task.
    1.5. Review and revision of the requirements of the system and the resulting System Model is performed.
2. The System Model produced so far is converted to the Software Model by applying patterns to redistribute functionality among system objects. The tasks performed, explained as relevant to business systems, are as follows:
    2.1. Patterns are applied to the System Model to iteratively redistribute features among objects (i.e. processing staff and

custodians) in order to enhance encapsulation, increase cohesion and reduce coupling, and also to introduce architecture. Reengineering patterns, especially those suggested in [Demeyer et al. 2003] for redistributing responsibilities among objects, are of utmost use in the starting iterations. These typically include:

- Moving behaviour close to data
- Eliminating navigation
- Splitting up God classes (Blobs)

Refactoring patterns proposed in [Fowler 1999] can also be used in conjunction with the above. Design patterns [Gamma et al. 1995, Buschmann et al. 1996] can be used in later iterations to help implement specific architectures and mechanisms typically present in the problem domain and tangible to users. Antipatterns can also be of use in the redistribution procedure [Brown et al. 1998]. The redistribution procedure is devised in such a way as to resolve the problems typically afflicting analysis approaches based on object-oriented real-world modeling [Isoda 2001]. Objects irrelevant to the system and actor-counterparts without any justification for existence in the system are gradually disposed of, and relationships not belonging to the system are not introduced into the models because of the interaction-oriented and feature-driven nature of the System Model and the redistribution procedure (example in Figure 71, page 331). Behavioural models are updated in each iteration of the redistribution procedure.

2.2. Applying the patterns ultimately results in custodian objects being merged with the data objects they had under custody. This marks the transition from the problem-domain-based system to the computer system, signifying the transition to solution domain. The resulting Software Object Models comprise the functional component of The Software Model (example in Figure 75, page 335). UML class diagrams are then produced based on the Object Models, depicting the classes in

the system and their relationships. Inheritance hierarchies are introduced in order to enhance abstraction (patterns for refactoring inheritance can be of use in this context [Fowler 1999]). The Software Class Models thus produced comprise the main structural component of the Software Model.

2.3. Behavioural diagrams inherited from the System Model are updated according to the new Software Class/Object Models. The resulting Software Interaction Models comprise the behavioural component of the Software Model. Message passing should be clearly depicted.

2.4. Preparation of initial versions of class and method prologues

2.5. Review and revision of the requirements

2.6. Review of the resulting Software Model

3. Review of the results of the subprocess, the project plan and the requirements

4.3.2.4 Architectural Design

The tasks performed in this subprocess (mostly in parallel) are as follows:

1. Convey the Software Model to the implementation domain through adding implementation-specific detail and restructuring it in order to facilitate implementation and accommodate the domain-independent parts of the system. The user interface is designed, and the Software Model is enriched with architectural design patterns.
2. Identify the architecture of the domain-independent parts of the system: hardware and software platforms, infrastructure components (such as middleware and databases), utilities for logging/exception-handling/start-up/shutdown, design standards and tools, and the choice of component architecture (such as JavaBeans or COM), are all added to the Software Model.
3. Review the results, the project plan and the requirements.

4.3.2.5 Planning by Feature

The main tasks performed in this subprocess are as follows:

1. Determine the development sequence by scheduling the development of the feature sets (*activities*), thereby producing a Development Plan.
2. Assign feature sets to development coordinators.
3. Assign classes to developers.
4. Review the resulting development plan, the project plan and the requirements.

4.3.2.6 Feature-Driven Iterative-Incremental Development

Almost identical to the iterative-incremental engine in the FDD methodology, the iterative subprocesses is where strands of design-build iterations start off as each development coordinator (called *Chief Programmer*) selects the set of features (called the *Work Package*) that should be developed in each of the iterations performed under his supervision, and forms a Features Team to do the job in the timeframe set in the development plan.

Design by Feature

The tasks performed in this subprocess are as follows:

1. Study the Software Model in order to obtain a better understanding of the particulars of the features.
2. Refine and complete the sequence diagrams in the Software Interactions Models, which as the behavioural component of the Software Model, are required to show how software objects should interact at run-time in order to implement each of the features.
3. Refine the Software Object Models (class diagrams) so that they support the sequence diagrams produced in the previous task.
4. Write Class- and Method-prologues for the elements of the Software Object Models.
5. Inspect the design for errors, inconsistencies and areas for improvement.
6. Review and revise the Work Package (the features and the iteration schedule).

The products of this subprocess are transferred to the next subprocess as a *Design Package* consisting of the sequence diagrams produced, the refinements made to

the Software Model, the prologues, and the notes on the design alternatives explored, constraints, and assumptions.

Build by Feature

The tasks performed in this subprocess are as follows:

1. Implement classes and methods according to the specifications given in the Design Package.
2. Conduct a code inspection.
3. Unit-test the code to ensure that all classes satisfy the functionality required.
4. Integrate the increment with the system built so far, if the implemented classes are successfully inspected and unit-tested.
5. Review the results, the development plan, the project plan and the requirements.

4.3.2.7 Transition

The main tasks performed in this subprocess are as follows:

1. Test and validate the complete system.
2. Integrate the system with existing systems.
3. Convert legacy databases and systems to support the new release.
4. Train the users of the new system.
5. Deploy the new system.

4.4 Requirements-Based Review of the Design

Before proceeding to implementation, it is important to review the methodology design according to the requirements defined in the previous chapter, and to modify the requirements if necessary. Due to the risk-based approach of the development effort, the methodology design is volatile at the start of transition to the implementation phase, and is bound to be refined and perfected during the iterative application of the design-implementation-test cycle introduced in Chapter 1, yet regular requirements-based reviews of the methodology are essential for

ensuring quality and maintaining the focus of the effort. Table 1 shows how each requirement has been addressed in the final methodology design, thereby identifying the requirements that remain to be addressed, and requirements that need modification. It also shows how the design has been influenced by existing methodologies and process patterns/metamodels in addressing the requirements. In this regard, Table 1 is complemented by Table 2 of Chapter 5 (page 306), which tabulates the results of a requirements-based review of the *implemented* methodology.

4.5 Summary

An iterative design process has been devised for performing methodology design. The process produces a blueprint of the target methodology through flexible and adaptive application of a set of four design approaches: *instantiation* of process metamodels, *artefact-oriented* definition of the process (around an artefact-chain), *composition* of process patterns, and *integration* of features from existing methodologies.

The design process is dependent on the results of the analysis phase; i.e. the process-centred descriptions, the criteria-based analysis results, and the requirements. In order to focus the effort on satisfying requirements of highest relevance, requirements are prioritized at the start of each design iteration, based on the abstraction level and scope of the design activity planned to be undertaken in the iteration. The design activities performed in each iteration make use of the results of the process-centred review and analysis of object-oriented methodologies, process patterns and process metamodels carried out in the analysis phase, not the least as a source of ideas as to what features are desirable in a methodology and what pitfalls should be avoided. Furthermore, the review and analysis results are also the basis of the *Integration* design approach – one of the four design approaches available for use during the iterative design process – in which features, ideas and techniques from existing methodologies are integrated to form the design.

Transition to the implementation phase occurs as soon as the blueprint is deemed complete enough; i.e. to a degree that ensures the relative stability of the design so

that implementation can be started. It should be reiterated that each application of the design process is in turn part of an iteration of the Design-Implementation-Test cycle of the methodology development lifecycle (as explained in Chapter 1). The decision to start implementation is therefore dependent on the level of detail targeted in each iteration of the cycle; however, a high level of risk introduced during design may prompt an early transition to implementation and test.

Table 1. Satisfaction of methodology requirements in the design phase (continued on next page)

REQUIREMENT			ADDRESSED/ NOT ADDRESSED	DETAILS	FOLLOW-UP ACTION
Process	Clarity, rationality, accuracy, consistency of definition		Not Addressed	N/A	➢ To be addressed in the implementation phase.
	Coverage of generic development lifecycle activities		Addressed (except Maintenance)	➢ Addressed at the lifecycle, subprocess and task levels (influenced by SPEM, OPF, and Ambler's OOSP). ➢ Maintenance has not been added in order to avoid commitment to a specific maintenance strategy.	➢ This requirement needs to be amended in order to accommodate the lack of coverage for maintenance.
	Support for umbrella activities	Risk management	Addressed	➢ Addressed through iterative-incremental development (mainly influenced by FDD), preliminary analysis (influenced by OPF and DSDM), risk-based planning (influenced by DSDM and FDD), prototyping (inspired by RUP and XP), continuous verification and validation (influenced by ASD), regular product/plan reviews (influenced by ASD and Crystal), and continuous integration (influenced by FDD).	➢ Supporting features to be further elaborated in the implementation phase.
		Project management	Addressed	➢ Addressed through project planning, scheduling and control activities incorporated in the subprocesses of the methodology, and provisions for review and revision of the plans throughout the process (influenced by ASD, Scrum and FDD).	➢ To be further elaborated - and complemented with resource management, and team management techniques in particular - during the implementation phase.
		Quality assurance	Addressed	➢ Addressed through regular technical reviews (influenced by ASD and Crystal), continuous verification and validation during iterative development (influenced by ASD), and requirements traceability (influenced by Catalysis and FDD).	➢ Supporting features to be further elaborated in the implementation phase.
	Seamlessness and smoothness of transition between phases, stages and activities		Addressed	➢ Addressed through the artefact chain – governing the process through analysis and design (inspired by Catalysis) – and the iterative-incremental development engine (inspired by FDD).	➢ Supporting features to be further elaborated in the implementation phase.
	Basis in the requirements		Addressed	➢ Addressed through the feature-driven approach governing all development activities throughout the process (influenced by FDD).	➢ Supporting features to be further elaborated in the implementation phase.
	Testability and Tangibility of artefacts, and traceability to requirements		Addressed	➢ Addressed via basis in real-world modeling (influenced by Fusion and EUP), fractal modeling (influenced by Catalysis), gradual seamless transformation of artefacts through analysis and design (influenced by Catalysis and BON), and the feature-driven nature of artefacts throughout the process (inspired by FDD).	➢ Supporting features to be further elaborated in the implementation phase.
	Encouragement of active user involvement		Addressed	➢ Addressed through constant participation of user representatives throughout the process (influenced by FDD).	➢ Supporting features to be further elaborated in the implementation phase.

Table 1. Contd.

REQUIREMENT			ADDRESSED/ NOT ADDRESSED	DETAILS	FOLLOW-UP ACTION
Process	Practicability and practicality		Addressed	➢ Addressed through avoiding complexity at all levels (influenced by FDD), adhering to risk-based development (influenced by DSDM and FDD), incorporating project management activities (influenced by ASD, Scrum and FDD), and using techniques and strategies for focusing the development (e.g. feature-driven model chain and architectural design; influenced by FDD and RUP respectively).	➢ Supporting features to be further elaborated in the implementation phase.
	Manageability of complexity		Addressed	➢ Addressed through the hierarchical structure of the methodology, and keeping subprocesses, activities and tasks cohesive and easy to understand (influenced by Catalysis and FDD).	➢ Supporting features to be further elaborated in the implementation phase.
	Extensibility / Configurability / Flexibility / Scalability		Not Addressed (Except Scalability)	➢ Scalability was addressed through plan-based, model-driven and architecture-centric basis of the process (influenced by Catalysis, RUP and FDD).	➢ Extensibility, Configurability, and Flexibility to be addressed in the implementation phase.
	Application scope (*Information Systems*)		Addressed	➢ Partially addressed through concentrating on business systems as commonly encountered information systems (influenced by Catalysis). ➢ Applicability to other kinds of information systems has not been explored.	➢ Applicability to information systems other than business systems to be explored in the implementation phase.
Modeling Language	Support for object-oriented modeling	Structural – Functional – Behavioural	Addressed	➢ Addressed through using appropriate UML-based diagrams at different levels (influenced by Catalysis). Functional modeling has been addressed through the use of a notation similar to that of UML collaboration diagrams, yet in a data-flow oriented context (inspired by DFDs and their use in EUP and FOOM).	➢ Supporting features to be further elaborated in the implementation phase.
		Logical to Physical	Addressed	➢ Addressed through the model chain, starting at the problem-domain level and proceeding to detailed design (influenced by Catalysis).	➢ Supporting features to be further elaborated in the implementation phase.
		At different levels of granularity	Addressed	➢ Addressed through fractal modeling (influenced by Catalysis) at different granularity levels (Enterprise level – System level – Subsystem/Package level – Inter-object level – Intra-object level).	➢ Supporting features to be further elaborated in the implementation phase.
		Formal and Informal features	Not Addressed (Formal)	➢ Informal features implemented through UML.	➢ Formal modeling features to be considered during the implementation phase.
	Provision of strategies and techniques for tackling inconsistency and complexity		Not Addressed	N/A	➢ To be addressed during the implementation phase.

Chapter 5

Implementation

The Implementation phase is concerned with detailing, extending and refining the methodology blueprint produced during the Design phase. The objective is to convert the blueprint (described in Chapter 4) into a detailed methodology specification that is directly usable by system developers. This chapter explains the implementation process and presents the end result, i.e. the implemented methodology.

5.1 Implementation Process

In any development effort, implementation means building the designed product in a form usable by the intended end users. In a methodology development effort such as this, implementation deals with adding pragmatic fine-grained detail to the methodology's design and representing it in a form usable to the intended audience, i.e. software engineers. A User Guide is the normal medium for representing a software development methodology, but there is no single standard format for a methodology user guide. Therefore, devising a suitable template for the user guide is the first task in the implementation of the designed methodology. The User Guide structure thus defined is not only used for representing the final methodology, but also guides the perfection and refinement of the methodology through focusing the methodology development effort on issues which are expected to be addressed in a typical methodology user guide.

According to the requirements described in Chapter 3, the definition of the methodology should provide concise yet comprehensive, clear, rational, accurate, detailed and consistent description of the methodology lifecycle and work-units, producers (roles), modeling language, work-products, techniques and rules, and issues pertaining to umbrella activities. In order to satisfy this set of requirements,

a multiple-view and top-down approach has been chosen for describing the methodology: three views of the methodology are provided, while focus is gradually shifted from high-level specifications to fine-grained detail. The three views through which the methodology is represented are as listed below:

1. *Process-Centred*: focusing on the lifecycle and the work-units performed in the methodology (subprocesses, activities and tasks), describing all other elements of the methodology - roles performing the process, work-products produced during the process and the modeling languages used for expressing them, techniques and rules, and issues pertaining to umbrella activities - in the context of the process.
2. *Work-Product-Centred*: focusing on the work-products, the modeling languages in which they are expressed, their interdependencies and their trend of evolution in the course of the methodology. All other elements of the methodology are described as secondary to the work-products.
3. *Role-Centred*: focusing on the people (producers) involved in the methodology and the relevant management issues. All other elements of the methodology are described as pertinent to the roles.

The above is based on the notion supported by prominent methodology metamodels – especially the OPEN Process Framework (OPF) [Firesmith and Henderson-Sellers 2001] and the Software Process Engineering Metamodel (SPEM) [OMG 2002] – that a software development methodology consists of three types of basic components: *work-units* (organized in *stages*, and ultimately a *lifecycle*), *work-products* (described using *modeling languages*), and *producers (roles)*. This general metamodel has been used for the instantiation and composition of software development methodologies [Firesmith and Henderson-Sellers 2001]. Furthermore, some form of this multi-view approach (although rather unstructured and informal) can be seen in the user guides of a number of modern methodologies; RUP [Kruchten 2003], USDP [Jacobson et al. 1999], and FDD [Palmer and Felsing 2002] are prominent examples. Having three views of the methodology not only highlights the issues pertinent to each of these three types of components, it also makes it possible to de-clutter the description of the methodology through keeping fine-grained detail where it is most relevant. For example, low-level development task details are confined to the process-centred

view, modeling language and diagramming issues are mainly addressed in the work-product-centred view, and team-related issues are solely attended to in the role-centred view.

The template used for describing the user guide has thus taken shape as shown in Figure 42. The template excludes examples at this stage, as these are produced during testing and can be later added as complements to the user guide.

User Guide Template

- Overall Process
  - Lifecycle: subprocesses and their order of execution
  - Work-products
  - Roles and teams
- Process-centred description of the methodology; consisting of phase descriptions, each of which covers the following:
  - Entry criteria
  - Tasks and their order of execution; the following are explained for each task:
    - Work to be performed
    - Work-products to be produced
    - Roles involved
  - Verification
  - Exit criteria
- Work-product-centred description of the methodology; consisting of work-product descriptions, each of which covers the following:
  - Structure and modeling language
  - Dependencies
  - Trend of evolution throughout the process: Subprocesses and tasks where the work-product is created or modified
  - Producers: Roles involved in producing/modifying the work-product throughout the process
  - Production methods and guidelines
  - Consistency issues
- Role-centred description of the methodology
  - Roles: Responsibilities throughout the process
  - Teams: Constitution and responsibilities

Figure 42. User guide template

5.2 End Result: Methodology User Guide

The following sections contain the detailed description of the methodology as presented in the framework of the proposed user-guide template. A high-level view

of the methodology is presented before delving into finer-grained detail. Examples of models (produced during the Test phase) are presented in Chapter 6, yet have been referenced in the relevant sections of this chapter.

5.2.1 Overall Process

A general view of the methodology lifecycle, the work-products produced, and the roles and teams involved in performing lifecycle activities is herein presented.

5.2.1.1 Lifecycle: Subprocesses and Their Order of Execution

The proposed process is based on smooth and seamless transition from real-world domain models to system models, and ultimately to software design models. In business systems, this can be achieved via use of patterns for iteratively transforming the models through redistributing functionalities and structures. The process consists of the following subprocesses:

1. Preliminary Analysis (feasibility analysis and preliminary planning): with the focus on preliminary feasibility study of the project, weighing the available resources against constraints and complexities involved. An overall plan is also produced for the development effort.
2. Real-world domain modeling and requirements elicitation: with the focus on modeling the problem domain into which the system is to be introduced. The system is then inserted into this *context* model, and its requirements are specified as FDD-style feature sets [Coad et al. 1999].
3. System specification: with the focus on iterative translation of the problem domain model first into a *system* model – in which the system is designed as a non-automated subunit of the problem domain – and ultimately to a *software* model, depicting the internal structure and behaviour of the computer-based system. In the case of business systems, adapted versions of reengineering-, refactoring- and design patterns are iteratively applied to the *system* model in order to produce the target *software* model.
4. Architectural design: with the focus on identifying an implementation-specific architecture for the system modeled so far, and determining the domain-independent infrastructure supporting the system.

5. Planning by feature: with the focus on scheduling the features for development, and then assigning the feature sets (*activities*), and the classes in the system model, to developers.
6. Feature-driven iterative-incremental development: During this iterative development phase, Each feature-set-developer (called *Chief Programmer*) selects the set of features (called the *Work Package*) that should be developed in each of the iterations performed under his supervision, and develops the feature sets by commissioning class-developers (called *Class Owners*) to cooperate in order to design and implement the features. The constituent subphases, in the order that they are performed in each iteration, are as follows:
    6.1. Design by feature: with the focus on determining how the features in the work package should be realized at run-time by interactions among objects.
    6.2. Build by feature: with the focus on coding and unit-testing the necessary items for realization of the features in the work package. The implemented items that pass the tests are then integrated into the main build.
7. Transition: with the focus on validating the system and releasing it into the user environment.

Figure 43 shows the lifecycle and its subprocesses.

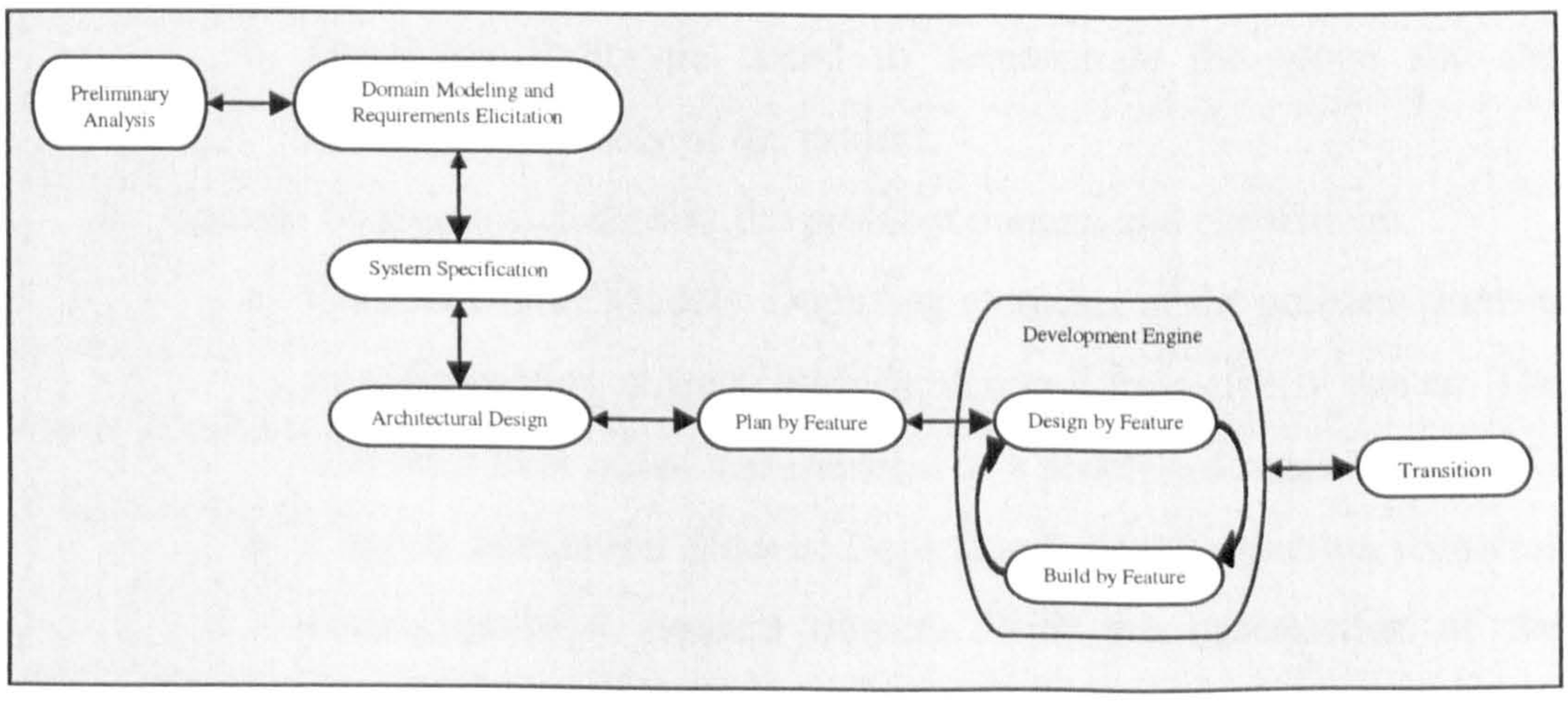

Figure 43. The lifecycle of the methodology

5.2.1.2 Work Products

The methodology is based on expressing functionality and functional requirements as FDD-style *Feature Sets* and *Features* [Coad et al. 1999, Palmer and Felsing 2002]. Features are relatively fine-grained user-valued pieces of functionality expressed in client-valued terms, conforming to the general template: <*action*> <*result*> <*object*>; for example, "*calculate* the *total value* of a *shipment*" or "*check* the *availability of seats* on a *flight*". Each feature is identified as a *Step* in one or more *Activities* (also called *Feature Sets*), which are expressed as conforming to the general template: <*action*><*-ing*> *a(n)* <*object*>; for example, *"reserving a seat"*. *Activities* in turn belong to *Areas* (or *Major Feature Sets*), which are expressed using the general template: <*object*> *management*; for example, *"reservations management"*. This three-layered structure allows the developers to adequately manage the complexity of functionalities and requirements.

The following work products are produced in the methodology:

1. Feasibility Analysis Package, which encapsulates the results of preliminary analysis and consists of:
    a. Feasibility Report: The Feasibility Report includes information on the scope of the system, high-level requirements, constraints and risks involved, the resources required, and alternative approaches to developing the system and results of their analysis.
    b. Feasibility Prototype: Used to demonstrate the scope and the technical feasibility of the project.
2. Context Model, which depicts the problem domain and consists of:
    a. Context Object Models: Depicting elements of the problem domain as collaborating objects, with data/control flow clearly shown. The system is later added and modeled as a problem domain object.
    b. Context Interaction Models: Depicting typical transaction scenarios among problem domain objects. With the introduction of the system, models are produced depicting the typical system usage scenarios.

c. Feature Lists: Job descriptions and functionalities are expressed as *areas* (major feature sets), *activities* (feature-sets), and *feature.* With the introduction of the system, system features are identified and set as functional requirements.
d. Context Vocabulary: A glossary of terms from the problem domain
e. Non-functional requirements and constraints

3. System Model: The result of extending and refining the Context Model, the System Model shows the internal constitution of the system designed as an extension to the problem domain, using the same notions and concepts as those found in the problem domain. The System Model consists of:
    a. System Object Models: Depicting intra-system elements as collaborating objects, using the same element types as those present in the problem domain.
    b. System Interaction Models: Showing typical interaction scenarios among system elements.
    c. Features list: Composed of features and feature sets assigned to intra-system objects and subsystems.
    d. Revised list of non-functional requirements and constraints
4. Software Model: The Software Model depicts the constituent elements of the software system, and is the result of applying feature redistribution patterns to the System Model. It consists of:
    a. Software Object/Class Models: Object Models depict typical links and data flows among system objects, and are complemented by Class Models, showing the classes of the system and their relationships. Architectural information and domain-independent elements are added to these models in later subprocesses of the development lifecycle.
    b. Software Interaction Models: Depicting typical object interactions in the software system.
    c. Class and Method prologues
    d. Revised list of features
    e. Revised list of non-functional requirements and constraints
5. Project Plan and Development Plan

6. Work Packages: A Work Package is the set of features that a Chief Programmer has chosen to be designed and built in each iteration of the iterative-incremental development engine.
7. Design Packages: A Design Package encapsulates the results of the Design stage in each of the iterations of the iterative-incremental development engine, and is used as a basis for implementing the features in the iteration's Build-by-Feature stage. It consists of:
    a. Refinements made to the Software Model during detailed design in order to facilitate the implementation of the features specified in the iteration's Work Package.
    b. Class and method prologues detailing the structure and particulars of classes and methods.
    c. Notes on the design alternatives explored, the constraints specified, and the assumptions made during design.
8. Verification and validation reports
9. Executable Package, consisting of executables and run-time components
10. User Guides and Operation Manuals

The models produced: the Context Model, the System Model and the Software Model, are in fact different evolution stages of one, single model. The Context Model is extended and refined into the System Model, and the System Model is in turn converted into the Software Model using pattern-based mapping. The Software Model is later perfected and enriched with architectural and detailed design specifications, after which it is used as a basis for implementation.

UML (Version 1.5) [OMG 2003] is the main modeling language used for diagramming in the methodology. UML *class diagrams* (used for producing Class Models), *activity diagrams* (used for producing Interaction Models), *sequence diagrams* (used for producing Interaction Models), and *component diagrams* (used for modeling subsystem architectures in Class Models and Object Models) are all produced according to UML specifications. However, Object Models – which capture functional and structural aspects of the context, system and software – use a notation similar to UML *collaboration diagrams*, but in a data-flow-oriented context analogous to *Data Flow Diagrams (DFD)*, i.e. through replacing message/signal flow by data/control flow, and ignoring sequencing.

Figure 44 shows the work products and their interdependencies.

5.2.1.3 Roles and Teams

The roles involved in the development methodology, many of which are extended versions of FDD roles [Palmer and Felsing 2002], are as listed below:

1. Project Manager: Overall manager of the development effort
2. Client Representative: Makes decisions on behalf of the client
3. Domain Expert: Provides knowledge on the problem domain
4. Ambassador User: Supplies user feedback
5. Chief Architect: Acts as modeling coordinator
6. Modeling Expert: Provides guidance on object-oriented modeling
7. Patterns Advisor: Provides assistance on using design, reengineering and refactoring patterns for producing the Software Model
8. Development Manager: Coordinates development teams during iterative-incremental development subphases
9. Chief Programmer: Directs detailed design and implementation activities during iterative-incremental development
10. Class Owner: Performs detailed design, implementation and test on classes put under his ownership

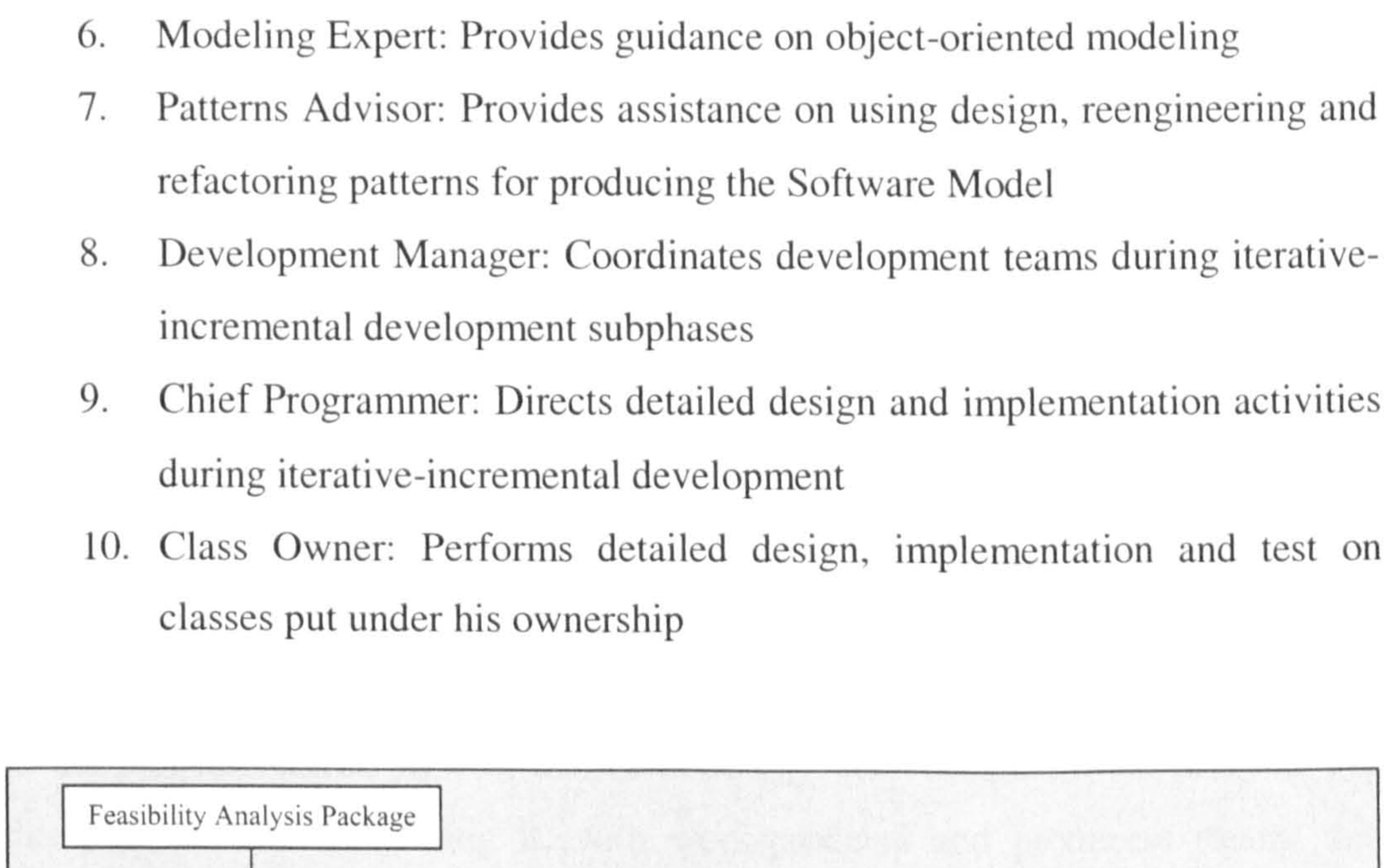

Figure 44. Work products and their interdependencies

The teams undertaking the execution of subprocesses and activities in the course of the methodology are as follows:

1. Preliminary Analysis Team: Performing feasibility analysis and preliminary planning during the Preliminary Analysis subprocess.
2. Modeling Team: Performing real-world problem domain modeling and requirements elicitation during systems analysis.
3. Model Conversion Team: Producing the *system* and *software* models during the System Specification subprocess.
4. Architectural Design Team: Producing a blueprint for the architecture of the system, based on which detailed design and implementation will be performed.
5. Planning Team: Producing a plan for the iterative-incremental development phase.
6. Features Team: In charge of the iterative-incremental design stages of the methodology, performing detailed design and implementation in pre-planned iterations.
7. Transition Team: In charge of releasing the system into the user environment.

5.2.2 Process-Centred Description of the Methodology

In the process-centred view of the methodology, the focus is on the lifecycle and the subprocesses comprising it, with work-products and producers (teams and roles) viewed in the context of – and secondary to – the process. Figure 45 shows a high-level process-centred view of the methodology. The following sections describe the details in each of the subprocesses of the methodology.

5.2.2.1 Preliminary Analysis (Feasibility Analysis and Preliminary Planning)

The Preliminary Analysis subprocess is mainly concerned with acquiring high-level information about the system in order to assess the feasibility of the development effort and set the groundwork for commencement of the project. Figure 46 shows the tasks involved in this subprocess and the work-products produced.

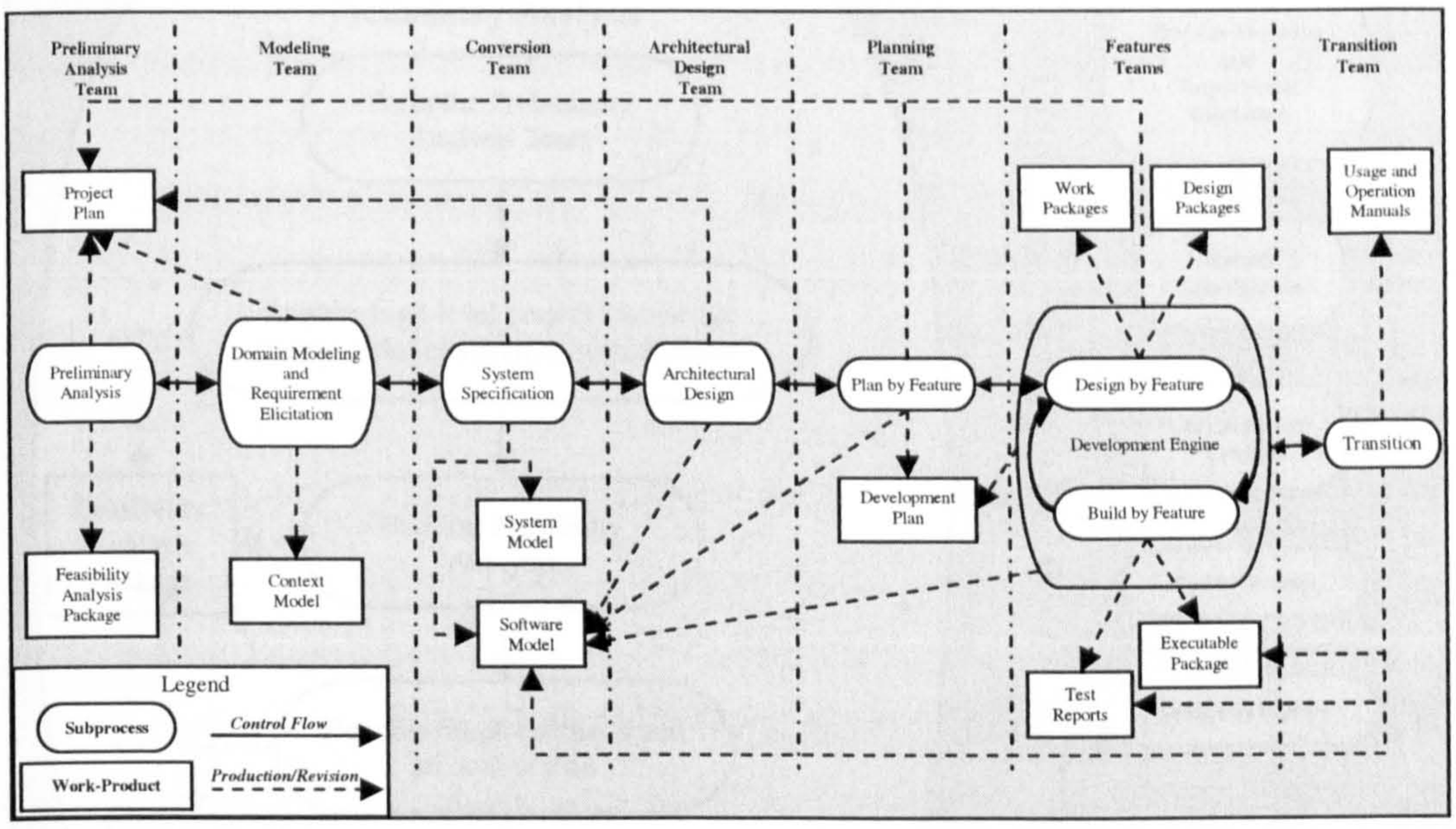

Figure 45. Process-centred view of the methodology: lifecycle, teams responsible for carrying out the subprocesses, and work-products produced or revised

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Request made by the client, and case established for considering the feasibility of the effort
- Resources available for performing the preliminary analysis

Tasks and Their Order of Execution

Preliminary Analysis is mainly a risk mitigation activity, aimed at identifying the characteristics, constraints and risks associated with the system and the development project, and assessing the feasibility of the development effort based on the knowledge acquired, thus avoiding the embarrassment - not to mention the financial implications - of committing to a project that has a significantly high possibility of failure.

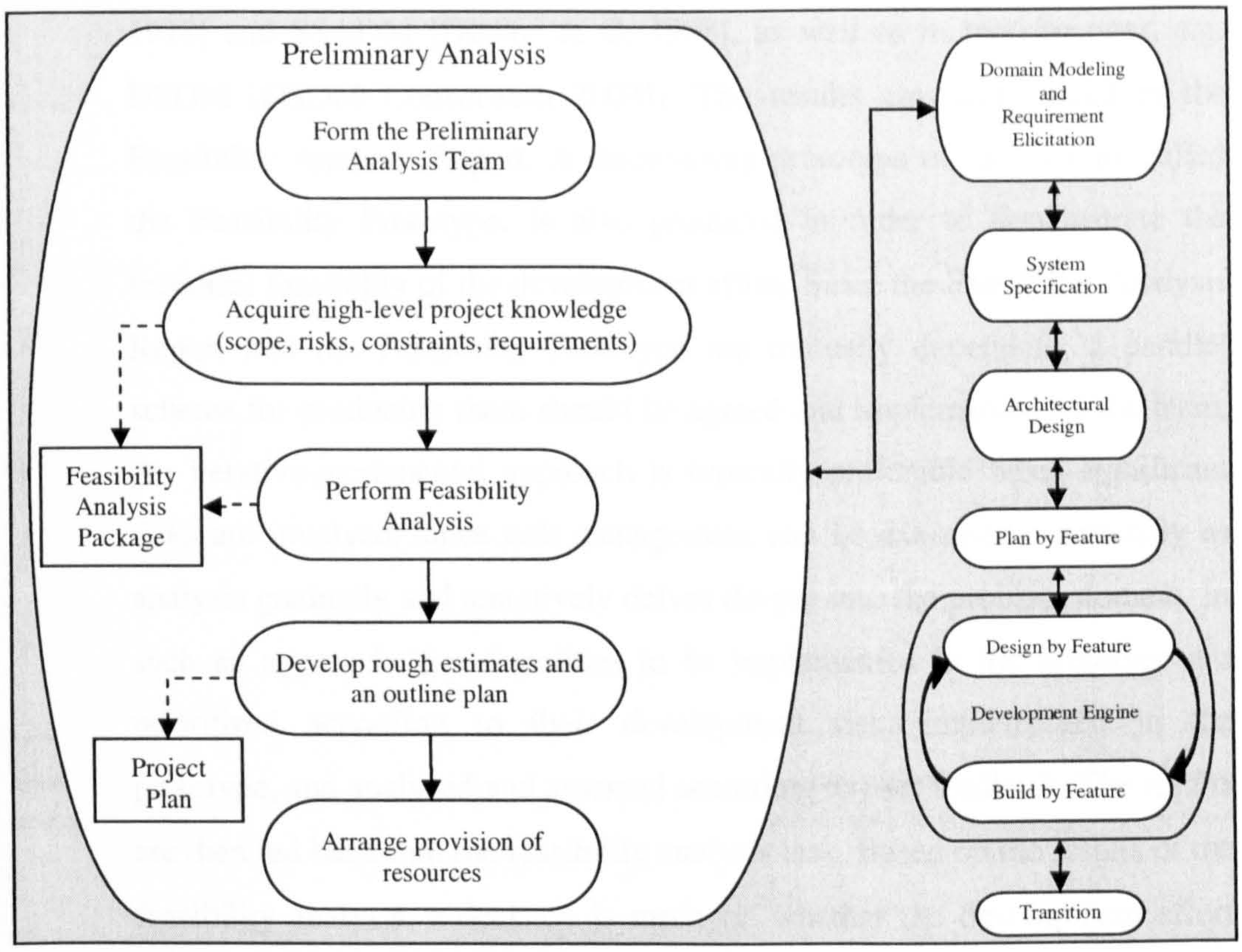

Figure 46. Preliminary Analysis subprocess: tasks and work-products

The tasks performed in this subprocess, typically performed sequentially, are as follows:

1. Form the Preliminary Analysis Team, consisting of a Project Manager in charge of the development effort, a Client Representative who makes the decisions on behalf of the client, a number of Ambassador Users and Domain Experts to help understand the complexities of the problem domain, and a number of Chief Programmers to develop the prototype of the system and provide technical expertise.
2. Acquire high-level knowledge as to the nature of the project, its scope, and the risks and constraints involved. A set of high-level requirements, expressed as *major feature sets* (*areas*) and their constituent *feature sets* (*activities*), is also compiled.
3. Perform the traditional activities of feasibility analysis, exploring alternative development approaches and architectural configurations paying special attention to technical, schedule, financial, operational and political feasibilities (performed in traditional methodologies, e.g. SA [DeMarco

1978] and SSADM [Downs et al. 1988], as well as in modern ones, e.g. DSDM [DSDM Consortium 2003]). The results are summarized in the Feasibility Analysis Report. A throw-away prototype of the system, called the Feasibility Prototype, is also produced in order to demonstrate the technical feasibility of the development effort. Since the Feasibility Analysis Report and the Feasibility Prototype are mutually dependent, a parallel scheme for producing them should be agreed and implemented by the team. An iterative-incremental approach is typically preferable when significant risks are involved, since risk management can be exercised continually as analysis gradually and tentatively delves deeper into the problem domain. In such an approach, functionalities to be implemented in the prototype are prioritized according to their development risk, implemented in the prototype, and analyzed and assessed according to user feedback. The results are then fed back into the feasibility analysis task. Based on the results of the feasibility analysis, a decision is made on whether the development effort should be commenced or aborted. Subsequent tasks (tasks 4 and 5 on this list) are only performed if the decision is to commence the project.

4. Develop rough estimates and an overall *Outline Plan* for the project.
5. Make arrangements for provision of resources for commencement of the project.

Work Products

The following work-products are produced in this subprocess:

- The results of the first two tasks are compiled in the *Feasibility Report*. The report may be complemented by a primitive prototype of the system (called the *Feasibility Prototype*), the main purpose of which is to demonstrate the scope and the technical feasibility of the project. The Feasibility Report includes information on:
  a. scope of the system and high-level requirements (expressed as feature sets)
  b. constraints and risks involved
  c. alternative approaches to developing the system and the results of their analysis

d. resources required

- Initial Project Plan (only produced if feasibility analysis results in the decision to commence the project)

Roles Involved

The Preliminary Analysis Team which carries out this subprocess consists of the following roles:

- Project Manager: Responsible for
  a. leading the team
  b. providing and managing resources
  c. facilitating operations
  d. resolving issues with the client and third parties
  e. enforcing standards and schedules
- Domain Expert: Helping understand the problem domain
- Ambassador User: Providing realistic and hands-on user feedback
- Chief Programmer: Developing the prototype of the system and providing technical expertise
- Client Representative: Responsible for:
  a. Defining constraints and high-level non-functional requirements
  b. making decisions as to stopping or commencing the project

Verification

The Preliminary Analysis Team verifies the results. The primary concern should be ensuring that constraints and risks (functional, managerial, technical, financial, schedule, political, etc.) likely to jeopardize the feasibility of the project have not been overlooked. Outside verification may be sought if deemed necessary by the client.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Decision reached as to the commencement or abandonment of the project, based on the results of the feasibility analysis
- Agreement made on the scope of the system and the constraints set by the client
- Commitments made on both sides to provide the services and resources expected from them as agreed by the team
- Approval of the overall Project Plan

5.2.2.2 Real-World Domain Modeling and Requirements Elicitation

As its name implies, this subprocess is where the problem domain is explored and modeled as is. The system is then introduced in the models and its requirements are identified. Figure 47 shows the tasks involved in this subprocess and the work-products produced or modified.

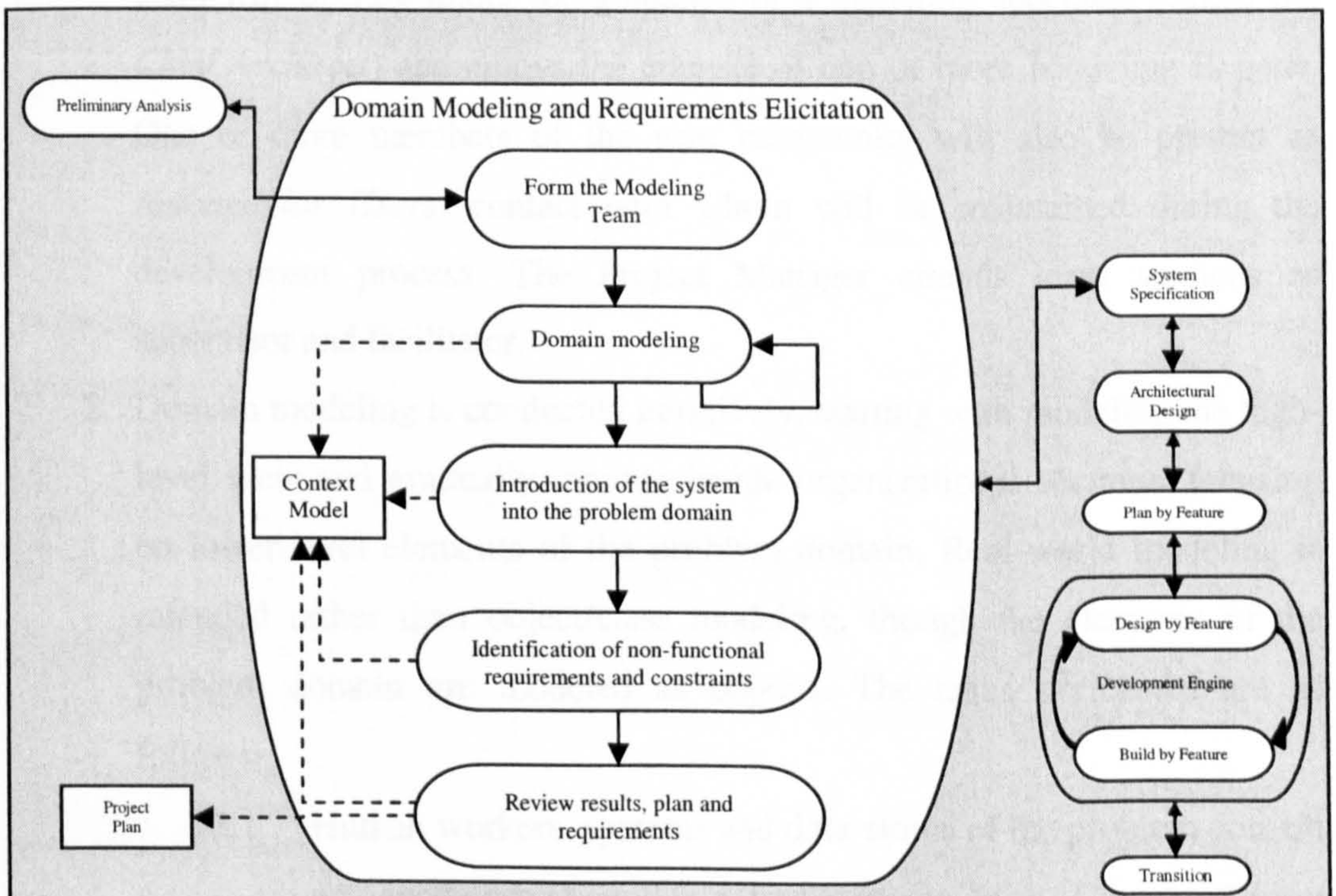

Figure 47. Domain Modeling and Requirements Elicitation subprocess: tasks and work-products

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Decision made by the client to commence the project
- Provision of resources required for performing the subprocess

Tasks and Their Order of Execution

The Context Model is produced by real-world modeling of the problem domain and adding the system to it as an object.

The tasks performed in this subprocess are as follows:

1. Form the Modeling Team, consisting of several software development professionals (*Chief Programmers*), and one or more Domain Experts. The team will operate under the guidance of a modeling specialist (called the *Chief Architect*) and enjoys the counsel of one or more Modeling Experts. One or more members of the user community will also be present as *Ambassador Users*, contact with whom will be maintained during the development process. The Project Manager attends team sessions as supervisor and facilitator.
2. Domain modeling is conducted iteratively, starting with modeling the high-level view and gradually moving inside organizational sections, focusing on lower-level elements of the problem domain. Real-world modeling is intended rather than object/class modeling, though the elements of the problem domain are modeled as objects. The tasks performed are as follows:
   2.1. Human workers, systems and data-stores of the problem domain are modeled as collaborating objects in a Context Object Model. A notation similar to UML collaboration diagrams is used for representing the model [OMG 2003], except that links are adorned with data/control flows, but not sequence numbers (example in Figure 56, page 314). Organizational boundaries are preserved, modeled through using UML packages and

component diagrams. The resulting functional models comprise the main bulk of the Context Model.

2.2. Typical transaction scenarios are modeled in UML activity diagrams (with swimlanes depicting the participating objects, as shown in Figure 58, page 315) and/or UML sequence diagrams (Figure 62, page 319). The resulting Context Interaction Models comprise the behavioural part of the Context Model.

2.3. Job descriptions and functionalities are expressed as *areas* (major feature sets), *activities* (feature-sets), and *features* [Coad et al. 1999, Palmer and Felsing 2002]. Feature lists are compiled and added to the Context Model.

2.4. A glossary of terms from the problem domain is compiled (Figure 63, page 321).

3. Introduction of the system into the problem domain: The system is added as an object to the Context Model (Context Object Models) and feature sets are assigned to the system through redistribution and/or duplication (Figure 57, page 314). New feature sets are added as deemed necessary by the Modeling Team, and the feature lists in the Context Model are updated. The features of the system comprise the functional requirements and as such will guide and focus the development activities throughout the rest of the lifecycle (Table 3, page 320). Typical scenarios of interaction with the system are also modeled and the Context Interaction Models are updated accordingly (Figure 60, page 317).
4. Non-functional requirements and constraints are identified and added to the Context Model.
5. The results, the plan and the requirements are reviewed and revised.

Work Products

The following work-products are produced in this subprocess:

- Context Model, consisting of:
    a. Context Object Models, with the system added and modeled as a problem domain object

b. Context Interaction Models, including models depicting typical scenarios of system usage
c. Feature Lists, including system features identified and set as functional requirements
d. Context Vocabulary, containing the glossary of terms compiled from the problem domain
e. Non-functional requirements and constraints

- Revised Project Plan

Roles Involved

The Modeling Team which carries out this subprocess consists of the following roles:

- Project Manager: Responsible for
  a. leading the team
  b. providing and managing resources
  c. facilitating operations
  d. resolving issues with the client and third parties
  e. enforcing standards and schedules
- Domain Expert: Helping understand the problem domain
- Ambassador User: Providing realistic and hands-on user feedback
- Chief Architect: Providing modeling expertise and guiding the modeling effort
- Modeling Expert: Providing advice on object-oriented modeling issues
- Chief Programmer: Development expert

Verification

The Modeling Team verifies the results, seeking advice from other Domain Experts if necessary. The primary concern should be ensuring that major functionality has been captured in the Context Model. Outside verification may be sought if deemed necessary by the client.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Context Model verified and approved by the team

5.2.2.3 System Specification

The System Specification subprocess focuses on the design of the system as an extension of the existing system using the types of elements originally found in the problem domain, and then converting the result to its computer system counterpart using object-oriented patterns. Figure 48 shows the tasks involved in this subprocess and the work-products produced or modified.

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Context Model approved by the Chief Architect as adequately capturing the problem Domain
- Provision of resources required for performing the subprocess

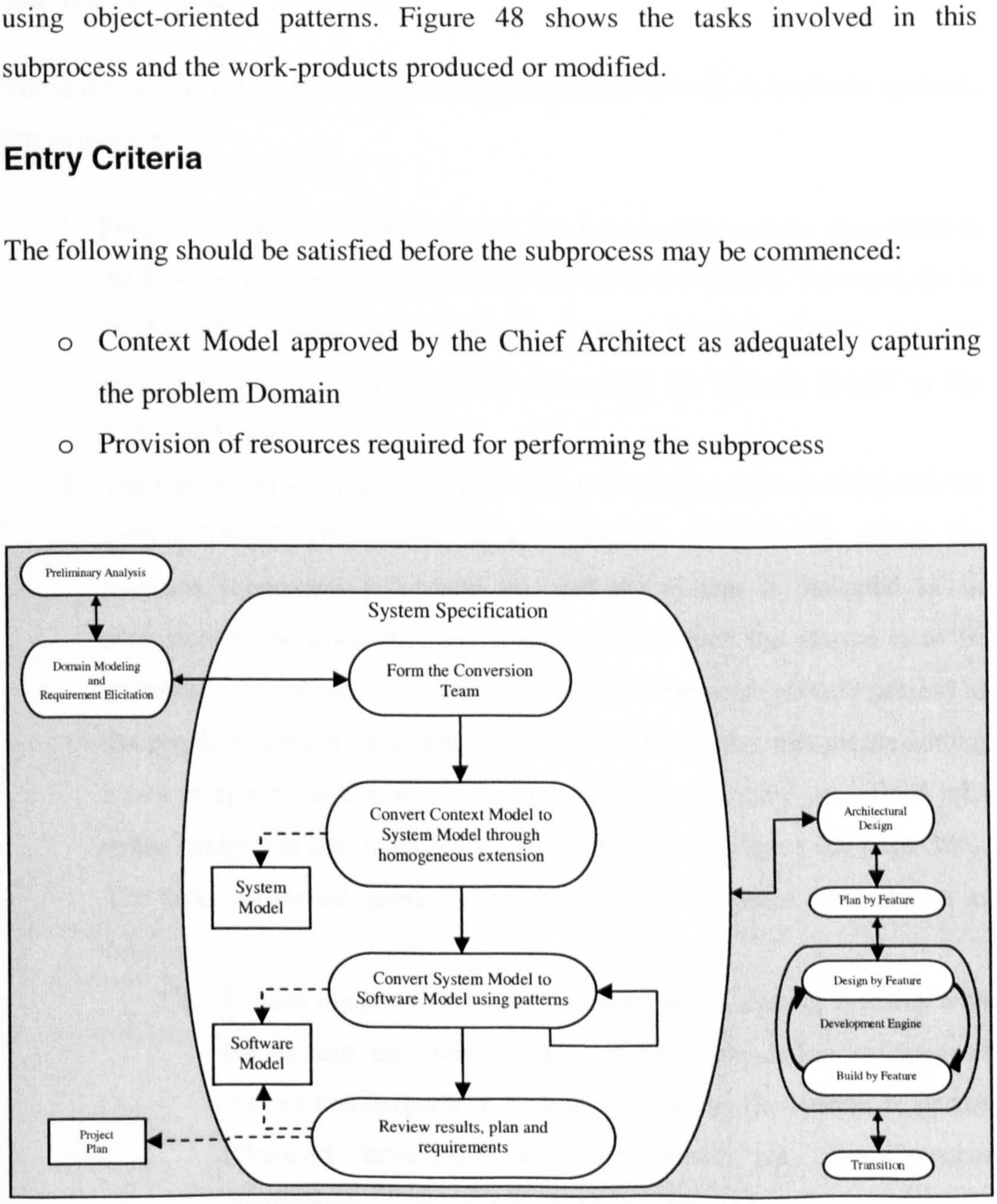

Figure 48. System Specification subprocess: tasks and work-products

Tasks and Their Order of Execution

Focus is shifted to the interior of the system, and the system is designed as an extension to the structure presently in place. The System Model thus built is then converted to the Software Model through using patterns to enhance *encapsulation* and *information hiding*. The goal is to provide a smooth transition from the problem domain to the solution domain and ultimately to the implementation domain, preserving object-orientation at all levels while keeping the artifacts tangible to all the people involved in the effort, especially Domain Experts. Features are given a pivotal role from the start and act as the basis binding the tasks and artifacts together.

The tasks performed in this subprocess, explained as relevant to business systems, are as follows:

1. Form the Model Conversion Team: The team consists of the same roles as the Modeling Team (the people may indeed be the same as those present in the Modeling Team), except that one or more Patterns Advisors are also present to provide advice during converting the System Model to the Software Model.
2. The Context Model built during the previous subprocess is converted to the System Model. The system object treated as a black box during the previous subprocess is opened up, and the system is designed as an extension to the organizational structure into which the system is to be ultimately introduced, using the same types of elements already present in the problem domain. In a business system, for instance, this means adding a new section or department consisting of staff performing predefined jobs using equipment and tools made available to them (Figure 65, page 324). The tasks performed, using examples relevant to business systems, are as follows:
    2.1. Human elements, systems and data stores sharing features with the system are moved inside system boundaries or assigned system counterparts if their attachment to the system is partial (elements interacting with the system may have system counterparts). The system is to be designed as a homogeneous

extension of the problem domain; this means that the same types of entities as seen in the problem domain are used for designing the system.

2.2. Each data-store and each flowing data object is assigned to a *custodian*; any access to any such object should be made via the custodian. Flowing-data custodians are file/document movers, transferring the file or document put in their custody between processing clerks. In a business system, data-store custodians are akin to file clerks and archive keepers. It should be noted that there is no limit on the number of staff assigned to the system. The resulting System Object Models – depicting non-sequenced data/control flowing among objects (intra-system and extra-system) – comprise the functional component of the System Model (Figure 68, page 327).

2.3. Typical interaction scenarios are identified, and relevant behavioural models (typically activity diagrams and interaction diagrams) are produced for each of the system's feature sets. The resulting System Interaction Models comprise the Behavioural component of the System Model (Figure 69, page 328).

2.4. Feature sets and features are assigned to the system elements based on the functionality assigned to the system as a whole and the interaction models produced in the previous task (Table 4, page 330). This is analogous to job definition and task assignment in manual business systems. The list of feature sets and features assigned to the objects will later be used in determining class methods.

2.5. Review and revision of the requirements of the system

2.6. Review of the resulting System Model

3. The System Model produced so far is converted to the Software Model by applying patterns to redistribute functionality among system objects. The model thus produced will ultimately be perfected and extended during later design and implementation subprocesses. The tasks performed, explained as relevant to business systems, are as follows:

3.1. Patterns are applied to the System Model to iteratively redistribute features among objects (i.e. processing staff and custodians) in order to enhance encapsulation, increase cohesion and reduce coupling, and also to introduce architecture. Reengineering patterns, especially those suggested in [Demeyer et al. 2003] for redistributing responsibilities among objects are of utmost use in the starting iterations. These typically include:

- Moving behaviour close to data
- Eliminating navigation
- Splitting up God classes (Blobs)

A number of Refactoring patterns proposed in [Fowler 1999] can also be used in conjunction with the above (indeed, some of them already are a part of the above patterns):

- Move method (feature)
- Move field
- Extract class
- Inline class
- Hide delegate
- Remove middle man
- Encapsulate field

Design patterns can be used in later iterations to help implement specific architectures and mechanisms typically present in the problem domain and tangible to users. Applying these patterns not only results in structures familiar to the user, but also facilitates the translation of these structures into solution domain and implementation domain class structures. Design patterns especially useful in this context are:

- GoF patterns [Gamma et al. 1995]:
  - Wrapper:
    - Adapter: to standardize interfaces
    - Decorator: for dynamic reassignment of responsibilities (features)
  - Facade: for inter-departmental/inter-group interfacing

- Proxy: already used in assigning custodians, can also be used for adding middle men if necessary
- Command: to encapsulate features or feature-sets (possibly transaction processing chains), making it possible to pass them like processing instruction manuals
- Mediator: to centralize complex inter-object communications (analogous to appointing a facilitator or manager)
- Observer: to implement change monitors (auditors/supervisors) in order to ensure consistency and the enforcement of business rules
- State: to facilitate dynamic change of roles (dynamic job descriptions)
- Strategy: to enable dynamic assignment of algorithms (changing work procedures)
- Visitor: for setting up specialized service-provider departments/sections, with the knowledge of how to provide specific kinds of service to each and every client department/section

- GoV patterns [Buschmann et al. 1996]:
  - Broker: for defining inter-departmental go-betweens (dispatch-offices)
  - Command Processor: to define special jacks-of-all-trades; i.e. dynamically configurable clerks/teams that take part in the processing once they are supplied with the know-how (commands)
  - Layers: for implementing hierarchical departmental/management organizational structures

- Master-Slave: to implement certain team and management structures
- Pipes-and-Filters: to define overall transaction-processing architecture

Antipatterns can also be of use in the redistribution procedure, especially the Poltergeist and Swiss-Army-Knife antipatterns [Brown et al. 1998]. The redistribution procedure is devised in such a way as to resolve the problems typically afflicting analysis approaches based on object-oriented real-world modeling [Isoda 2001]. Objects irrelevant to the system and actor-counterparts without any justification for existence in the system are gradually disposed of, and relationships not belonging to the system are not introduced into the models because of the data-flow oriented and feature-driven nature of the System Model and the redistribution procedure (Figure 71, page 331). Behavioural models are updated in each iteration of the redistribution procedure.

3.2. Applying the patterns ultimately results in custodian objects being merged with the data objects they had under custody. This marks the transition from the problem-domain-based system to the computer system, signifying the transition to solution domain. The resulting Software Object Models comprise the functional component of the Software Model. Class diagrams are then produced based on the Object Models, depicting the classes in the system and their relationships. Inheritance hierarchies are introduced in order to enhance abstraction (patterns for refactoring inheritance can be of use in this context [Fowler 1999]). The Software Class Models thus produced comprise the main structural component of the Software Model.

3.3. Behavioural diagrams inherited from the System Model are updated according to the new Software Class/Object Models. The resulting Software Interaction Models comprise the behavioural component of the Software Model. Message passing should be clearly depicted (Figure 76, page 336).

3.4. Preparation of initial versions of class and method prologues

3.5. Review and revision of the requirements

3.6. Review of the resulting Software Model

4. Review the results of the subprocess, the plan and the requirements

Work Products

The following work-products are produced in this subprocess:

- System Model
  a. System Object Models
  b. System Interaction Models
  c. Revised Features List
  d. Revised list of non-functional requirements and constraints
- Software Model
  a. Software Object/Class Models, consisting of Object Models depicting typical links and data flows among system objects, and Class Models, showing the classes of the system and their relationships.
  b. Software Interaction Models
  c. Initial versions of class and method prologues
  d. Revised list of features
  e. Revised list of non-functional requirements and constraints
- Revised Project Plan

Roles Involved

The Model Conversion Team which carries out this subprocess consists of the following roles:

- Project Manager: Responsible for
  a. leading the team
  b. providing and managing resources
  c. facilitating operations
  d. resolving issues with the client
  e. enforcing standards and schedules

- Domain Expert: Helping understand the problem domain
- Ambassador User: Providing realistic and hands-on user feedback
- Chief Architect: Providing modeling expertise and guiding the modeling effort
- Modeling Expert: Providing advice on object-oriented modeling issues
- Chief Programmer: Development expert
- Patterns Advisor: Providing expertise on patterns and their application for redistributing functionality among system elements

Verification

The Model Conversion Team verifies the results, making sure the conversions have not resulted in lost information or redundant clutter. The primary concern should be ensuring that features have been preserved during model conversion, and have indeed been realised and implemented by the designed system, in the System Model as well as the Software Model. Outside verification may be sought if deemed necessary by the team.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Software Model verified and approved by the team
- Features list reviewed and approved as consistent with the Software Model

5.2.2.4 Architectural Design

Focused on designing an overall implementation-specific architecture for the system, this subprocess defines the infrastructure based on which multi-team, iterative-incremental detailed design and implementation will be carried out in the following subprocesses. Figure 49 shows the tasks involved in this subprocess and the work-products modified.

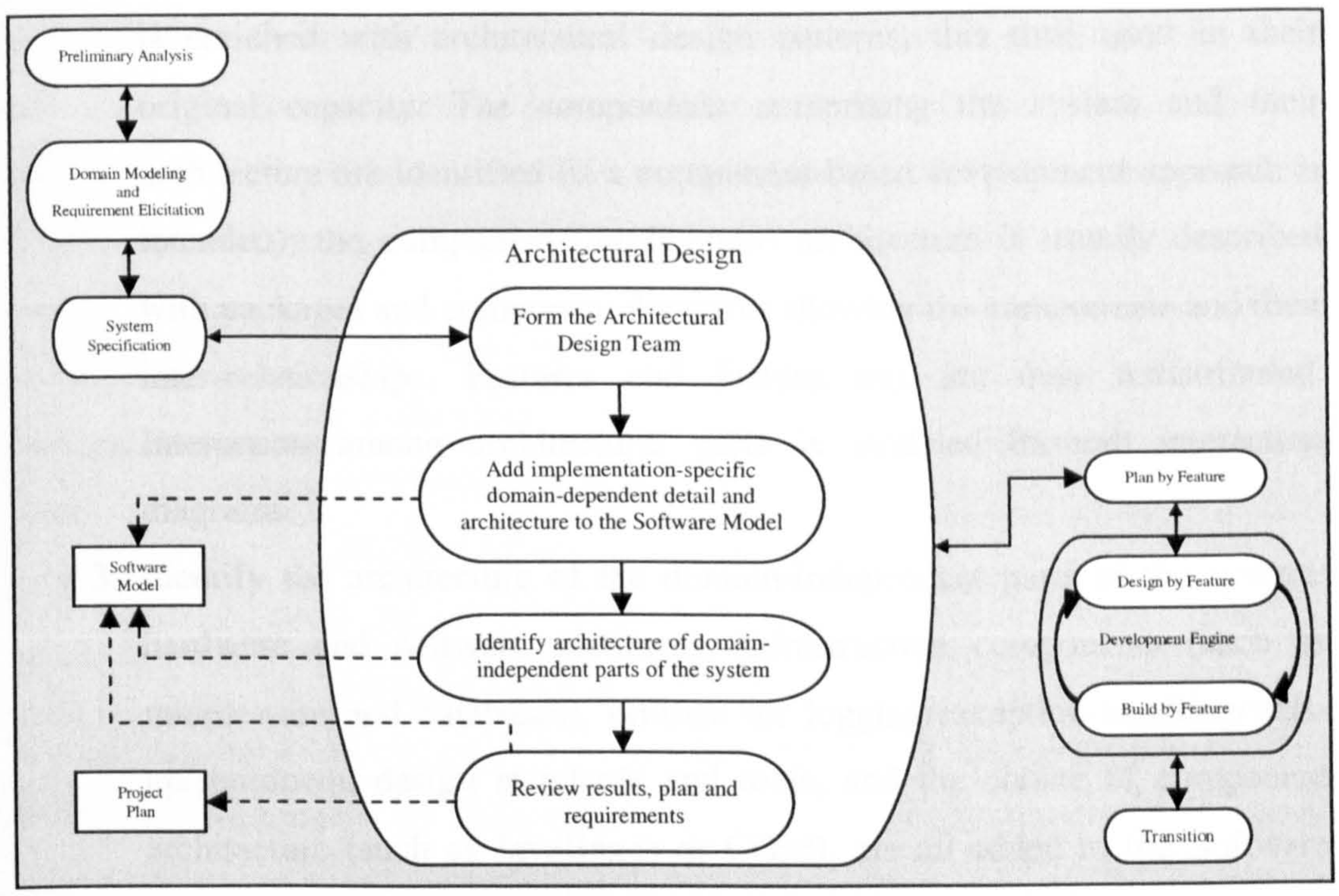

Figure 49. Architectural Design subprocess: tasks and work-products

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Software Model approved by the Conversion Team as adequately complete for architectural design to start
- Provision of resources required for performing the subprocess

Tasks and Their Order of Execution

The tasks performed in this subprocess are as follows:

1. Form the Architectural Design Team: consisting of the same roles as the Conversion Team active in the previous subprocess, except that Domain Experts are replaced by Design Experts with knowledge on architectural design techniques and domain-independent technologies.
2. Convey the Software Model to the implementation domain through adding implementation-specific detail and restructuring it in order to facilitate implementation and accommodate the domain-independent parts of the system. The user interface is designed in this task, and the Software Model

is enriched with architectural design patterns, this time used in their original capacity. The components comprising the system and their architecture are identified (if a component-based development approach is intended); the component (*application*) architecture is usually described with packages and component diagrams showing the components and their inter-relationships. Features and feature sets are duly redistributed. Interaction among architectural parts is modeled through interaction diagrams.

3. Identify the architecture of the domain-independent parts of the system: hardware and software platforms, infrastructure components (such as middleware and databases), utilities for logging/exception-handling/start-up/shutdown, design standards and tools, and the choice of component architecture (such as JavaBeans or COM), are all added to the Software Model. Component diagrams are used to show these physical components and their inter-relationships. Interactions are shown in collaboration diagrams.
4. Review the results, the plan and the requirements

Work Products

The following work-products are produced in this subprocess:

- Revised Software Model
    a. Revised versions of Software Object/Class Models spanning architectural information, user interface and domain-independent components added during the subprocess; consisting of Object Models depicting typical links among objects, and Class Models, showing the classes and their relationships.
    b. Revised versions of Software Interaction Models
    c. Revised versions of class and method prologues
    d. Revised list of features covering feature sets assigned to architectural and domain-independent units
    e. Revised list of non-functional requirements and constraints
- Revised Project Plan

Roles Involved

The Architectural Design Team which carries out this subprocess consists of the following roles:

- Project Manager: Responsible for
  a. leading the team
  b. providing and managing resources
  c. facilitating operations
  d. resolving issues with the client and third parties
  e. enforcing standards and schedules
- Design Expert: providing knowledge on architectural design techniques and domain-independent technologies
- Ambassador User: Providing realistic and hands-on user feedback
- Chief Architect: Providing modeling expertise and guiding the modeling effort
- Chief Programmer: Development expert
- Patterns Advisor: Providing expertise on patterns and their application for redistributing functionality among system elements

Verification

The Architectural Design Team verifies the results, making sure all major architectural and domain-independent elements needed for implementing the system are identified. The primary concern should be ensuring that links between domain entities and domain-independent components have been adequately set up, and system features have been preserved during design and have indeed been realised and implemented by the designed system. The user interface should be validated by the Ambassador Users. The only remaining design activity is the detailed design of the classes, which is performed during the cycles of the iterative-incremental development engine in the penultimate subprocess of the lifecycle. Outside verification may be sought if deemed necessary by the team.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Software Model (including the user interface) verified and approved by the team as covering the implementation-specific architectural and domain-independent components necessary for the implementation process to commence
- Features list reviewed and approved as consistent with the Software Model

5.2.2.5 Planning by Feature

This subprocess is where the feature-driven iterative-incremental engine of the development process is planned and the appropriate feature-development task assignments are made. Figure 50 shows the tasks involved in this subprocess and the work-products produced or modified.

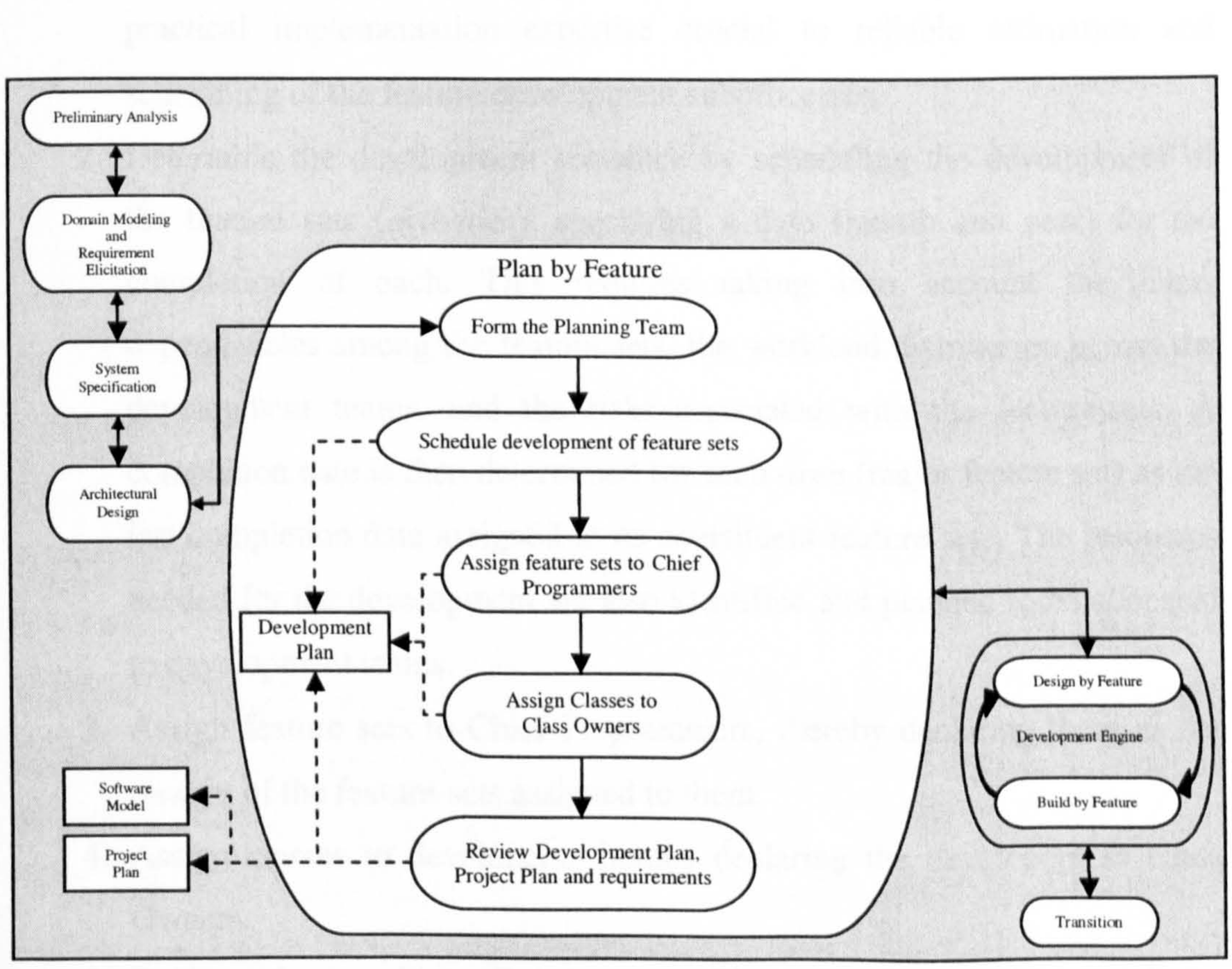

Figure 50. Plan by Feature subprocess: tasks and work-products

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Software Model (especially the Features List therein) approved by the Architectural Design Team as adequately complete and stabilized
- Provision of resources required for performing the subprocess

Tasks and Their Order of Execution

The tasks performed in this subprocess are as follows:

1. Form The Planning Team: The Planning Team typically consists of the Project Manager as the leader of the team, the Development Manager as the resource manager and coordinator of the Features Teams responsible for the ultimate implementation of the system, and the Chief Programmers involved in the development as leaders of Features Teams, providing practical implementation expertise crucial to reliable estimation and scheduling of the feature development subprocesses.
2. Determine the development sequence by scheduling the development of the feature sets (*activities*), specifying a date (month and year) for the completion of each. This requires taking into account the inter-dependencies among the feature sets, the workload distribution across the development teams, and the risks associated with the feature-sets. A completion date is then determined for each *area* (major feature set) as the last completion date assigned to its constituent feature sets. The resources needed for the development are also identified and planned to be allocated to development teams.
3. Assign feature sets to Chief Programmers, thereby declaring them as the *owners* of the feature-sets assigned to them.
4. Assign classes to developers, thereby declaring the developers as Class Owners.
5. Review the resulting Development Plan, the Project Plan and the requirements

Work Products

The following work-products are produced in this subprocess:

- Development Plan, covering
    a. Development schedule
    b. Feature-set and class assignments
    c. Resource allocations
- Revised Features List (in the Software Model)
- Revised Project Plan

Roles Involved

The Planning Team which carries out this subprocess consists of the following roles:

- Project Manager: Responsible for
    a. leading the team
    b. providing and managing resources
    c. facilitating operations
    d. resolving issues with the client and third parties
    e. enforcing standards and schedules
- Development Manager: The resource manager and coordinator of the Features Teams during the iterative development subprocesses
- Chief Programmer: Development expert

Verification

The Planning Team verifies the results, making sure that all feature sets have been scheduled and assigned to Chief Programmers, and all classes have been assigned to Class Owners. The primary concern should be ensuring that reasonable completion dates have been determined based on inter-dependencies among the feature sets, the risks associated, and the workload distribution across the development teams. Care should also be taken in ensuring that all major resources required have been identified and verified as obtainable and ready to be allocated. Outside verification may be sought if deemed necessary by the team.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Development Plan verified and approved by the team as reasonably complete
- Features list reviewed and approved as consistent with the Software Model

5.2.2.6 Feature-Driven Iterative-Incremental Development

Strands of design-and-build iterations start off as each Chief Programmer selects the set of features (called the *Work Package*) that should be developed in each of the iterations performed under his supervision, and forms a Features Team to do the job. A Chief Programmer selects features and schedules his iterations according to the Development Plan. Typically, at any point during this development period, several iterations are being performed concurrently, some of them supervised by the same Chief Programmer, with each of the Class Owners taking part in several iteration-teams simultaneously.

5.2.2.6.1 Design by Feature

In each iteration of this subprocess, detailed design of the classes and methods involved in the implementation of the features in the Work Package is carried out. Figure 51 shows the tasks involved in this subprocess and the work-products produced or modified.

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Software Model (especially the Features List therein) approved by the Architectural Design Team as adequately complete and stabilized to be used as the basis for implementation
- Development Plan verified and approved by the Planning Team as reasonably complete
- Provision of resources required for performing the subprocess

Tasks and Their Order of Execution

The tasks performed in this subprocess are as follows:

1. Form a Features Team, which will design and build the feature(s) selected for development in the current iteration under the supervision of the Chief Programmer who owns the features. A Work Package should first be defined by the Chief Programmer, showing the projected completion date of the current iteration and the features chosen for detailed design and implementation therein. After defining the Work Package and identifying the set of classes that might be involved in the realization of the features in the Work Package, the Chief Programmer brings together the owners of these classes. Included in the team are one or more Modeling Experts who are commissioned to help with the design modeling. One or more Ambassador Users are also present to provide feedback on the design.
2. Study the Software Model in order to obtain a better understanding of the particulars of the features. This task is usually undertaken for high-risk features, the development of which usually requires a deeper understanding of the data, algorithms, and constraints involved.
3. Refine and complete the sequence diagrams in the Software Interactions Models, which as the behavioural component of the Software Model, are required to show how software objects should interact at run-time in order to implement each of the features. The features team also meticulously logs the alternative design models it has explored, as well as the constraints and assumptions that apply.
4. Refine the Software Object/Class Models so that they support the sequence diagrams produced in the previous task. This usually means that new elements are added to the model, some of the existing elements are changed, and refactoring is necessitated as a consequence.
5. Write Class- and Method-prologues for the elements of the Software Object Models. These relatively low-level design details are produced by the Class Owners as the last design artifacts needed before the coding can commence.

6. Design inspection is performed by the Features Team (possibly in consultation with other people involved in the project) in order to verify the integrity of the design artifacts produced.
7. Review and revise the Work Package (the features and the iteration schedule)

The products of this subprocess are transferred to the next subprocess as a package. This *Design Package* consists of the sequence diagrams produced, the refinements made to the Software Model, the prologues, and the notes on the design alternatives explored, constraints, and assumptions.

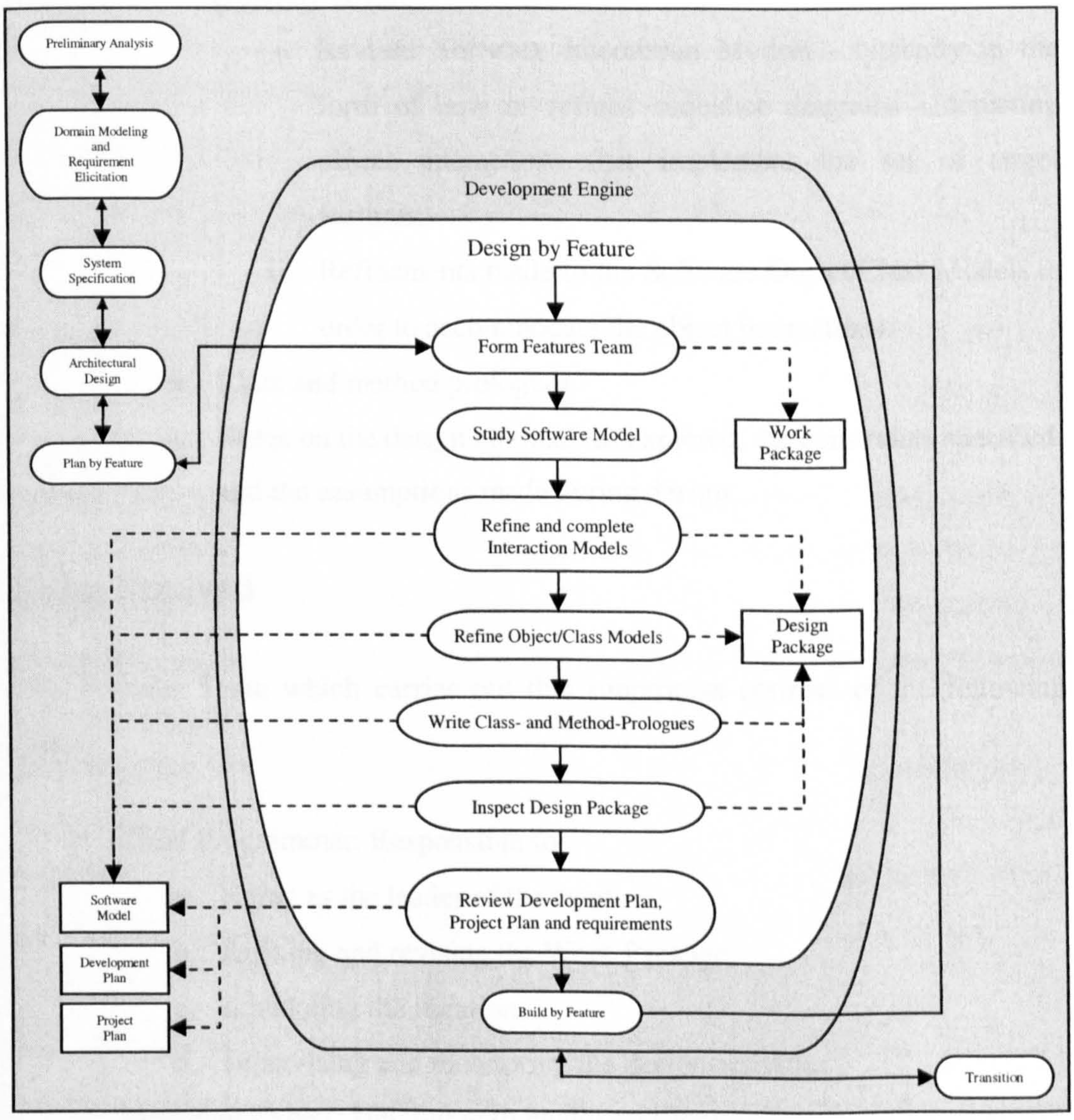

Figure 51. Design by Feature subprocess: tasks and work-products

Work Products

The following work-products are produced in this subprocess:

- Work Package: consisting of:
    a. A set of features that the Chief Programmer leading the team has chosen to be designed and built in the iteration
    b. A projected completion date for the iteration.
- Design Package: consisting of:
    a. Refinements made to the Software Model in order to facilitate the implementation of the features in the Work Package. The revisions typically cover:
        i. Revised Software Interaction Models - typically in the form of new or refined sequence diagrams - depicting object interactions that implement the set of target features.
        ii. Refinements made to the Software Object/Class Models in order to accommodate the object interactions.
    b. Class and method prologues
    c. Notes on the design alternatives explored, the constraints specified, and the assumptions made during design

Roles Involved

The Features Team which carries out this subprocess consists of the following roles:

- Chief Programmer: Responsible for
    a. acting as the leader of the team
    b. defining and revising the Work Package
    c. scheduling the iterations
    d. supervising and monitoring the design activities
- Modeling Expert: Helping with model revisions
- Ambassador User: Providing feedback on the design
- Class Owner: Undertaking the detailed design of software classes and their methods

Features Teams are collectively coordinated and provided with resources by the Development Manager.

Verification

The Features Team verifies the results, making sure that all features in the Work Package have been covered. The primary concern should be ensuring that the behavioural models introduced or revised during this subprocess do indeed implement the features specified in the Work Package. Care should also be taken in ensuring that necessary refinements and refactorings are made to other models of the Software Model, especially the Class Models, and that no inconsistencies have crept into the Software Model as the result of the revisions.

Outside verification may be sought if deemed necessary by the team or the Development Manager.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Work Package verified as feasible
- Design Package verified and approved by the Features Team as covering the features in the latest version of the Work Package, and ready for implementation in the next subprocess
- Software Model verified and approved by the Features Team as consistent and updated with the necessary revisions

5.2.2.6.2 Build by Feature

This subprocess is where the Design Package produced in the previous subprocess is implemented, tested and integrated with the system built so far. Figure 52 shows the tasks involved in this subprocess and the work-products produced or modified.

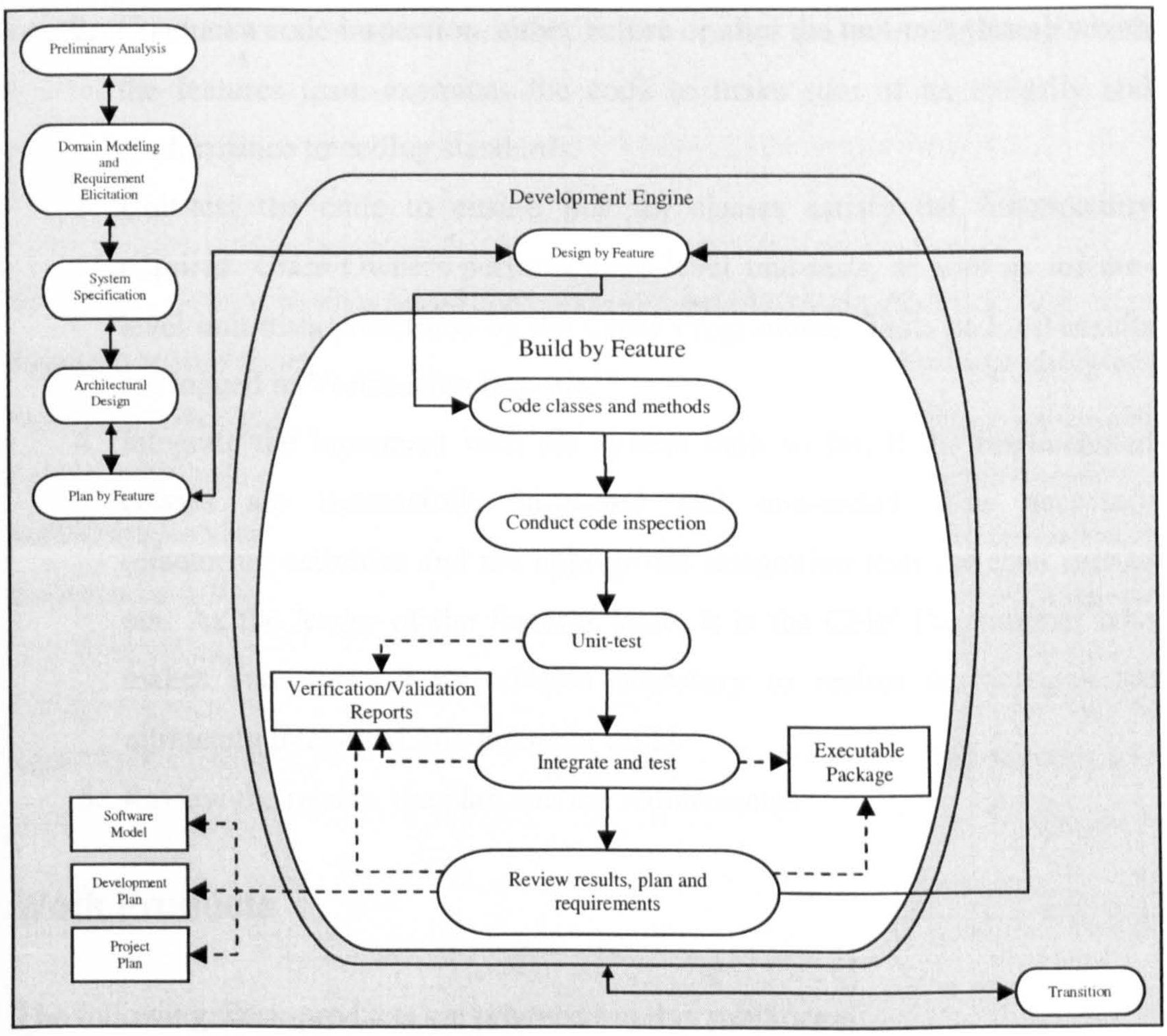

Figure 52. Build by Feature subprocess: tasks and work-products

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Design Package verified and approved by the Features Team as ready to be implemented
- Provision of resources required for performing the subprocess

Tasks and Their Order of Execution

The tasks performed in this subprocess are as follows:

1. Implement classes and methods according to the specifications given in the Design Package. Each of the Class Owners implements the necessary items (including the unit-testing code) in the classes he or she owns.

2. Conduct a code inspection, either before or after the unit-test, during which the features team examines the code to make sure of its integrity and conformance to coding standards.
3. Unit-test the code to ensure that all classes satisfy the functionality required. Class Owners perform class-level unit-tests, as well as feature-level unit-tests prescribed by the Chief Programmer. Tests and test-results are logged in Verification Reports.
4. Integrate the increment with the system built so far, if the implemented classes are successfully inspected and unit-tested. The necessary refactoring activities and the appropriate integration tests are then carried out. As the leader of the features team, it is the Chief Programmer who makes sure that all the classes necessary to realize the features are ultimately integrated into the main build.
5. Review the results, the plan and the requirements

Work Products

The following work-products are produced in this subprocess:

- Revised Executable Package, with the executable increment built in the iteration (consisting of system executables and run-time components) properly integrated.
- Verification and Validation Reports: containing the results of the tests and the feedback provided by Ambassador Users
- Revised Features List (preserving consistency with the Software Model)
- Revised Project- and Development Plans

Roles Involved

The Features Team which carries out this subprocess consists of the following roles:

- Chief Programmer: Responsible for
    a. acting as the leader of the team
    b. revising the Work Package
    c. scheduling the iterations

d. supervising and monitoring the implementation and test activities

- Modeling Expert: Helping with model interpretation and revision
- Ambassador User: Providing validation feedback on the system
- Class Owner: Undertaking the implementation and testing of software classes and their methods

Features Teams are collectively coordinated and provided with resources by the Development Manager.

Verification

The Features Team verifies the results, making sure that all features in the Work Package have been implemented and tested. The primary concern should be ensuring that all necessary unit- and integration tests have been carried out, and system validation has been performed based on feedback provided by Ambassador Users. Care should also be taken in ensuring that necessary refinements and refactorings are made to the Executable Package after the increment has been integrated, and that all verification and validation results are logged in relevant reports. Outside verification may be sought if deemed necessary by the team or the Development Manager, especially in case of crucial and high-risk features.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Executable Package verified and approved as satisfying the features in the Work Package, and validated by Ambassador Users.
- Verification and Validation Reports properly produced
- Revised Features List reviewed and approved as consistent with the Software Model

5.2.2.7 Transition

The Transition subprocess is mainly focused on system-wide verification and validation and the deployment of the implemented system in the user environment.

Figure 53 shows the tasks involved in this subprocess and the work-products produced or modified.

Entry Criteria

The following should be satisfied before the subprocess may be commenced:

- Executable Package verified and approved by the Development Manager and the Project Manager as ready to be deployed
- Provision of resources required for performing the subprocess

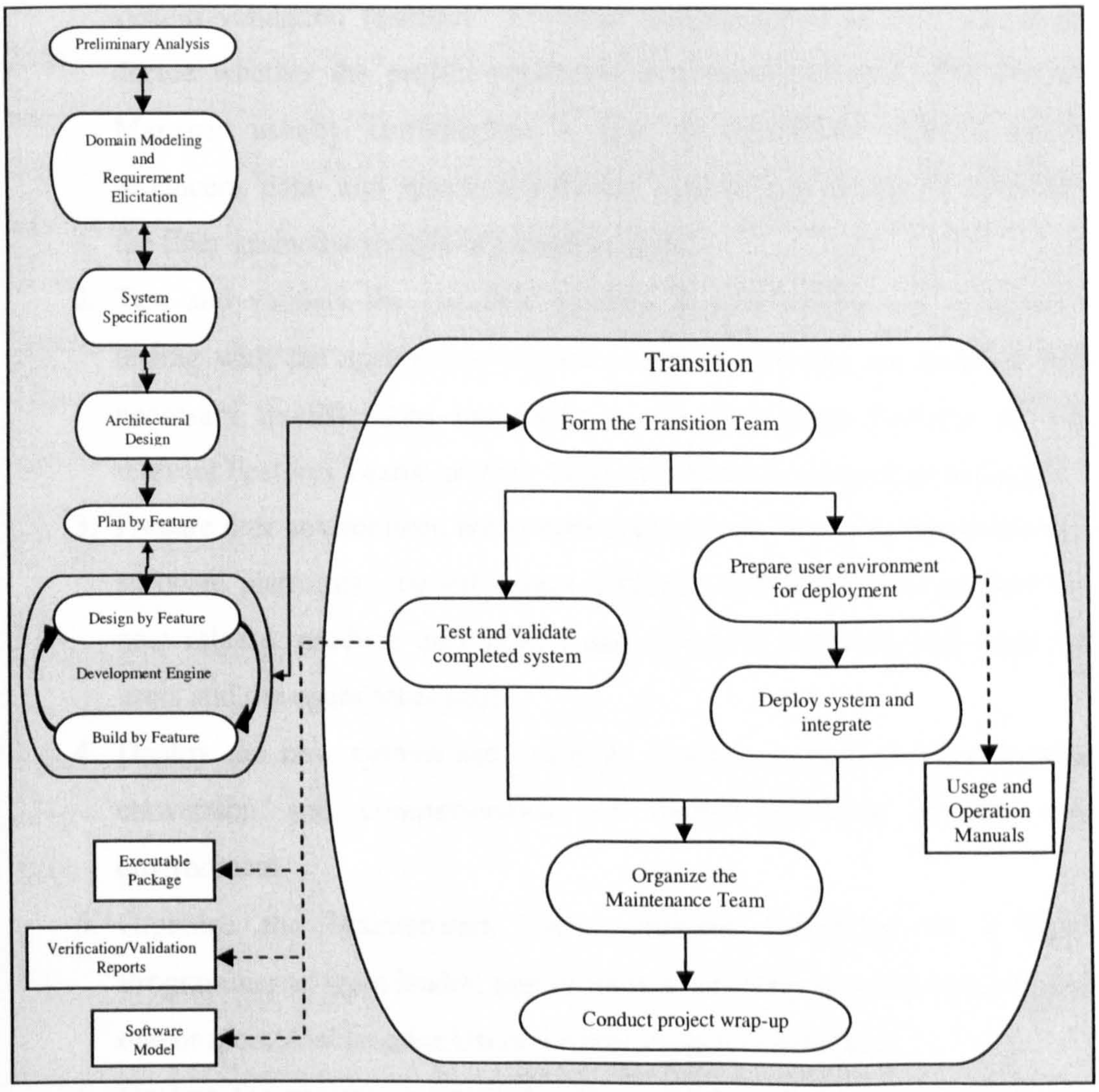

Figure 53. Transition subprocess: tasks and work-products

Tasks and Their Order of Execution

The tasks performed in this subprocess (many of which in parallel) are as follows:

1. Form the Transition Team: The Transition Team typically consists of the Project Manager as the leader of the team, the Development Manager as the coordinator of the Features Teams responsible for rectifying the system problems encountered during transition, the Chief Programmers involved in system correction as leaders of Features Teams, Class Owners as developers responsible for the ultimate debugging and testing during system correction activities, and one or more Ambassador Users providing system validation feedback. A Client Representative is also present to decide whether the project objectives have been achieved. The Project Manager usually commissions a host of instructors, documentation producers, data- and system conversion experts, and others to undertake the finer-grained activities of transition tasks.
2. Test and validate the complete system: System testing and acceptance testing with the appropriate reports produced. Defects are rectified and necessary modifications are made to the Executable Package by the relevant Features Teams, and the Software Model is updated accordingly.
3. Prepare user environment for system deployment: Set up the hardware and software platforms, convert legacy databases and systems to support the new release, produce user guides and operation manuals, and train the users and the operational staff
4. Deploy the new system and integrate it with existing systems: System conversion and commencement of system operation in the user environment
5. Organize the Maintenance Team, typically consisting of a Chief Programmer as team leader, one or more Class Owners as developers, and one or more Ambassador Users for providing user feedback
6. Declare the project as finished: When Deployment is carried out to the satisfaction of the Transition Team, especially the Client Representative, project wrap-up is conducted; the project is reviewed and the lessons learned from the project are compiled and recorded in order to be used in future projects.

Work Products

The following work-products are produced in this subprocess:

- Revised Executable Package, with the necessary corrections and modifications applied based on the results of the system level verification and validation carried out in the subprocess
- Revised Software Model
- Verification and Validation Reports: containing the results of the system tests and the feedback provided by Ambassador Users
- User Guides and Operation Manuals

Roles Involved

The Transition Team which carries out this subprocess consists of the following roles:

- Project Manager: Leader of the team
- Development Manager: Coordinator of the Features Teams carrying out the corrections and alterations to the executable system
- Client Representative: Deciding whether the project has been successfully concluded
- Chief Programmer: Leaders of Features Teams
- Class Owners: Active in the Features Teams, implementing corrections and modifications made to the Executable Package
- Ambassador User: Providing system validation feedback

Verification

The Transition Team verifies the results, making sure that the system has been verified as satisfying the Features List, and validated and deployed in the user environment to the satisfaction of Ambassador Users and the Client Representative. The primary concern should be ensuring that all necessary tests have been carried out, and that the operational platforms and the system have been correctly installed. Care should also be taken in ensuring that users and operational staff are properly trained and maintenance teams have been set up and organized.

Outside verification may be sought if deemed necessary by the team or the Client Representative.

Exit Criteria

The following should be satisfied before the subprocess may be concluded:

- Installed system verified and approved as satisfying the features in the Features List, and validated by Ambassador Users and the Client Representative
- Verification/Validation Reports and usage/operation manuals duly produced
- Users and operational staff trained, and Maintenance Team organized

5.2.3 Work-Product-Centred Description of the Methodology

In the work-product-centred view of the methodology, the focus is on the artefacts produced, their structure and their dependencies, with tasks and producers viewed in the context of – and secondary to – the work-products. Figure 54 shows a high-level work-product-centred view of the methodology, depicting the usage span of the work-products and the points in time when they affect each other during the enactment of the methodology.

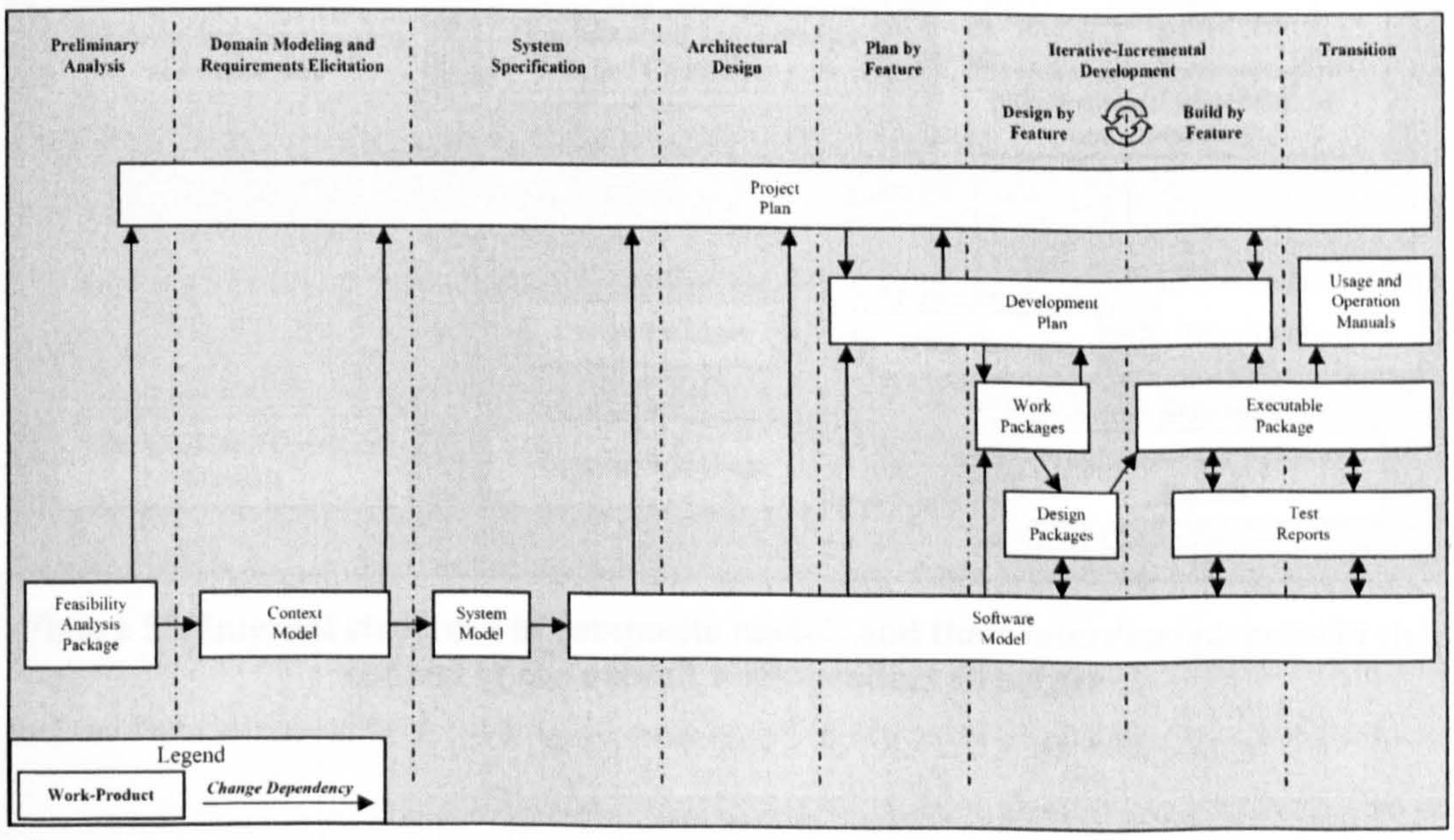

Figure 54. Work-products of the methodology: usage span, dependencies and mutual effects

Many of the packages and models produced in the methodology are composite work-products. Figure 55 shows the internal structure of these composite work-products and summarizes the interdependencies. Of the work-products shown in this figure, only those which are specific to the methodology are described in detail; those work-products for which a well-established template and production method already exists (and has been approved as sufficient for the needs of this methodology) have been excluded from the work-product-centred description of the methodology, on the grounds that any description will be a repetition of what is already known.

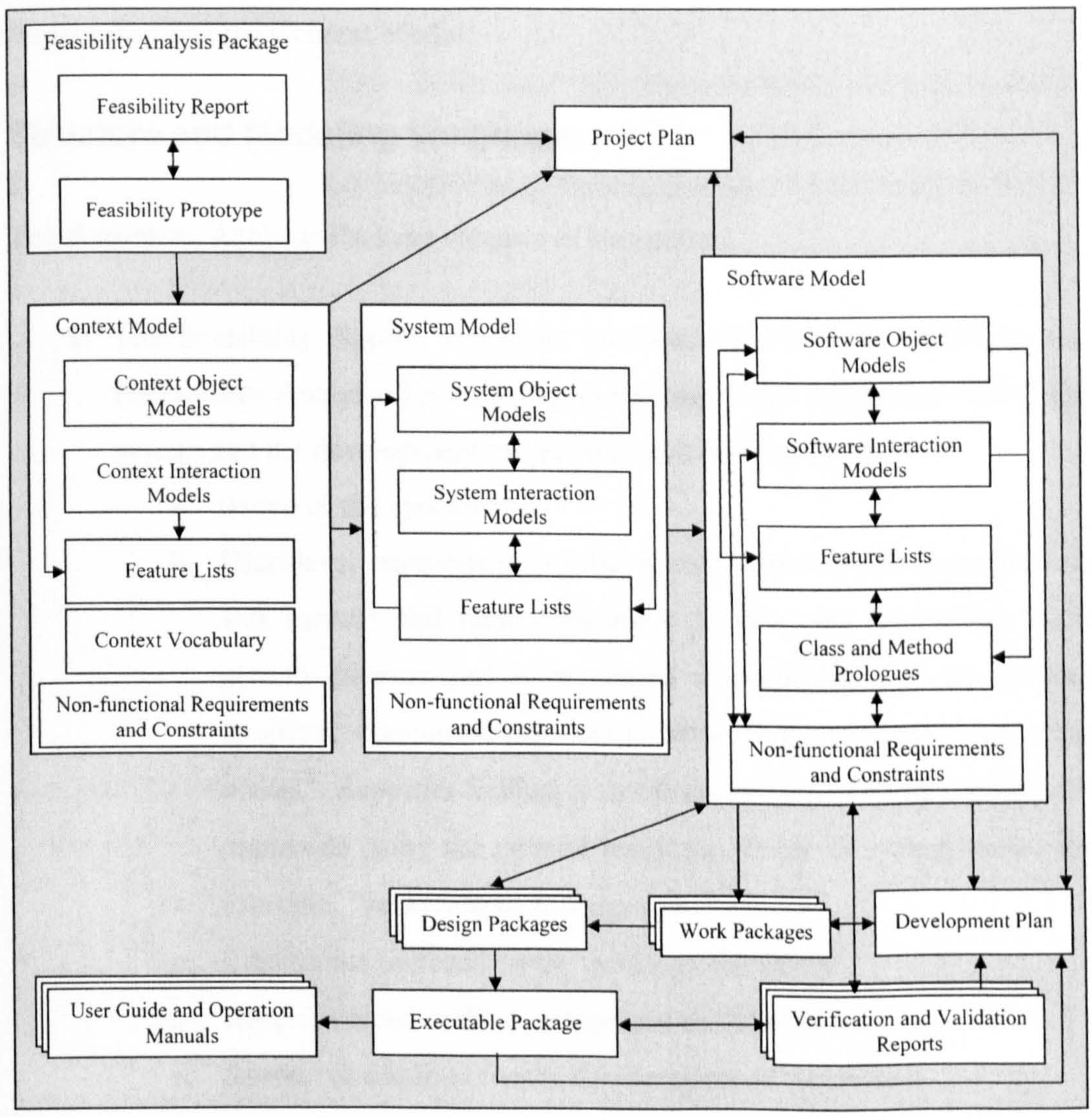

Figure 55. Internal structure of composite models and their interdependencies in the context of the overall work-product structure

The following sections describe the details of each of the work-products, including internal structure and modeling language(s), dependencies, trends of evolution, producers involved, production methods and guidelines, and consistency issues.

5.2.3.1 Feasibility Analysis Package:

The Feasibility Analysis Package is one of the main products produced during the *Preliminary Analysis* subprocess, and contains high-level knowledge about the system and the development project as delineated by the client and explored by the analysts. It lays the groundwork for commencement of the development project and detailed analysis of the problem domain, and is used as a basis for developing the Project Plan and the Context Model.

Structure and Modeling Language

The Feasibility Analysis Package consists of two parts:

1. The Feasibility Report, the exact structure of which is agreed by the Preliminary Analysis Team, encapsulates high-level information about the system and the development project on the following issues:
    a. Scope of the system
    b. High-level requirements of the system, expressed as *major feature sets* (*areas*) and their constituent *feature sets* (*activities*); each *activity* (*feature set*) is expressed as conforming to the general template: *<action><-ing> a(n) <object>*; for example, *"reserving a seat"*. *Activities* belong to *areas* (*major feature sets*), which are expressed using the general template: *<object> management*; for example, *"reservations management"*.
    c. Constraints imposed on the development effort
    d. Risks involved in the development of the system
    e. Resources required for the development of the system
    f. Alternative general approaches to developing the system and results of their feasibility analysis, based on the high-level knowledge so far acquired
    g. Verdict on whether the development of the system should go ahead; considering the scope and requirements of the system, the

constraints imposed, the risks involved, the resources available, and the alternative development approaches so far approved as feasible

2. The Feasibility Prototype: a throw-away prototype used to demonstrate the scope and the technical feasibility of the project; the prototype specifically addresses key requirements, critical technical risks, and alternative architectures and development approaches.

Dependencies

The following dependencies exist between this work-product and other artefacts produced during the enactment of the methodology:

1. The constituent parts of the package (Feasibility Analysis Report and Feasibility Prototype) are mutually dependent.
2. The knowledge about the system and the development project compiled in this package is used for developing the initial version of the Project Plan during the *Preliminary Analysis* subprocess.
3. The high-level view of the system and the problem domain portrayed in this package is elaborated and refined during the *Real-World Domain Modeling and Requirements Elicitation* subprocess, ultimately resulting in the Context Model.

Trend of Evolution

Creation: The Feasibility Analysis Package is created in the Preliminary Analysis subprocess.

Usage Span: The usage span encompasses the *Real-World Domain Modeling and Requirements Elicitation* subprocess, as well as the subprocess where it is created.

Update and Revision: This work product is not updated in subprocesses other than where it is created. Revision is not performed unless a reiteration of the *Preliminary Analysis* subprocess is carried out, typically as a result of circumstances indicating a critical flaw in the knowledge compiled in the product.

Producers Involved

The Preliminary Analysis Team produces and, if necessary, revises the Feasibility Analysis Package. The composition of the team and the responsibilities of the team-members, as pertaining to the production of the Feasibility Analysis Package, are as follows:

- The Project Manager responsible for leading the team, providing and managing resources, facilitating operations, resolving issues with the client and third parties, and enforcing standards and schedules.
- One or more Domain Experts helping the team gain better understanding about the problem domain, the scope and high-level requirements of the system, and the complexities and risks involved; they also provide expert opinion on financial, operational and political feasibility of alternatives, and assess the prototype of the system produced as part of the Feasibility Analysis Package.
- One or more Ambassador Users providing realistic and hands-on knowledge about the system; they also provide user feedback on the prototype, as well as operational and political feasibility of alternatives.
- One or more Chief Programmers developing the prototype of the system and providing expert opinion on issues pertaining to software development; these issues include development risks and constraints, resources required, and technical and schedule feasibility of alternatives. The Chief Programmers may also commission other programmers to help in the development of the Feasibility Prototype.
- The Client Representative responsible for defining constraints and high-level non-functional requirements, ratifying the development approach to be taken, and making the ultimate decision on whether the project should be commenced or aborted.

Production Methods and Guidelines

Methods and guidelines for producing feasibility analysis reports and prototypes are relatively well-established, yet the following are suggested as noteworthy guidelines:

- Since the constituent parts of the package are mutually dependent, a parallel scheme for producing them should be agreed and implemented by the team. An iterative-incremental approach is preferable when significant risks are involved. In this approach, feasibility analysis is conducted in each of the iterations, and if further prototyping is deemed necessary by the team, functionalities to be implemented in the prototype are identified and prioritized according to their development risk, after which high-priority functionalities are selected and implemented in the prototype, and the new version of the prototype is analyzed and assessed according to user feedback. The results are then fed back into the iterative process to be used in a new round of feasibility analysis.
- User involvement is essential if an accurate picture of the system and its operational feasibility is to be obtained. The Project Manager should encourage and facilitate the involvement of Ambassador Users in the activities.

Consistency Issues

The following consistency rules should be observed when making changes to the Feasibility Analysis Package:

- If changes are made to system and project parameters (primarily scope, requirements, resources, and constraints) that may change subprocess/task execution times, alter subprocess/task interdependencies or priorities, or require changes to resources or resource-allocation schemes, appropriate adjustments should be made to the Project Plan in order to maintain consistency and keep the plans up-to-date.
- Any changes to system parameters (primarily scope and requirements), should be duly propagated to the Context Model.

5.2.3.2 Project Plan

The Project Plan is initially created in the *Preliminary Analysis* subprocess, and is reviewed, revised and refined at the end of each subsequent subprocess based on the progress of the project and any circumstances requiring a change in the plans. It

contains subprocess-level and task-level planning and scheduling information, governing the management and control of the project.

The Project Plan depends on the information captured about the project, the problem domain and the system, and as the development progresses, analysis, design and implementation results affect the plan as better estimation of development times and resources becomes possible. Hence, whereas it is the information captured in the Feasibility Analysis Package that is used for producing the initial version of the Project Plan in the *Preliminary Analysis* subprocess, the plan is also affected by the Context Model and the Software Model during later subprocesses. The Project Plan itself is used as a basis for producing the Development Plan, which governs the iterative-incremental development engine; at the end of each iteration, the Development Plan is reviewed and – if necessary – revised, which may in turn necessitate changes to the Project Plan.

Structure and Modeling Language

The Project Plan's exact structure is decided by the teams working during the project, yet it should include projected completion dates for tasks and subprocesses, the resources required and resource-allocation schemes, subprocess/task interdependencies and priorities, and project tracking features such as progress indicators.

The initial version of the plan produced during *Preliminary Analysis* contains project-level and subprocess-level scheduling, resource allocation, and interdependency information. As analysis progresses, task-level detail and tracking features are added to the plan.

Dependencies

The following dependencies exist between this work-product and other artefacts produced during the enactment of the methodology:

1. The Project Plan is created during the *Preliminary Analysis* subprocess based on the knowledge about the system and development project compiled in the Feasibility Analysis Package.

2. The planning and scheduling information captured in the Project Plan is used as a basis for producing the Development Plan during the *Plan by Feature* subprocess. The Development Plan governs the iterative-incremental development engine, and if altered during the iterations of the development engine, might necessitate modifications to the Project Plan.

Trend of Evolution

Creation: The Project Plan is created in the *Preliminary Analysis* subprocess, initially consisting of project-level and subprocess-level scheduling, resource allocation, and interdependency information.

Usage Span: The usage span encompasses all the subprocesses in the lifecycle.

Update and Revision: This work product is reviewed and – if necessary – revised in all subprocesses, since it is used by the Project Manager as a dynamic project tracking and control tool, and any issues causing changes in the progress of the project should be handled with the results duly reflected in the plan. Furthermore, the development and evolution of major work-products also affects the Project Plan, since it enables better estimation of development time and resources required:

- The detailed knowledge – about the problem-domain and the system – captured in the Context Model is used for adding detail and refining the Project Plan at the end of the *Real-World Domain Modeling and Requirements Elicitation* subprocess.
- The detailed system specifications captured in the Software Model are used for refining the Project Plan at the end of the *System Specification* subprocess.
- The architectural design detail added to the Software Model is used for refining the Project Plan at the end of the *Architectural Design* subprocess.
- The Development Plan is used for refining and updating the Project Plan at the end of the *Plan by Feature* subprocess.
- Modifications made to the Development Plan during the execution of the iterative-incremental development engine are reflected in the Project Plan at the end of each iteration.

Producers Involved

The Preliminary Analysis Team produces the Project Plan. The composition of the team and the responsibilities of the team-members, as pertaining to the production of the Project Plan, are as follows:

- The Project Manager responsible for leading the team, providing and managing resources, facilitating operations, resolving issues with the client and third parties, and enforcing standards and schedules. The Project Manager is the principal producer of the Project Plan, using the information provided by other team members and the lessons learned from the development of the Feasibility Prototype for estimating development times, determining subprocess/task interdependencies and priorities, devising resource-allocation schemes, and scheduling tasks. As the main user of the Project Plan as a monitoring and control tool, the Project Manager is also responsible for refining, updating and maintaining the Project Plan during later subprocesses of the lifecycle.
- One or more Domain Experts helping the team gain better understanding about the problem domain, the scope and high-level requirements of the system, and the complexities and risks involved, all of which are used in estimating development time and determining the resources needed for carrying out the project.
- One or more Ambassador Users providing realistic and hands-on knowledge about the system, which complements the information provided by Domain Experts.
- One or more Chief Programmers providing expert technical opinion on issues pertaining to software development; these issues include the development risks and constraints involved, as well as the time and resources required, and are therefore indispensable in the estimation and scheduling activities performed by the Project Manager when developing the Project Plan.
- The Client Representative responsible for defining constraints and high-level non-functional requirements, and ratifying the initial version of the Project Plan.

Production Methods and Guidelines

Project planning is a well-established practice, yet the following can be mentioned as useful guidelines for performing the activity in the context of the overall methodology:

- The Project Plan is maintained by the Project Manager, who is also its principal user, yet it should also be visible to all participants in the development effort. Reflection on the accuracy and rationality of the plan is an ongoing process throughout the project (typically resulting in modifications made to the plan at the end of every subprocess), and relies heavily on feedback obtained from development teams.
- The Project Plan should be meticulously updated with progress tracking data and revised task completion dates during the execution of the iterative-incremental development engine. This requires close cooperation between the Project Manager and the Development Manager.

Consistency Issues

The following consistency rule should be observed when making changes to the Project Plan:

- Any changes made to the Project Plan during the iterative-incremental development subprocesses (typically as pertaining to task schedules) should be reflected in the Development Plan.

5.2.3.3 Context Model

The Context Model is created in the *Real-World Domain Modeling and Requirements Elicitation* subprocess, and is used for producing the System Model in the *System Specification* subprocess. It captures the structure and the dynamic behaviour of the elements of the problem domain – as encountered in reality – in object-oriented models. The system is later introduced in the model as an element of the problem domain, and the functionalities associated with it in the problem domain are defined as the functional requirements of the system.

The Context Model relies heavily on the information already captured in the Feasibility Analysis Package, in that the scope of the system and the high-level requirements captured in the Feasibility Analysis Report form the basis for the context modeling activity.

Structure and Modeling Language

The Context Model consists of the following parts:

1. Context Object Models: consisting of diagrams with a notation similar to UML collaboration diagrams [OMG 2003], but with links adorned with data/control flows (no sequence indicators), in which actors and storage elements of the problem domain (e.g. human workers, systems and data-stores in a business system) are modeled as collaborating objects. While based on regarding the problem domain as consisting of objects, the data/control-flow-oriented approach of the modeling provides a closer correspondence with the problem domain and the workflow-oriented view that Domain Experts tend to have of the problem domain, and hence facilitates real-world modeling (Figure 56, page 314). The system is later added and modeled as a problem domain object (Figure 57, page 314). Subsystems and organizational boundaries are preserved, modeled through using packages and/or separate component diagrams complementing the collaboration diagrams.
2. Context Interaction Models: consisting of UML activity diagrams with swimlanes depicting the participating objects (Figure 58, page 315) and/or sequence diagrams (Figure 62, page 319) which depict typical scenarios of interaction among problem-domain objects [OMG 2003]. With the introduction of the system, models are produced depicting typical system usage scenarios (Figure 60, page 317).
3. Feature Lists: Encompassing Job descriptions and functionalities of domain objects (and organizational units and subsystems) expressed as *areas* (major feature sets) and their constituent *activities* (feature-sets), and – where needed – the finer-grained *features* of each *activity*. With the introduction of the system, system features are also identified and added to

the lists, comprising the functional requirements of the system (Table 3, page 320).

4. Context Vocabulary: A glossary of terms from the problem domain (Figure 63, page 321).
5. Non-functional requirements and constraints.

Dependencies

The following dependencies exist between this work-product and other artefacts produced during the enactment of the methodology:

1. The Context Model is created based on the knowledge compiled in the Feasibility Analysis Package about the problem domain and the system.
2. The information captured in the Context Model about the problem domain and the system is used as a basis for producing the System Model during the *System Specification* subprocess. The System Model is in fact produced through extending and refining the Context Model.
3. The detailed knowledge acquired about the system is used for revising the Project Plan at the end of the *Real-World Domain Modeling and Requirements Elicitation* subprocess.

Trend of Evolution

Creation: The Context Model is created in the *Real-World Domain Modeling and Requirements Elicitation* subprocess. Based on the high-level specifications of the system already defined in the Feasibility Analysis Package, real-world-modeling of the problem domain is conducted, and results are structured into the Context Model.

Usage Span: The usage span encompasses the *System Specification* subprocess as well as the subprocess where the work-product is created.

Update and Revision: This work product is not updated in subprocesses other than where it is created. Revision is not performed unless a reiteration of the *Real-World Domain Modeling and Requirements Elicitation* subprocess is carried out, typically

as a result of circumstances indicating a critical flaw in the knowledge compiled in the product.

Producers Involved

The Modeling Team produces the Context Model. The composition of the team and the responsibilities of the team-members, as pertaining to the production of the Context Model, are as follows:

- The Project Manager responsible for leading the team, providing and managing resources, facilitating operations, resolving issues with the client and third parties, and enforcing standards and schedules.
- The Chief Architect responsible for coordinating modeling activities; the Chief Architect plans modeling iterations, determines the scope and abstraction level of the modeling to be performed in each iteration, forms modeling sub-teams (if necessary) and assigns responsibilities to teams and team-members, arranges and facilitates information gathering activities and sessions, coordinates the sub-teams, and verifies and integrates the models produced.
- One or more Domain Experts helping the team gain better understanding about the problem domain.
- One or more Ambassador Users providing realistic and hands-on knowledge about the system, which complements the information provided by Domain Experts.
- One or more Modeling Experts providing advice on object-oriented modeling methods and techniques.
- One or more Chief Programmers providing expert technical opinion on issues pertaining to software development. Involving Chief Programmers early in the modeling process not only familiarizes them with the problem domain and system requirements, but also means that the team can benefit from their expertise for early identification of technical risk issues pertaining to system requirements, thus making it possible to assess the technical feasibility of the requirements before committing to them.

Production Methods and Guidelines

Based on the scope of the system delineated in the Feasibility Analysis Package, iterative real-world domain modeling is performed in order to produce the Context Model. Complemented by the well-established practices of requirements engineering, the following is one possible method for conducting context modeling:

- *Iterative real-world domain modeling*: Conducted by the Modeling Team and coordinated by the Chief Architect, exploration and modeling of the problem domain is performed iteratively and in a top-down fashion, starting from organizational units and subsystems of the problem domain and gradually moving deeper, shifting focus on fine-grained system elements. The following tasks are performed:
    A. *Iteration Planning*: The Chief Architect schedules the iterations and determines the scope and abstraction level of the modeling to be performed in each iteration.
    B. *Iterative Modeling Engine*: The following tasks are performed in each iteration under the supervision of the Chief Architect:
        I. The Chief Architect assigns responsibilities to teams and team-members, forming modeling sub-teams to work on different parts of the problem domain. The sub-teams are briefed on the modeling scope and granularity intended in the iteration.
        II. Sub-teams conduct information gathering, domain exploration and modeling. System observation and JAD sessions are particularly useful in this context. Domain Experts, Ambassador Users and Modeling Experts should be heavily involved in this activity. Structural elements at the subsystem level and the data/control flows are modeled in UML component diagrams and/or via packages in collaboration diagrams. Fine-grained elements (systems, actors and storage elements) and their data/control flows are modeled in non-sequenced data-flow-oriented collaboration diagrams. Typical interaction

scenarios are modeled in UML activity diagrams and/or sequence diagrams. Functionalities and responsibilities of problem domain elements are identified and modeled as feature sets and features (depending on the granularity of the elements). Constraints and business rules are also identified and recorded. A glossary of problem-domain terms is compiled and perfected as modeling progresses. The Chief Architect arranges and facilitates information-gathering activities and sessions, and coordinates the sub-teams.

III. The models produced in the iteration are reviewed, verified and integrated into the Context Model by the Modeling Team, in team sessions closely supervised by the Chief Architect.

C. *Introduction of the target system*: The system is added as an object to the Context Object Models. Feature sets are then redistributed among objects; features and feature sets are thus assigned to the system and feature lists are updated. Typical scenarios of interaction with the system are modeled and Context Interaction Models are updated accordingly. Non-functional requirements, constraints and business rules of the system are specified and added to the Context Model.

Consistency Issues

The following consistency rule should be observed when making changes to the Context Model:

- Any changes to system parameters (primarily scope and requirements), should be duly propagated to the System Model.
- If the changes affect system and project parameters (primarily scope, requirements, resources, and constraints), appropriate adjustments should be made to the Project Plan in order to maintain consistency and keep the schedule up-to-date.

5.2.3.4 System Model

The result of extending and refining the Context Model, the System Model shows the internal constitution of the system and its place in the problem domain. It is modeled as an extension to the problem domain, using the same notions and concepts as those found in the problem domain. In a business system this amounts to designing the system as a new addition to the organization already in place, staffed and provisioned in its own right as a new department or section, rather than a virtual, to-be-computer-based utility. A transition from what is considered *conceptual* or *essential* in OOSDMs to the so-called *specification* is thus delayed in order to keep the models as tangible as possible for as long as possible to both developers and Domain Experts.

Structure and Modeling Language

The System Model consists of the following parts:

1. System Object Models: Mainly consisting of diagrams with a notation similar to UML collaboration diagrams (but with links adorned with data/control flows with no sequence indicators), in which elements belonging to the system or interacting with it (staff of the system, passive objects and their custodians, and relevant elements outside the system including actors) are modeled as collaborating objects (Figure 68, page 327). Subsystems of the system are modeled through using packages and/or separate UML component diagrams.
2. System Interaction Models: Consisting of UML sequence diagrams (Figure 69, page 328) and/or activity diagrams (with swimlanes depicting the participating objects from inside and outside the system), depicting typical scenarios of interaction among system objects and also between system objects and outside actors.
3. Features list: Based on system functionalities identified and depicted in the Context Model, features and feature sets are assigned to intra-system subsystems and objects, and listed in the features list (Table 4, page 330).
4. Revised list of non-functional requirements and constraints.

Dependencies

The following dependencies exist between this work-product and other artefacts produced during the enactment of the methodology:

1. The System Model is created through extending and refining the Context Model.
2. The information captured in the System Model about the problem domain and the system is used as a basis for producing the Software Model during the *System Specification* subprocess. The System Model is in fact converted to the Software Model through applying feature redistribution patterns.

Trend of Evolution

Creation: The System Model is created in the *System Specification* subprocess by extending and refining the Context Model.

Usage Span: The usage span is limited to the *System Specification* subprocess, during the execution of which the System Model is converted to the Software Model.

Update and Revision: This work product is not updated in subprocesses other than where it is created. Revision is not performed unless a reiteration of the *System Specification* subprocess is carried out, typically as a result of circumstances indicating a critical flaw in the knowledge compiled in the product.

Producers Involved

The Model Conversion Team produces the System Model. The composition of the team and the responsibilities of the team-members, as pertaining to the production of the System Model, are as follows:

- The Project Manager responsible for leading the team, providing and managing resources, facilitating operations, resolving issues with the client and third parties, and enforcing standards and schedules.
- The Chief Architect responsible for coordinating modeling activities.

- One or more Domain Experts helping the team gain a better understanding of the problem domain and the element types therein; this will then be used in designing the system as an extension to the problem domain.
- One or more Ambassador Users providing realistic and hands-on knowledge about the system, which complements the information provided by Domain Experts.
- One or more Modeling Experts providing advice on object-oriented modeling methods and techniques.
- One or more Chief Programmers providing expert technical opinion on issues pertaining to software development.

Production Methods and Guidelines

The System Model is produced through extending the Context Model. The different parts of the System Model are produced as described below:

1. System Object Models are produced through the following steps:

   A. Through consultation with Domain Experts and Ambassador Users, the system is designed as a non-computer-based extension to the existing structure, as if a new internal section is added. Early models can be informal sketches (Figure 65, page 324). The internal structure is for the Model Conversion Team to decide, yet a few ground rules should be observed:

      I. Problem domain objects sharing features with the system are moved inside system boundaries or assigned system counterparts if their attachment to the system is partial (elements interacting with the system may have system counterparts).

      II. Passive flowing- or storage elements are modeled as objects and assigned to *custodian* objects which act as proxies; any access to any passive object should be made via the custodian. In business systems, flowing-data custodians are akin to file/document movers, transferring the file or document put in their custody between processing clerks; data-store custodians are akin to file

clerks and archive keepers. It should be noted that there is no limit on the number of staff assigned to the system, so the number of custodians is expected to be high.

III. Additional objects – if needed – should be of the same general types as those seen in the problem domain; e.g. in business systems, these include clerks, managers, archives, etc.

B. Collaboration diagrams with links adorned with data/control flows (without sequence indicators) are produced depicting collaborations among system objects and external objects. Subsystems of the system are modeled through using packages and/or separate component diagrams.

2. System Interaction Models are produced depicting typical interaction scenarios satisfying the system's requirements. Activity diagrams and/or interaction diagrams are produced for each of the system's feature sets.
3. Feature lists are produced through assigning feature sets and features to the subsystems and objects of the system based on the functionality assigned to the system as a whole and the interaction models produced in the previous task.
4. Non-functional requirements and constraints are updated and added to the System Model.

Consistency Issues

The following consistency rule should be observed when making changes to the System Model:

- Any changes should be duly propagated to the Software Model.

5.2.3.5 Software Model

The Software Model depicts the constituent elements of the software system, and is the result of applying feature redistribution patterns to the System Model. Created in the *System Specification* subprocess, the Software Model is the pivotal model during design and implementation of the system, and is therefore continually revised and updated during the remaining subprocesses of the lifecycle.

Structure and Modeling Language

The Software Model consists of the following parts:

1. Software Object/Class Models: Object Models – with a notation similar to UML collaboration diagrams [OMG 2003], except that data/control flow is shown instead of message flow, and sequencing is ignored – depict typical links and data/control flows among system objects (Figure 75, page 335). Object Models are complemented by Class Models (UML class diagrams [OMG 2003]), showing the classes of the system and their relationships.
2. Software Interaction Models: Typical object interactions are modeled in UML interaction diagrams [OMG 2003] (Figure 76, page 336). The messages are also denoted.
3. Class and Method prologues
4. Revised list of features
5. Revised list of non-functional requirements and constraints

Dependencies

The following dependencies exist between this work-product and other artefacts produced during the enactment of the methodology:

1. The Software Model is created through extending and refining the System Model.
2. The detailed knowledge acquired about the system is used for revising the Project Plan at the end of the *Software Specification* and *Architectural Design* subprocesses, and for producing the Development Plan in the *Plan by Feature* subprocess.
3. The information captured in the Software Model about the system and its requirements is used as a basis for selecting features for development – organized as Work Packages – during iterations of the *Design by Feature* subprocess.
4. The structural/behavioural information and requirements of the system specified in the Software Model are used as a basis for producing Design Packages during iterations of the *Design by Feature* subprocess. The

Software Model is then updated with the detailed design features of the system captured in the Design Packages.

5. The requirements and functional specifications of the system defined in the Software Model are used as a basis for performing verification and validation of the system during the *Build by Feature* and *Transition* subprocesses, resulting in the production of the Verification and Validation (Test) Reports. Corrections and improvements are then made to the Software Model, if necessitated by the test results.

Trend of Evolution

Creation: The Software Model is created in the *System Specification* subprocess by applying feature redistribution patterns to the System Model.

Usage Span: Being the pivotal model in the design and implementation of the system, the usage span of the Software Model encompasses the *System Specification*, *Architectural Design*, *Plan by Feature*, *Design by Feature*, *Build by Feature*, and *Transition* Subprocesses; i.e. over the entire remaining subprocesses of the lifecycle.

Update and Revision: This work product is continually updated and revised through the lifecycle. It is augmented with architectural design details during the *Architectural Design* subprocess, enriched with detailed design features during iterations of the *Design by Feature* subprocess, and refined and improved as a result of corrections found necessary through verification and validation of the executable system in the *Build by Feature* and *Transition* subprocesses.

Producers Involved

The Model Conversion Team produces the Software Model. The composition of the team and the responsibilities of the team-members, as pertaining to the production of the Software Model, are as follows:

- The Project Manager responsible for leading the team, providing and managing resources, facilitating operations, resolving issues with the client and third parties, and enforcing standards and schedules.

- The Chief Architect responsible for coordinating modeling activities. The Chief Architect is put in charge of maintaining the Software Model during the rest of the lifecycle.
- One or more Domain Experts helping the team gain better understanding about the problem domain.
- One or more Ambassador Users providing realistic and hands-on knowledge about the system, which complements the information provided by Domain Experts.
- One or more Chief Programmers providing expert technical opinion on issues pertaining to software development.
- One or more Patterns Advisors providing expertise on patterns and their application for redistributing functionality among system elements

Production Methods and Guidelines

The Software Model is produced through iterative application of feature redistribution patterns to the System Model. The model thus produced will be refined and extended during later design and implementation subprocesses. The different parts of the Software Model are produced as described below:

1. Software Object/Class Models: Patterns are iteratively applied to System Object Models and System Interaction Models to redistribute features among objects in order to enhance encapsulation, increase cohesion and reduce coupling, and also to introduce architecture. During earlier iterations, Reengineering patterns [Demeyer et al. 2003] are applied to redistribute responsibilities among objects. These patterns typically include:
   A. Moving behaviour close to data
   B. Eliminating navigation
   C. Splitting up God classes (Blobs)

   Refactoring patterns [Fowler 1999] are applied in conjunction with the above (indeed, some of them already are a part of the above patterns) and also in later iterations. Antipatterns [Brown et al. 1998] can also be of use in conjunction with refactoring patterns, especially the Poltergeist (for identifying redundant objects) and the Swiss-Army-Knife (for breaking up

overly complex classes). Design patterns [Gamma et al. 1995, Buschmann et al. 1996] are used in later iterations to help implement specific architectures and mechanisms typically present in the problem domain and tangible to users.

Applying the patterns (Figure 71, page 331) prunes the models of classes irrelevant to the system and actor-counterparts without any justification for existence in the system, and relationships not belonging to the system (Figure 75, page 335). Applying the patterns ultimately results in custodian objects being merged with the data objects they had under custody. Class Models (UML class diagrams) are then produced based on the Object Models, depicting the classes in the system and their relationships. Inheritance hierarchies are introduced in order to enhance abstraction

2. Behavioural models (UML sequence diagrams or activity diagrams) are updated in each iteration of the redistribution procedure, and Software Interaction Models are produced (Figure 76, page 336).
3. Initial versions of class and method prologues are prepared.
4. Feature sets and features are refined and compiled in feature lists.
5. Non-functional requirements and constraints are revised.

Consistency Issues

The following consistency rules should be observed when making changes to the Software Model:

- If anytime during the lifecycle changes are made to the Software Model that affect system and project parameters (primarily scope, requirements, resources, and constraints), appropriate adjustments should be made to the Project Plan in order to maintain consistency and keep the schedule up-to-date.
- If changes made during the execution of the iterative-incremental development subprocesses (Design by feature and Build by Feature) affect system and project parameters (primarily scope, requirements, resources, and constraints), appropriate adjustments should be made to the Development Plan in order to maintain consistency and keep the development schedule up-to-date.

5.2.4 Role-Centred Description of the Methodology

The role-centred description of the methodology focuses on the roles involved in the subprocesses of the lifecycle, and how they cooperate in teams in order to perform their tasks.

5.2.4.1 Roles: Responsibilities throughout the Process

The roles involved in the methodology are as listed below:

1. Project Manager: active in all subprocesses, directs and manages the development effort, facilitating development, enforcing the schedule and conformance to standards, managing resources, and resolving issues with the client and third parties.
2. Client Representative: mainly active in the first and last subprocesses, provides decision and feedback on behalf of the client.
3. Domain Expert: mainly active in the first two subprocesses, provides information on the problem domain, helping clarify complexities and risks associated with the problem domain.
4. Ambassador User: active in nearly all subprocesses, provides continuous feedback on the models and the system itself, thus enabling continuous validation to be exercised.
5. Chief Architect: active in all activities where modeling is performed or models are revised, coordinates the modeling activities and maintains the Software Model.
6. Modeling Expert: active in all activities where modeling is performed, provides modeling advice to the teams.
7. Patterns Advisor: mainly active when using patterns to map the System Model to the Software Model and during design-related activities, provides.expertise on the use of patterns.
8. Development Manager: active during design and implementation activities, supervises the development teams and manages the day-to-day resourcing required to keep the project on track.
9. Chief Programmer: active during all activities, providing team supervision and development expertise.

10. Class Owner: programmer active during detailed design and implementation activities and put in charge of implementing, testing and maintaining specific software classes.

5.2.4.2 Teams: Constitution and Responsibilities

The teams undertaking the execution of subprocesses and activities in the course of the methodology are as follows:

1. Preliminary Analysis Team: The Preliminary Analysis Team obtains high-level information on the project and analyses the feasibility of undertaking the development effort during the first subprocess of the lifecycle. It typically consists of:
    a. The Project Manager in charge of the development effort
    b. The Client Representative who makes the decisions on behalf of the client
    c. One or more Ambassador Users providing user feedback
    d. One or more Domain Experts to help understand the complexities of the problem domain
    e. One or more Chief Programmers providing prototyping skills and technical counsel
2. Modeling Team: The Modeling Team performs real-world domain modeling during analysis. It typically consists of:
    a. The Project Manager as supervisor and facilitator
    b. The Chief Architect as the modeling coordinator
    c. One or more Ambassador Users providing user feedback
    d. One or more Domain Experts helping the team to better understand the problem domain
    e. One or more Modeling Experts providing advice on object-oriented modeling issues
    f. One or more Chief Programmers providing practical development counsel
3. Model Conversion Team: The Model Conversion Team produces the System Model and converts it into the Software Model using pattern-based

techniques during the System Specification subprocess. It typically consists of:

a. The Project Manager as supervisor and facilitator
b. The Chief Architect as the modeling coordinator
c. One or more Ambassador Users providing user feedback
d. One or more Domain Experts providing knowledge on the problem-domain
e. One or more Modeling Experts providing advice on object-oriented modeling issues
f. One or more Chief Programmers supplying development-related advice
g. One or more Patterns Advisors providing advice on patterns of feature redistribution and architectural design

4. Architectural Design Team: Active during the Architectural Design subprocess, the Architectural Design Team identifies an implementation-specific architecture for the system modeled so far, and determines the domain-independent infrastructure supporting the system. It typically consists of:

a. The Project Manager as supervisor and facilitator
b. The Chief Architect as the modeling coordinator
c. One or more Ambassador Users providing user feedback
d. One or more Design Experts providing knowledge on architectural design techniques and domain-independent technologies
e. One or more Modeling Experts
f. One or more Chief Programmers supplying development-related advice
g. One or more Patterns Advisors providing advice on patterns of feature redistribution and architectural design

5. Planning Team: The Planning Team plans the iterative-incremental implementation of the features and is active during the Plan-by-Feature subprocess. It typically consists of:

a. The Project Manager as the leader of the team
b. The Development Manager as the resource manager and coordinator of the Features Teams

c. The Chief Programmers involved in the development providing practical implementation expertise crucial to reliable estimation and scheduling of the feature development subprocesses

6. Features Team: Features Teams are collectively coordinated by the Development Manager, and are active during the iterative-incremental development subprocesses. A Features Team typically consists of:
   a. A Chief Programmer as the leader of the team supervising the design-implementation-test activities of the development engine
   b. One or more Modeling Experts helping with the design
   c. One or more Ambassador Users providing user feedback on user-centred aspects of the design and the implemented system
   d. A number of Class Owners undertaking the implementation and testing of the software classes
7. Transition Team: The Transition Team is active during the last subprocess of the lifecycle and is responsible for deploying the implemented system into the user environment. It typically consists of:
   a. The Project Manager as the leader of the team
   b. The Development Manager as coordinator of the Features Teams that will carry out the corrections and alterations to the executable system
   c. A Client Representative deciding whether the project has been successfully concluded
   d. The Chief Programmers acting as leaders of Features Teams
   e. The Class Owners active in the Features Teams
   f. One or more Ambassador Users providing system validation feedback

5.3 Requirements-Based Review of the Implementation

Table 2 shows how each requirement has been addressed in the final implemented version of the methodology, thereby highlighting the requirements that have been met and those that have not been satisfied. It also shows how the implemented methodology has been influenced by existing methodologies and process

patterns/metamodels in addressing the requirements. In this regard, Table 2 complements Table 1 of Chapter 4 (page 233).

5.4 Summary

The blueprint produced in the Design phase of the methodology development lifecycle is refined and detailed during implementation. The end product should be usable by the users (i.e. software developers), and since user guides are the common medium for presenting methodologies in a useable form, the first step in implementing the methodology is to devise a user guide template focusing on the tasks performed, products produced, and producers involved in the methodology. The user guide template describes the methodology from three complementing viewpoints: *Process-Centred*, *Work-Product-Centred*, and *Role-Centred*.

The user template devised has been used for detailing the target OOSDM. The resulting methodology specification (*implementation*) is then fed into the next phase of the iterative Design-Implementation-Test cycle of the methodology development lifecycle: to be verified as functional and validated as conforming to the requirements.

Table 2. Satisfaction of methodology requirements in the implementation phase (continued on next page)

	REQUIREMENT		ADDRESSED/ NOT ADDRESSED	DETAILS	FOLLOW-UP ACTION
Process	Clarity, rationality, accuracy, consistency of definition		Addressed	➢ Addressed through the User Guide template used for defining the methodology (influenced by SPEM, RUP, USDP and FDD).	➢ To be enhanced in the test phase.
	Coverage of generic development lifecycle activities		Addressed	➢ Addressed at the lifecycle, subprocess and task levels (influences as listed in Table 1).	
	Support for umbrella activities	Risk management	Addressed	➢ Addressed through iterative-incremental development, preliminary analysis, risk-based planning, prototyping, continuous verification and validation, regular product/plan reviews, and continuous integration (influences as listed in Table 1).	
		Project management	Addressed	➢ Addressed through project planning, scheduling and control activities incorporated in the subprocesses of the methodology, and provisions for review and revision of the plans throughout the process (influences as listed in Table 1).	
		Quality assurance	Addressed	➢ Addressed through regular technical reviews, continuous verification and validation during iterative development, and requirements traceability (influences as listed in Table 1).	
	Seamlessness and smoothness of transition between phases, stages and activities		Addressed	➢ Addressed through the artefact chain and the iterative-incremental development engine (influences as listed in Table 1).	
	Basis in the requirements		Addressed	➢ Addressed through the feature-driven approach governing all development activities throughout the process (influences as listed in Table 1).	
	Testability and Tangibility of artefacts, and traceability to requirements		Addressed	➢ Addressed via basis in real-world modeling, fractal modeling, gradual seamless transformation of artefacts through analysis and design, and the feature-driven nature of artefacts throughout the process (influences as listed in Table 1).	
	Encouragement of active user involvement		Addressed	➢ Addressed through constant participation of user representatives throughout the process (influences as listed in Table 1).	
	Practicability and practicality		Addressed	➢ Addressed through avoiding complexity at all levels, adhering to risk-based development, incorporating project management activities, and using techniques and strategies for focusing the development (influences as listed in Table 1).	
	Manageability of complexity		Addressed	➢ Addressed through the hierarchical structure of the methodology, and keeping subprocesses, activities and tasks cohesive and easy to understand (influences as listed in Table 1).	

Table 2. Contd.

REQUIREMENT			ADDRESSED/ NOT ADDRESSED	DETAILS	FOLLOW-UP ACTION
Process	Extensibility / Configurability / Flexibility / Scalability		Addressed (Except Configurability and Flexibility)	➢ Scalability was addressed through plan-based, model-driven and architecture-centric process (influences as listed in Table 1). ➢ Extensibility was addressed through keeping the process as a cohesive core organized around a model chain (influenced by Coad-Yourdon, BON and Catalysis).	➢ Configurability and Flexibility not addressed. Implementing Flexibility via adding process review sessions should be explored.
	Application scope (*Information Systems*)		Addressed	➢ Partially addressed through concentrating on business systems as commonly encountered information systems (influences as listed in Table 1). ➢ Applicability to other kinds of information systems has not been explored.	➢ Applicability to information systems other than business systems is now considered beyond the scope of this thesis, and should be explored in further research projects.
Modeling Language	Support for object-oriented modeling	Structural – Functional – Behavioural	Addressed	➢ Addressed through using appropriate UML-based diagrams at different levels (influences as listed in Table 1).	
		Logical to Physical	Addressed	➢ Addressed through the model chain, starting at the problem-domain level and proceeding to detailed design (influences as listed in Table 1).	
		At different levels of granularity	Addressed	➢ Addressed through fractal modeling (influences as listed in Table 1) at different granularity levels (Enterprise level – System level – Subsystem/Package level – Inter-object level – Intra-object level).	
		Formal and Informal features	Not Addressed (Formal)	➢ Informal features implemented through UML. ➢ Formal features not considered, but seem feasible via using OCL.	➢ Implementation of formal features is now considered beyond the scope of this thesis.
	Provision of strategies and techniques for tackling inconsistency and complexity		Addressed	➢ Addressed through detailed specification of dependencies and consistency guidelines (influenced by Catalysis).	

Chapter 6

Test

In software engineering, testing is a process intended to build confidence in the software [Sommerville 2004]. In an iterative-incremental development process such as that prescribed by the meta-methodology applied herein, testing is performed iteratively, not only to build confidence in the end product, but also to guide the development process through focusing development on satisfying the requirements, and to mitigate development risks via early detection of design and implementation flaws.

In this research, testing has been applied as a continual activity to verify and validate the results of the two development phases of Design and Implementation. Two small business systems have been targeted as case studies for verifying the methodology and validating it against the requirements. This chapter contains an explanation of the test process and the results of applying the implemented methodology for the analysis and design of the two systems used as test inputs. As expected, activities directly concerning the model-chain have become the primary focus of testing, and the results clearly reflect this. This was mainly due to the pivotal role of the model chain in the methodology, and the novelty of the pattern-based approach applied in its production.

6.1 Test Process

As in any software testing effort, the test activity prescribed by the meta-methodology applied in this research consists of the four generic activities of designing test-cases, preparing test data, running the software with the test data, and comparing the results to test cases [Sommerville 2004]; the difference is that the software being tested here is a methodology, with the immediate consequence that *development situations* become the test data.

The meta-methodology used for developing the methodology is test-based, in that it prescribes testing as part of an iterative-incremental development *engine*, consisting of Design-Implementation-Test cycles. As explained in Chapter 1, although validation is iteratively performed at the end of the Design and Implementation activities, each Design-Implementation-Test cycle of this engine relies on the Test activity to perform verification and final validation of the system increment that has been developed.

In the context of methodology development, *verification* is performed in order to ensure that the methodology correctly implements its functions; in other words, to ensure that work products are successfully produced, culminating in the production of the target software system. Considering the scope of this research, test cases were mostly focused on areas of higher potential risk.

Validation tests the methodology against the requirements. Requirements-based reviews, conducted after each iteration of the Design and Implementation activities of the development engine, are validation activities and provide risk management and quality assurance, yet they cannot replace validation with test data, performed during the Test activity.

Since the methodology's scope of application is currently limited to information systems – with the main focus on business systems – two business systems have been chosen to act as development test-beds, providing the development situations necessary for testing the methodology. The resulting case studies have taken shape during the iterative-incremental development of the methodology, helping to gradually refine and sculpt the methodology into its final shape by detecting and correcting the flaws and smoothing the rough edges. In order to provide a wider coverage of diverse development situations, several points of difference have been introduced in the definition of the target systems; for example, one system is introduced in a fully manual problem domain, while the other problem domain already contains computer-based elements, and whereas one system is to be implemented and used as a local-access system, the other is web-based. The systems are briefly described below:

- A *Library System* providing basic library services to members and librarians; the existing library is managed through a manual process and depends on index cards for searching for books and keeping track of books, members, loans, returns and reservations. The computer-based system is to replace the index cards and provide additional facilities for searching and transaction management. The selection and definition of this particular problem domain as a case study was inspired by and based on the author's personal familiarity with the domain, gained through commercial and academic development projects.
- An *Estate Agency System* providing property search, property promotion, and transaction management facilities; the existing agency relies on a computer-based record management system used by agency clerks for storing information on properties, customers (buyers/sellers), and transactions. The target system is to provide online facilities to customers for searching properties, putting properties up for sale, requesting viewings, making offers on properties, and communicating with agents and clerks. It should also provide messaging, documentation and information management facilities to clerks, and reporting and communication facilities to agents. The definition of this system is loosely based on a preliminary user-story, later developed into the eXGrid case study [Ge et al. 2006].

Tangibility, simplicity and understandability were the main criteria considered in selecting these systems: however testable the methodology itself is, poor test data in the form of unfamiliar, complex, poorly documented or unexplored problem domains is bound to hamper the testing process. Considering the scope of this research, small-scale versions of these systems have been targeted, and where possible, selected subsets of the overall system functionality have been focused upon in order to avoid unwarranted complexity.

Due to the pivotal role of the model chain and the novel approaches applied to its production, the *Real-world Domain Modeling and Requirements Elicitation* subphase – where modeling starts – and the *System Specification* subphase – where model conversions are performed – have been targeted by the test cases. Other subphases are well-established cohesive activities – coupled together according to well-established development frameworks and metamodels – that have previously

been used in other methodologies [Cockburn 2004, DSDM Consortium 2003, D'Souza and Wills 1998, Palmer and Felsing 2002]. What is being tested is the ability of the methodology to enable successful production of design models that specify the class structure and inter-object behaviour of the software system in such a way that satisfies system requirements. The subsequent introduction of architectural design details into the models, and the ultimate production of class and method prologues leading to the software code, have already been done in other methodologies – the most prominent of which are Catalysis [D'Souza and Wills 1998] and FDD [Palmer and Felsing 2002] – and therefore pose a relatively minor degree of risk. It is true, however, that a comprehensive verification and validation of the methodology requires enactment in an industrial context, which has been suggested in Chapter 7 as a future task for furthering this research, yet verifying the model chain is essential for building an acceptable level of confidence in the methodology, and has indeed been crucial for gradual refinement of the approach, and the methods and techniques applied.

The following sections summarise the verification results, focusing on the work-products produced through applying the methodology for the analysis and design of the two systems mentioned above. Validation results have been tabulated and reported separately at the end of the chapter.

6.2 Case Study 1: Book Library System

The book library problem domain targeted in this case study is a currently manual system providing basic library services to members. Introducing a computer-based system through applying the proposed object-oriented methodology starts with exploring and modeling the problem domain, and progresses to designing the new system first as an extension to the current system and then as a software system. The following sections contain the results of the modeling activities performed on the system through the application of the development methodology. For sake of brevity, when modeling detailed aspects of the system, focus has been limited to the two basic functions of borrowing a book and returning a book.

Continuous verification was mostly performed on this case study, as a result of which the methodology was gradually refined. Some of the more significant results are given below:

- Activity diagrams should be given precedence over sequence diagrams in problem domain modeling, but the priority is reversed during later subprocesses. Verification showed that using activity diagrams facilitates requirements elicitation through highlighting the *features*, and they are better suited for modeling problem domains due to their superior modeling power, especially in modeling parallel work flows. Sequence diagrams, on the other hand, are better suited to specifying dynamic object communications, which come under focus later in the development process.
- Object Models should be kept data-flow-oriented and feature-driven. Verification showed that if data-flows were replaced by message-flows, continuity would be disrupted. This is particularly damaging to traceability and seamlessness. Specification of operations and message flow is therefore left to the System Specification subprocess, where they evolve from features, and are modeled in Software Class- and Interaction Models.
- It is best to apply redistribution patterns in a specific order for transforming the System Model into the Software Model. Verification showed that Reengineering Patterns should be applied first, then Refactoring Patterns, and then Design Patterns (as introduced in Section 5.2.2.3). This ensures that major anomalies are removed before the introduction of new structures.

6.2.1 Context Model

The Context Model components presented in this section include Context Object Models, Context Interaction Models, the Context Features List, and a partial Glossary of Terms. The Context Object Models show a representation of the problem domain as encountered in the real world, with the target system then added as a problem domain object. The Context Interaction Models depict the cooperation among problem domain objects for performing the *book-borrow* and *book-return* processes. Context Interaction models come in two versions: the first versions model the real world, and the latter ones depict the interactions after the target system is added as a problem domain object and is involved in inter-object

cooperation. The Context Features List shows the major feature sets (*areas*), as well as their constituent feature sets (*activities*) and bottom-level features (*steps*), with the detail mostly confined to features pertaining to *book-borrow* and *book-return* processes. The list also shows the assignment of feature sets and features to problem domain objects.

6.2.1.1 Context Object Models

Figure 56 shows the real-world Context Object model of the library problem domain. The diagramming notation resembles that of UML collaboration diagrams [OMG 2003], yet the semantics of the inter-object interactions does not conform to the UML, in that it depicts data flow instead of message/signal flow, and also because sequencing is ignored. The diagram also shows the assignment of feature-sets and features to objects. Objects and data flowing between them are direct models of the real world, yet the modeler can define a specific object to represent a typical object encountered in the problem domain, together with its typical properties and features; Librarian and Library-Member are examples of such objects. These typical objects are not to be called *classes* yet, in order to keep models tangible to Domain Experts and Ambassador Users for as long as possible. Figure 57 shows the Context Object Model after the target system is added as a problem domain object. Feature sets have been redistributed and new feature-sets have been added to the system.

6.2.1.2 Context Interaction Models

Figure 58 and Figure 59 show the scenarios for performing the *book-borrow* and *book-return* processes in the real-world library. UML activity diagrams are used at this stage with swimlanes depicting the active objects participating in the processes.

The use of swimlanes is essential in this context, since it is their use that makes the diagrams object-oriented. As seen in the figures, *Librarian* and *Member* are the two active objects cooperating to perform the *book-borrow* and *book-return* scenarios. Storage objects (such as card-racks) are not modeled as participating objects due to their passive roles in the system, nevertheless references to their usage by active objects can be seen in activity descriptions.

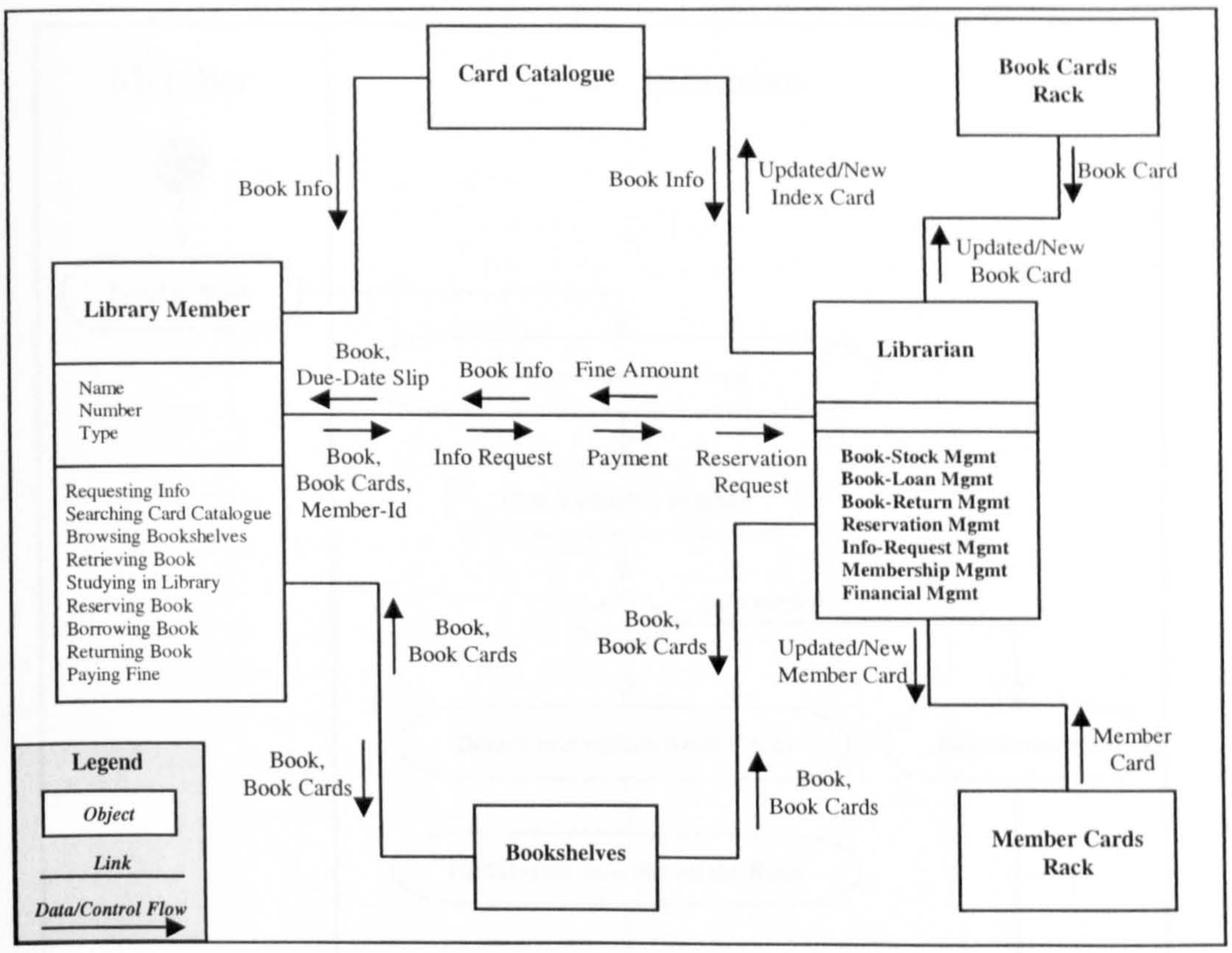

Figure 56. Context Object Model of the Book Library problem domain

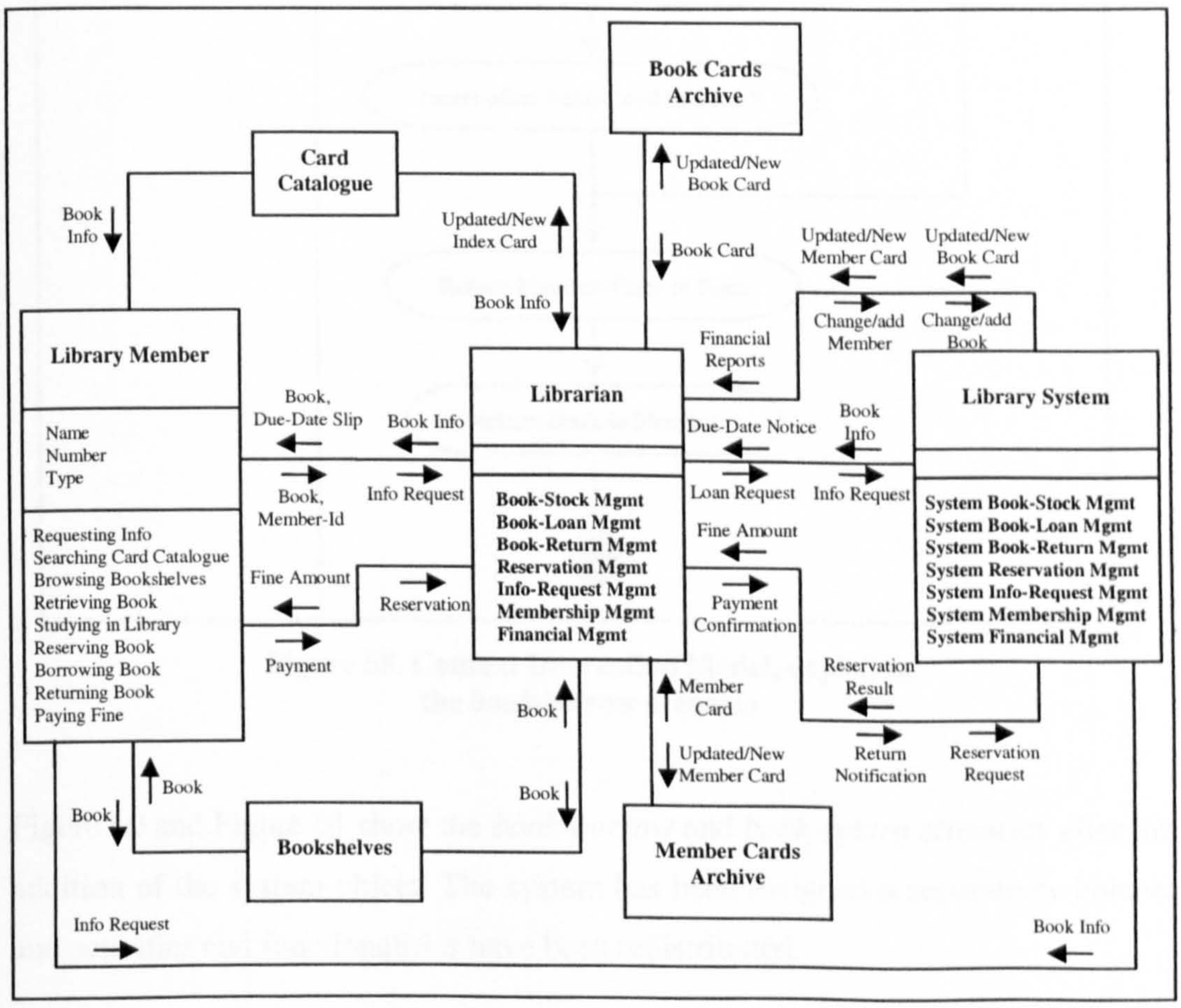

Figure 57. Context Object Model, with the system introduced as a problem domain object

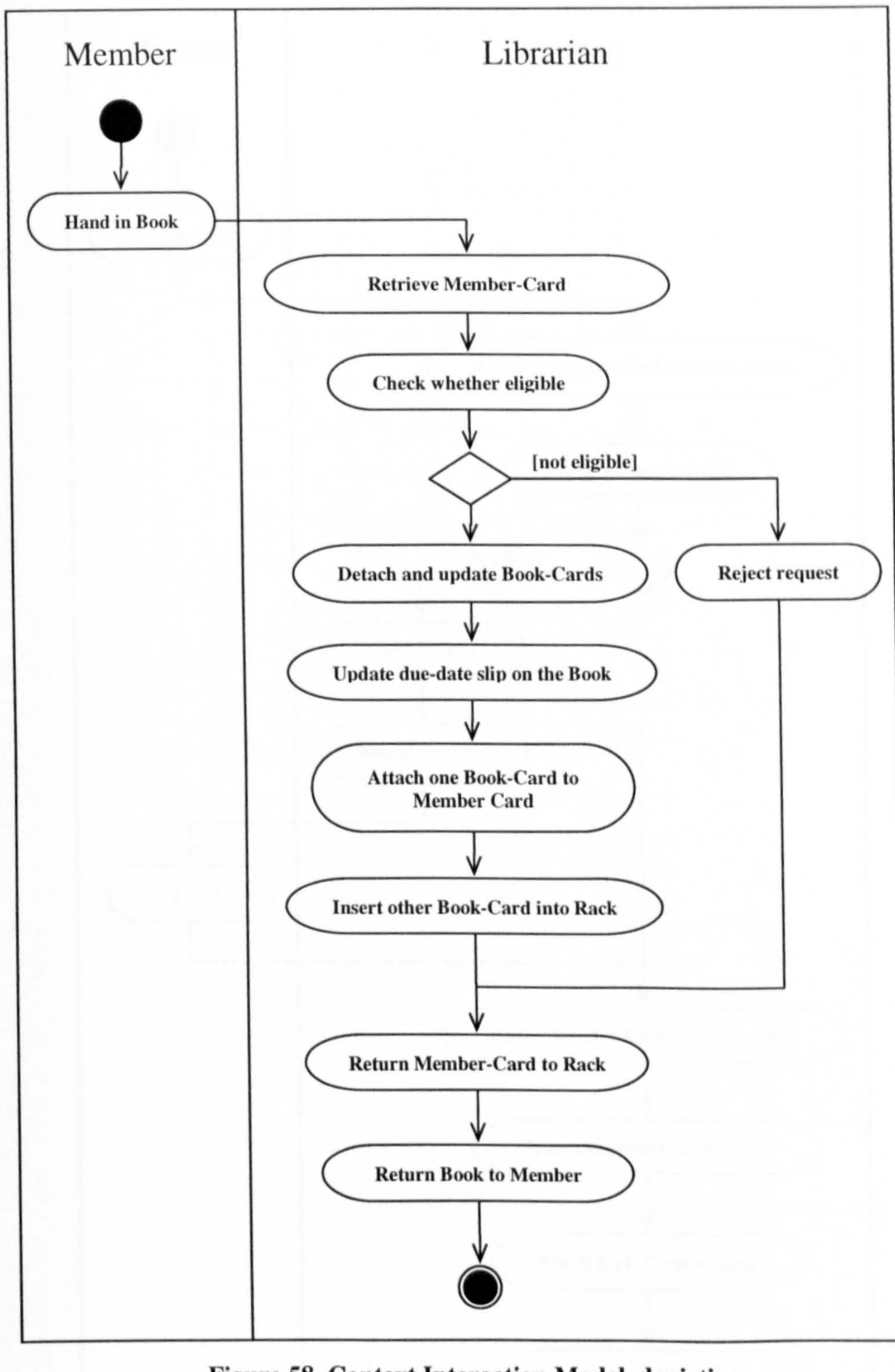

Figure 58. Context Interaction Model, depicting the *book-borrow* scenario

Figure 60 and Figure 61 show the *book-borrow* and *book-return* scenarios after the addition of the system object. The system has been assigned a separate swimlane, and activities and functionalities have been redistributed.

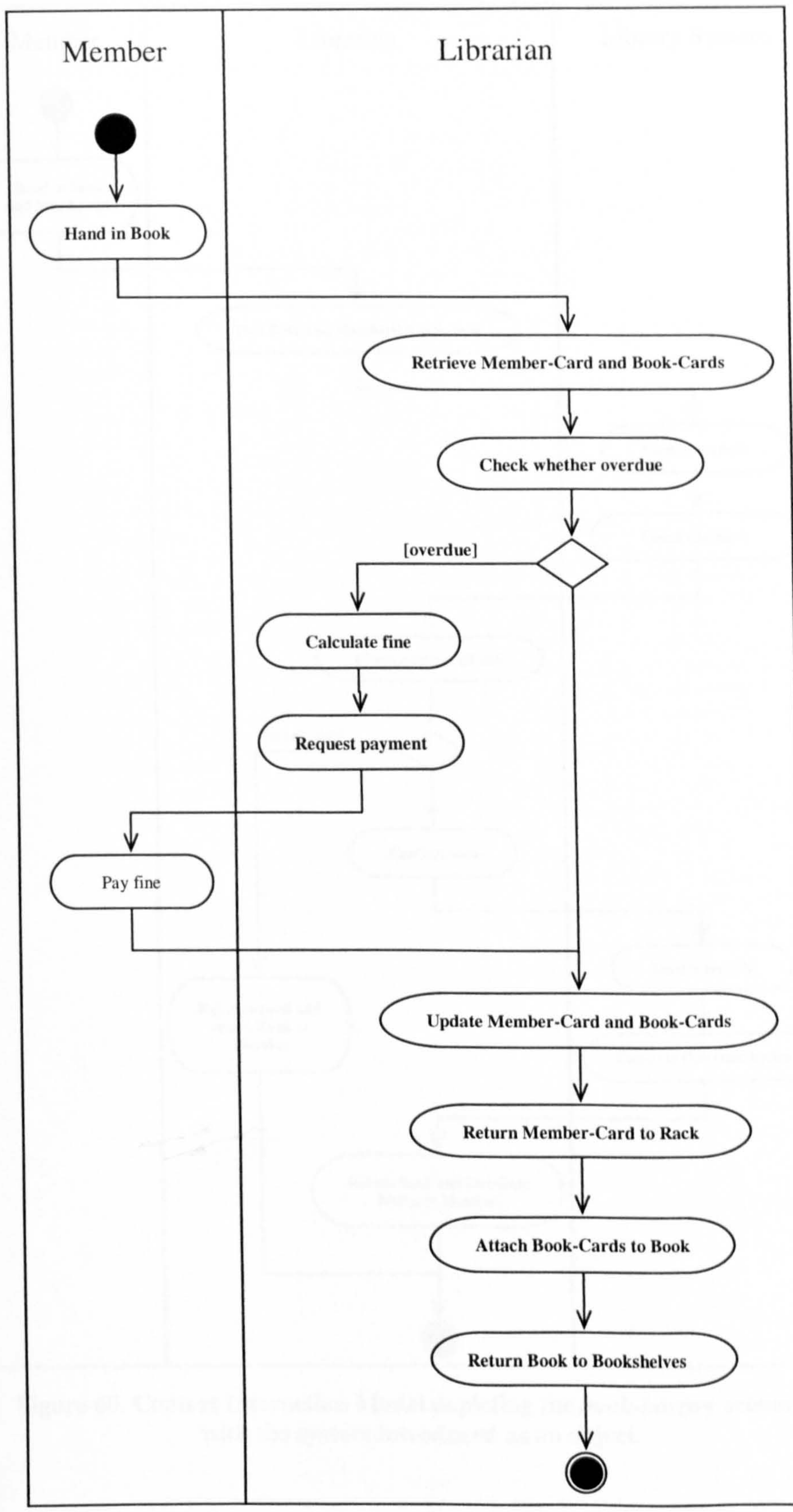

Figure 59. Context Interaction Model, depicting the *book-return* scenario

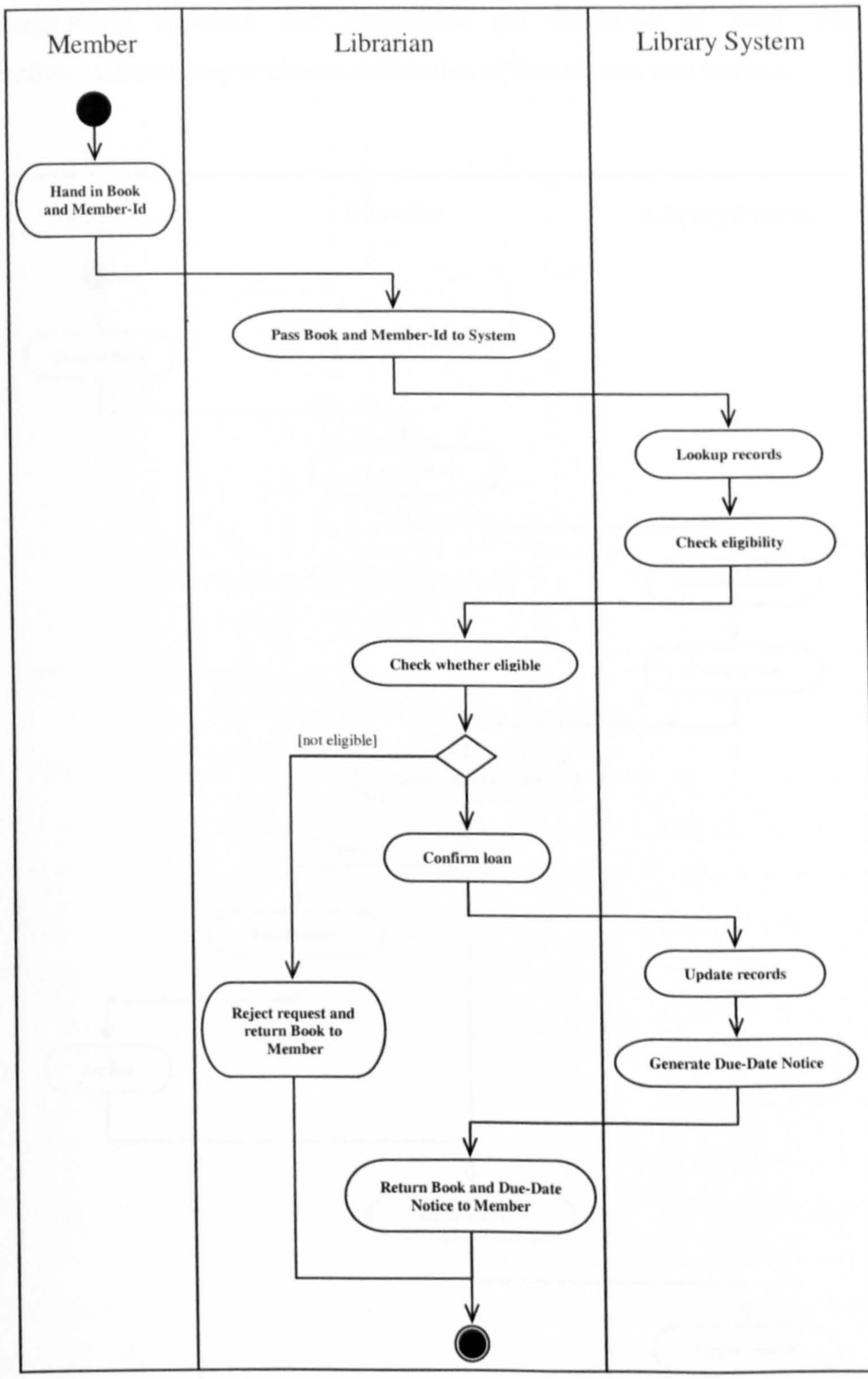

Figure 60. Context Interaction Model depicting the *book-borrow* scenario, with the system introduced as an object

Figure 62 is an alternative Context Interaction Model using UML sequence diagrams instead of activity diagrams. While activity diagrams provide a simpler informal tool for depicting cooperation scenarios, and are therefore quite suitable for the starting stages of the development, sequence diagrams tend to force a

message-based approach and necessitate the definition of more clear-cut operations, thus leading to clearer delineation of feature-sets and features.

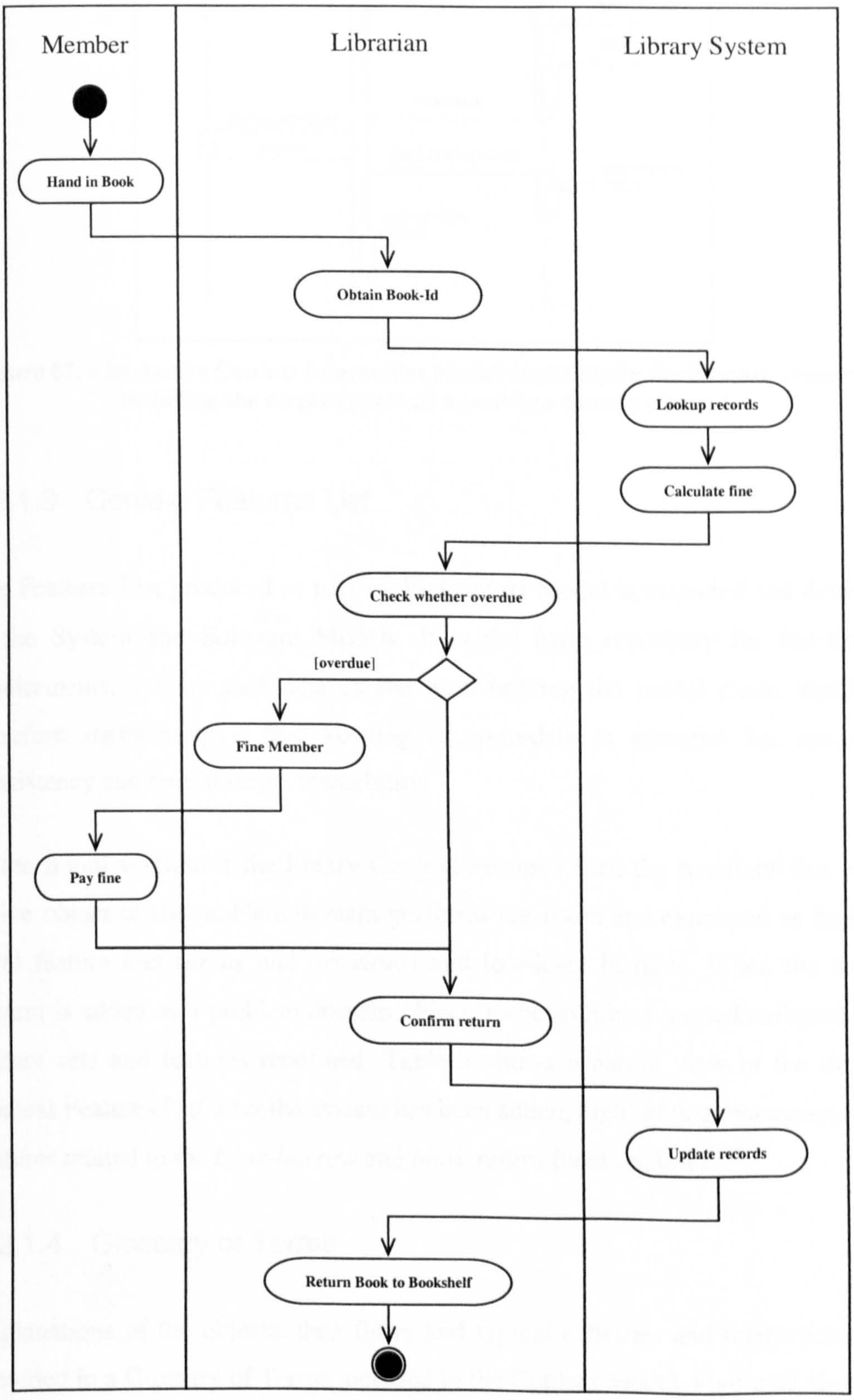

Figure 61. Context Interaction Model depicting the *book-return* scenario, with the system introduced as an object

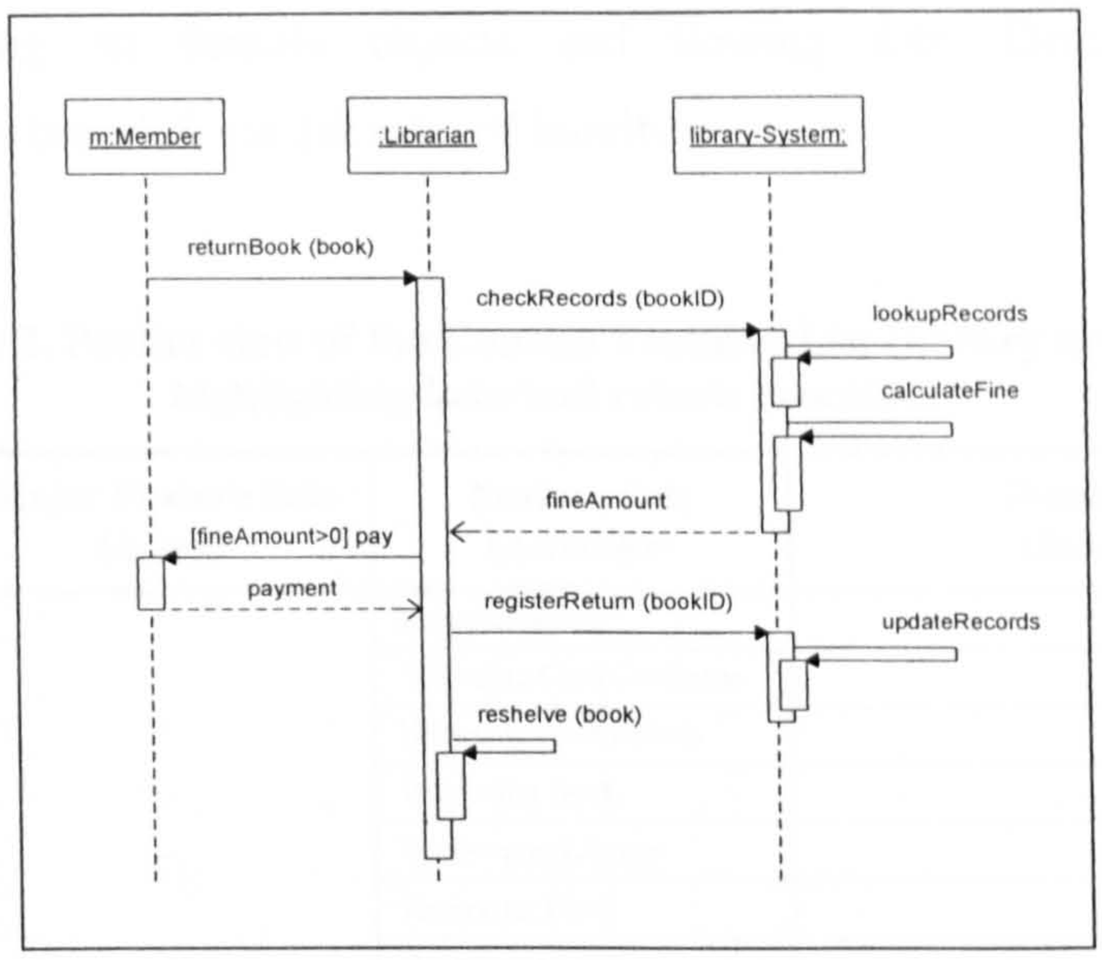

Figure 62. Alternative Context Interaction Model depicting the *book-return* scenario, including the target system as a problem domain element

6.2.1.3 Context Features List

The Features List produced as part of the Context Model is extended and detailed in the System and Software Models. It is the main repository for functional requirements, and as such acts as the base holding the model chain together, therefore maintaining it and keeping it up-to-date is essential for ensuring consistency and requirements traceability.

In the initial version of the library Context Features List, the functions that each active object of the problem domain performs are listed and expressed as higher-level feature sets (*areas* and *activities*) and low-level features. When the target system is added as a problem domain object, responsibilities are redistributed and feature sets and features redefined. Table 3 shows a partial view of the library Context Features List after the system has been added, highlighting feature-sets and features related to the *book-borrow* and *book-return* functionalities.

6.2.1.4 Glossary of Terms

Explanations of the objects, data flows and typical activities and interactions are provided in a Glossary of Terms included in the Context Model. Figure 63 shows a partial view of the Glossary of Terms in the library system's Context Model,

mainly focusing on domain objects and flowing data. Detailed structural information has been left out for sake of brevity.

Table 3. Partial view of the Context Features List (library system), highlighting loan and return processes

Object	Major Feature Sets (*Areas*)	Feature Sets (*Activities*)	Features (*Steps*)
Member	–	Requesting Info	…
		Searching Card Catalogue	…
		Browsing Bookshelves	…
		Retrieving Book	…
		Studying in Library	…
		Reserving Book	…
		Borrowing Book	…
		Returning Book	…
		Paying Fine	…
Librarian	Book-Stock Mgmt	…	…
	Book-Loan Mgmt	Passing loan info to System	…
		Passing results to Member	…
	Book-Return Mgmt	Passing Book to System	…
		Fining Member	…
		Reshelving Book	…
	Reservation Mgmt	…	…
	Info-Request Mgmt	…	…
	Membership Mgmt	…	…
	Financial Mgmt	…	…
System	System Book-Stock Mgmt	…	…
	System Book-Loan Mgmt	Verifying eligibility	Lookup loan specifications of *book*
			Lookup loan permissions of *member*
			Determine eligibility of *member*
		Registering loan	Determine loan duration for *book*
			Update loan history of *book*
			Update loan history of *member*
		Issuing Due-Date Notice	Retrieve specifications of *book*
			Retrieve specifications of *member*
			Generate due-date notice for *book*
	System Book-Return Mgmt	Calculating fine amount	Lookup loan history of *book*
			Lookup loan specifications of *member*
			Calculate fine payable by *member*
		Registering return	Update loan history of *book*
			Update loan history of *member*
	System Reservation Mgmt	…	…
	System Info-Request Mgmt	…	…
	System Membership Mgmt	…	…
	System Financial Mgmt	…	…

Book Library

Problem Domain Objects

Member	A person registered as a member and issued with a membership card. Each Member is assigned a unique membership number and a type, based on which the maximum number of books that can be borrowed by the member at any one time are determined, as well as the maximum loan duration and the fine amount that should be paid by the member if a book is returned after its due date.
Librarian	A person providing library services in the library.
Card Catalogue	A set of *index cards* – indexed according to *title*, *subject* and *author* – providing a searchable facility for obtaining book information.
Book-Cards Rack	A holder for *book cards*, which are sorted in the rack according to book number.
Member-Cards Rack	A holder for *member cards*, which are sorted in the rack according to member number.
Bookshelves	Holders for *books*, which are sorted in the shelves according to book number.

Flowing Data (or Objects)

Book	Each book volume in the library, which is assigned a unique book number and a type, based on which the maximum loan duration and the fine amount (in case of delays in returning the item) are determined. Each book is issued with a number of cards: two *book cards* – for recording loan information such as borrower numbers, due dates and actual return dates –, one *detail card* – which contains detailed information about the book and is permanently held in the *book-cards rack* –, and a number of *index cards* – which are put in the *card catalogue* for searches on author, title and subject. To each book a card sleeve and a *due-date slip* are attached. The sleeve is used for holding the book's two *book cards* when it is on the shelves.
Book Card	Two issued for each book, book cards record loan history and current loan data, especially borrowers' numbers, due dates and actual return dates. When a book is on the shelves, the cards are contained in the book in a special sleeve. When the book is lent, the number of the borrower and the due date are entered in the two cards, one of which is attached to the borrower's *member card* in the *member-cards rack* and the other is attached to the book's *details card* in the *book-cards rack*, thus enabling the librarian to cross-reference the members with their borrowed books. When a book is returned, the book cards are retrieved, updated with the return date and then reinserted in the sleeve on the book.
Member Card	Each *member* is issued with a member card which is permanently held in the *member-cards rack* and contains the member's personal data and loan history, including current loans. For each loan, the book number, due date and return date are recorded.
Index Card	These search cards enable the *member* to obtain *book info* from the *card catalogue* based on search criteria including title, author and subject.
Due-Date Slip	A slip which is attached to each *book* and holds loan history information including borrowers, due dates and return dates. When a book is loaned, the borrower's number and the due date are entered in the due-date slip. The due date is calculated based on the book type and the member type using loan duration tables. When a book is returned, the slip is updated with the return date.
Book Info	Detailed information on a *book*, which results from a search in the *book catalogue* or an *info request* from a *librarian*.
Info Request	A request for *book info* which is submitted to a *librarian* and typically includes values for search criteria – such as title, author or subject.
Reservation Request	A Request submitted to a *librarian* to reserve a specific *book*. Book number is supplied along with the request.
Fine Amount	Calculated by the *librarian* according to fining tables.
Payment	Fine amount paid in by the *member*.

Figure 63. Partial view of the glossary of terms for the library case study

6.2.2 System Model

The System Model components presented in this section include System Object Models, System Interaction Models, and the System Features List.

The System Object Models show the internal structure of the target system designed as an extension to the existing library structure, i.e. as a separate section consisting of library clerks and information storage facilities found in conventional offices. Detail has been limited to *book-borrow* and *book-return* processes.

The System Interaction Models depict the cooperation among system objects for performing the *book-borrow* and *book-return* processes.

The System Features List shows the feature sets (*activities*) and bottom-level features (*steps*) assigned to system objects, with the detail mostly confined to functionality as pertaining to *book-borrow* and *book-return* processes.

6.2.2.1 System Object Models

Since System Model components focus on the two system functionalities of *book-borrow* and *book-return*, a more restricted view of the latest version of the Context Object Model has been presented (Figure 64) as the basis for System Object Models.

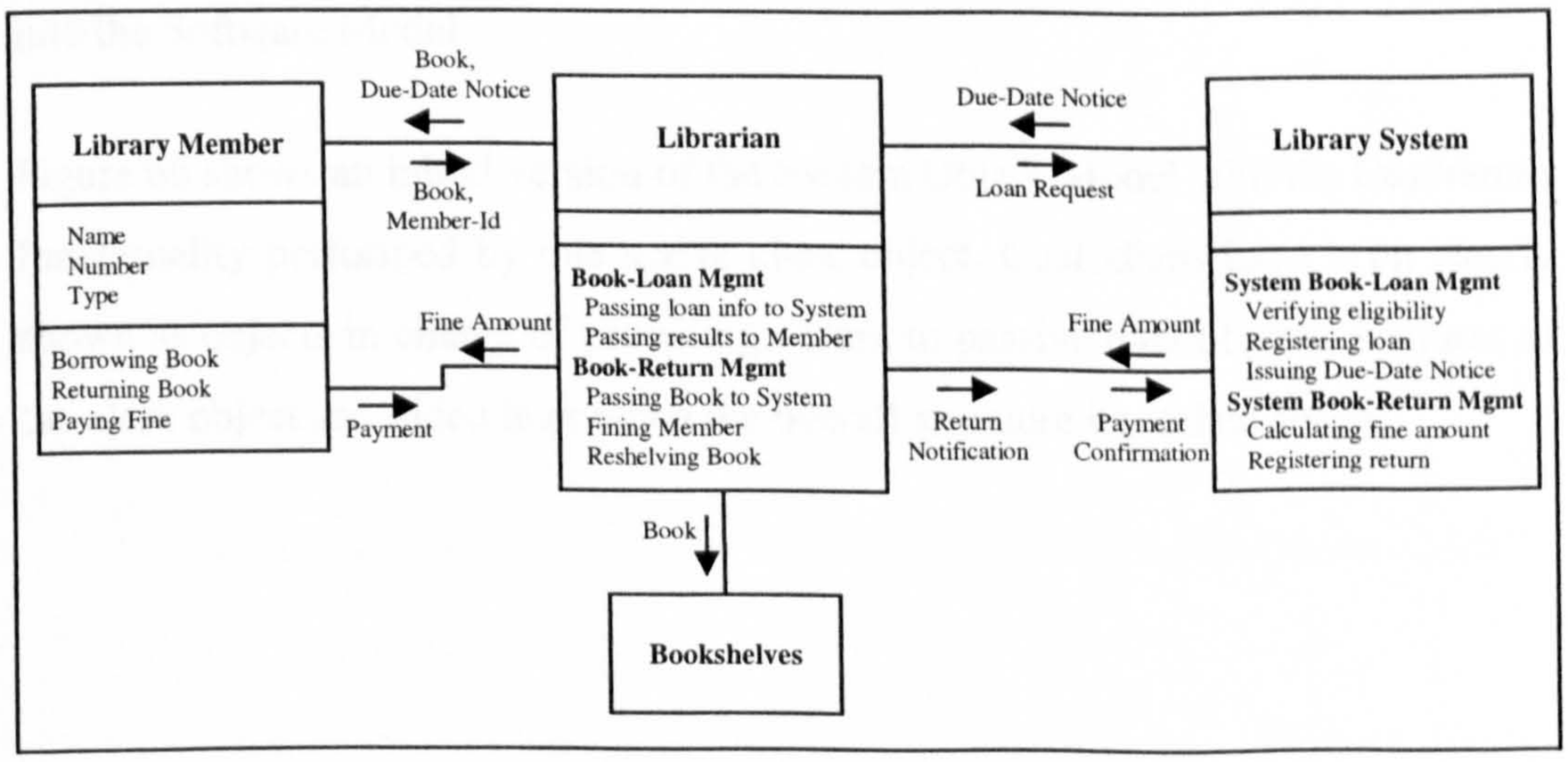

Figure 64. Context Object Model focusing on the *loan* and *return* functionalities of the library system

As already mentioned, the System Model is developed through designing the system as an extension to the existing library. Figure 65 shows an example of one such design.

The shaded area in Figure 65 shows the target system, with the objects colour-coded in order to be easily distinguished. Although not a part of the System Object Model, this type of blueprint can be safely used in order to provide a more realistic visualization of the target system to the Ambassador Users and Domain Experts. However, it will ultimately be represented as System Object Models.

As shown in this diagram, a number of custodians are put in charge of providing access to data stores and archives, and other custodians are on standby to take charge of flowing data when the need arises.

Each and every item retrieved from data storage (e.g. Book Record), produced by clerks, or supplied by entities external to the system, is given to a custodian, who controls access to the item until it either has to be returned to storage, is completely consumed during processing (and is therefore discarded), or is supplied to external entities. This ensures that passive data is always coupled with and encapsulated by active objects.

Custodians are tangible to Domain Experts and Ambassador Users, as they correspond to entities normally seen in business systems. Furthermore, they lay the groundwork for applying redistribution patterns for transforming the System Model into the Software Model.

Figure 66 shows an initial version of the System Object Model with the loan/return functionality performed by one active clerk object. Custodians have been clearly shown as objects in charge of providing access to passive data objects. Features of the clerk object are added later when the overall structure has taken shape.

Figure 65. Designing the library system as an extension to the existing structure

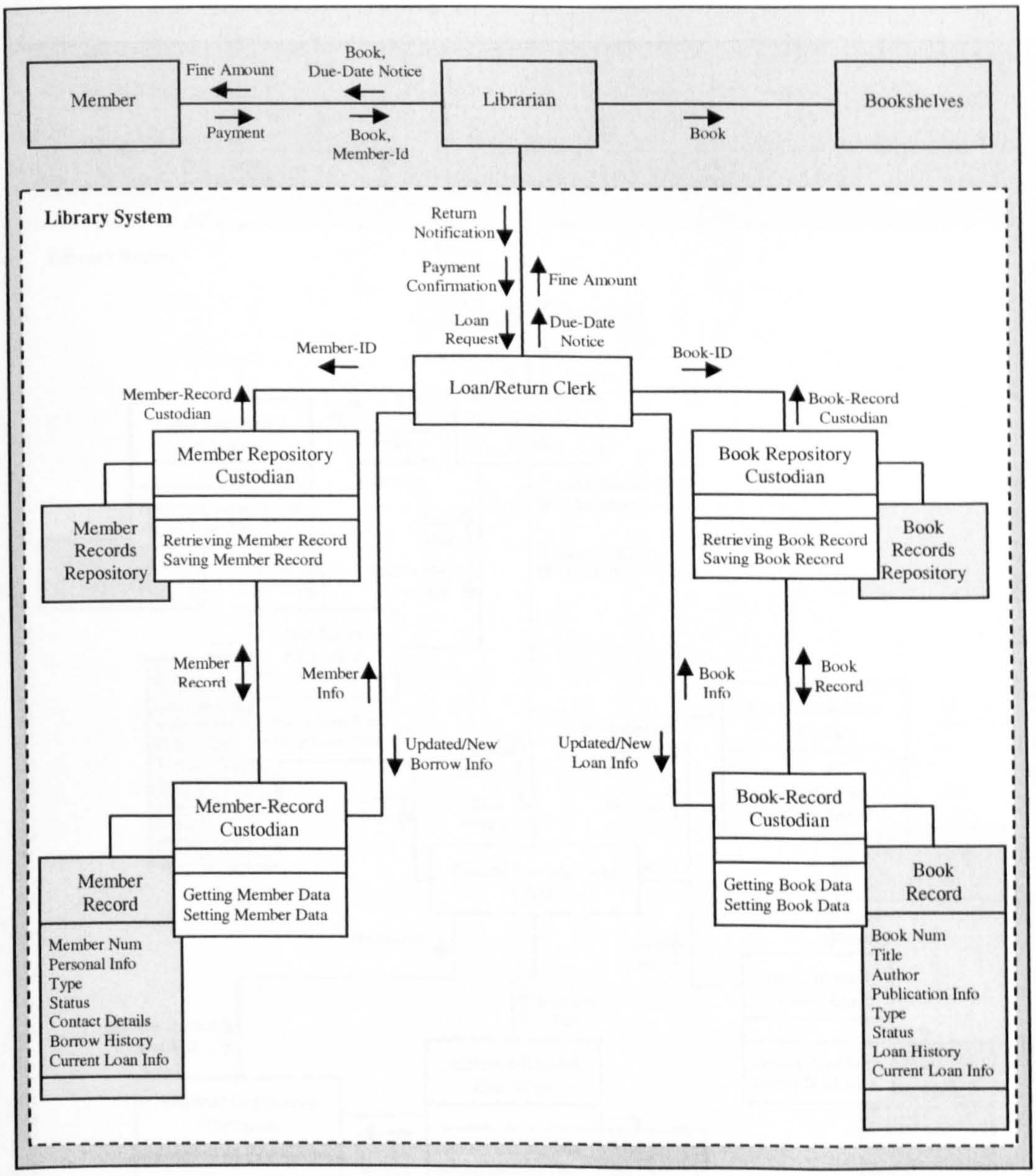

Figure 66. System Object Model using one system clerk for implementing the loan and return functionalities

Figure 67 shows an alternative design with two clerk objects. The former alternative has been chosen for final refinement due to its relative simplicity. Figure 68 shows the resulting System Object Model with feature-sets assigned to the clerk object, and custody relationships simplified in order to reduce diagram complexity.

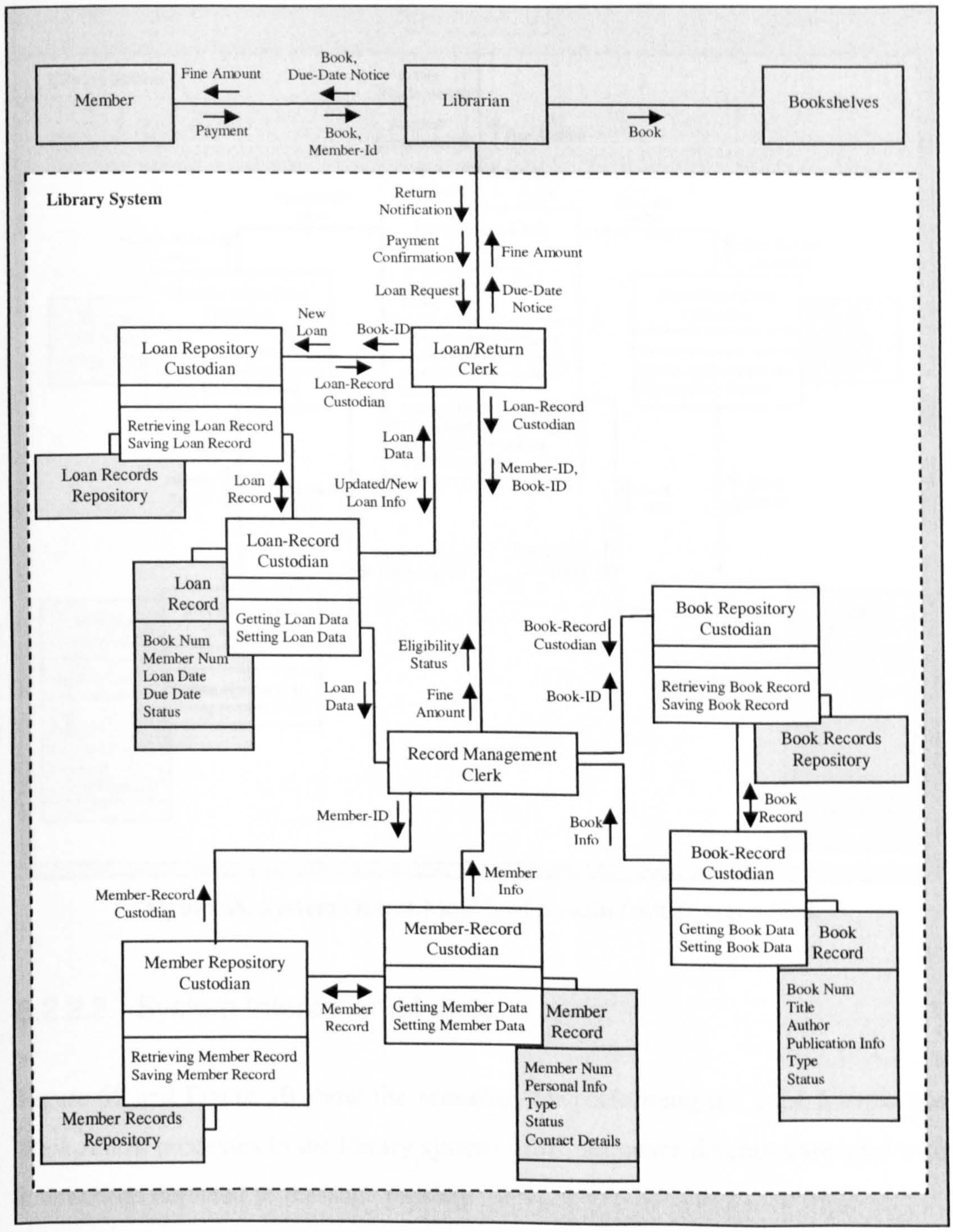

Figure 67. Alternative System Object Model using two system clerks for implementing the loan and return functionalities

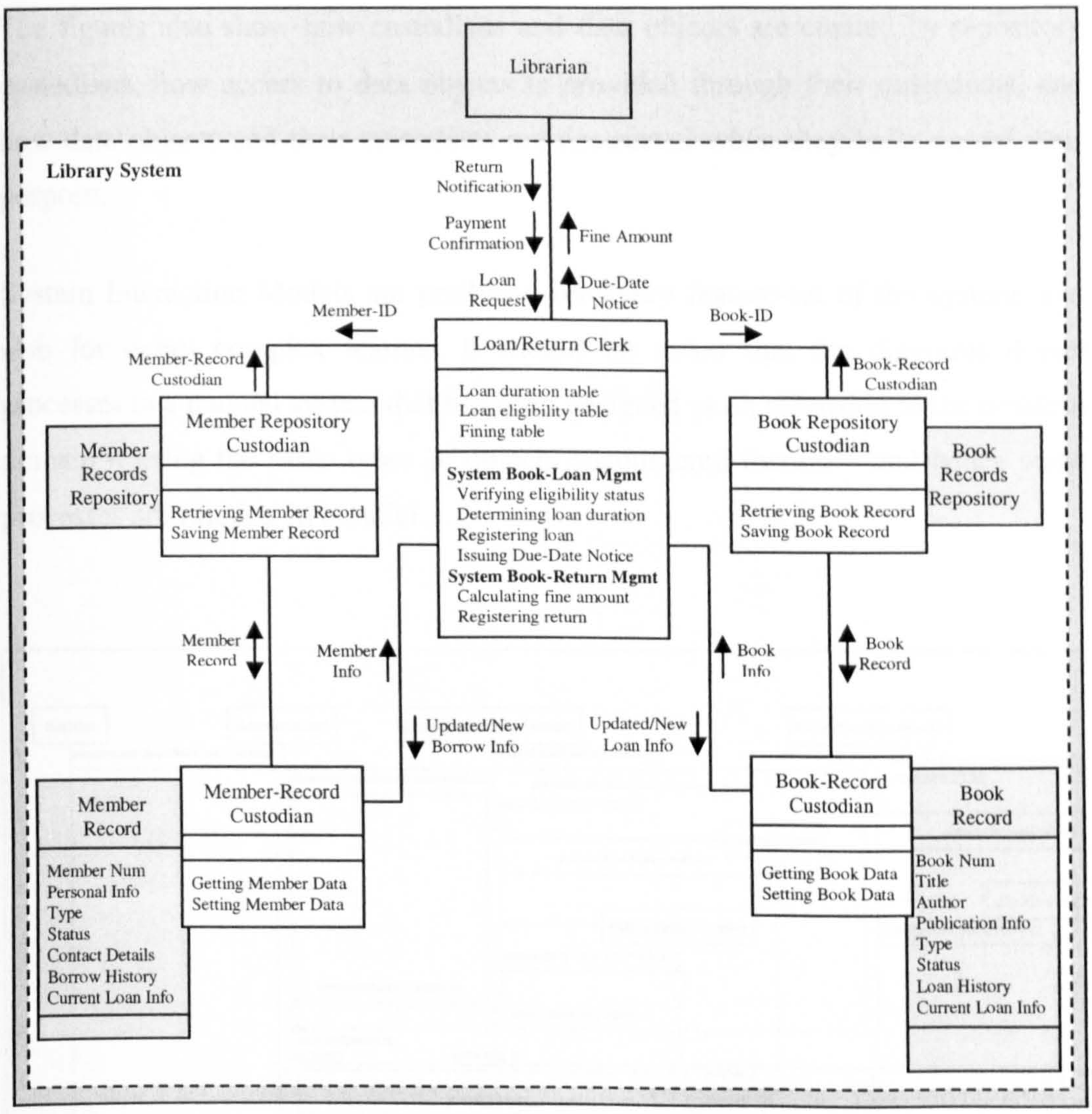

Figure 68. System Object Model, with main feature sets added

6.2.2.2 System Interaction Models

Figure 69 and Figure 70 show the scenarios for performing the *book-borrow* and *book-return* processes in the library system. UML sequence diagrams are used with interactions depicted as message passing.

As shown in the figures, data objects retrieved from data repositories are instantiated as transient objects which are assigned to specially instantiated custodian objects, and are destructed when the data is returned to the repository.

The figures also show how custodians and data objects are created by repository custodians, how access to data objects is provided through their custodians, and how data objects and their custodians are destructed when they have served their purpose.

System Interaction Models are produced for every feature-set of the system, and also for every complex feature. It should be noted that the diagrams depict processes in a manual system that has been designed as an extension to the problem domain – using the same types of objects encountered therein – and hence some processes are running in parallel.

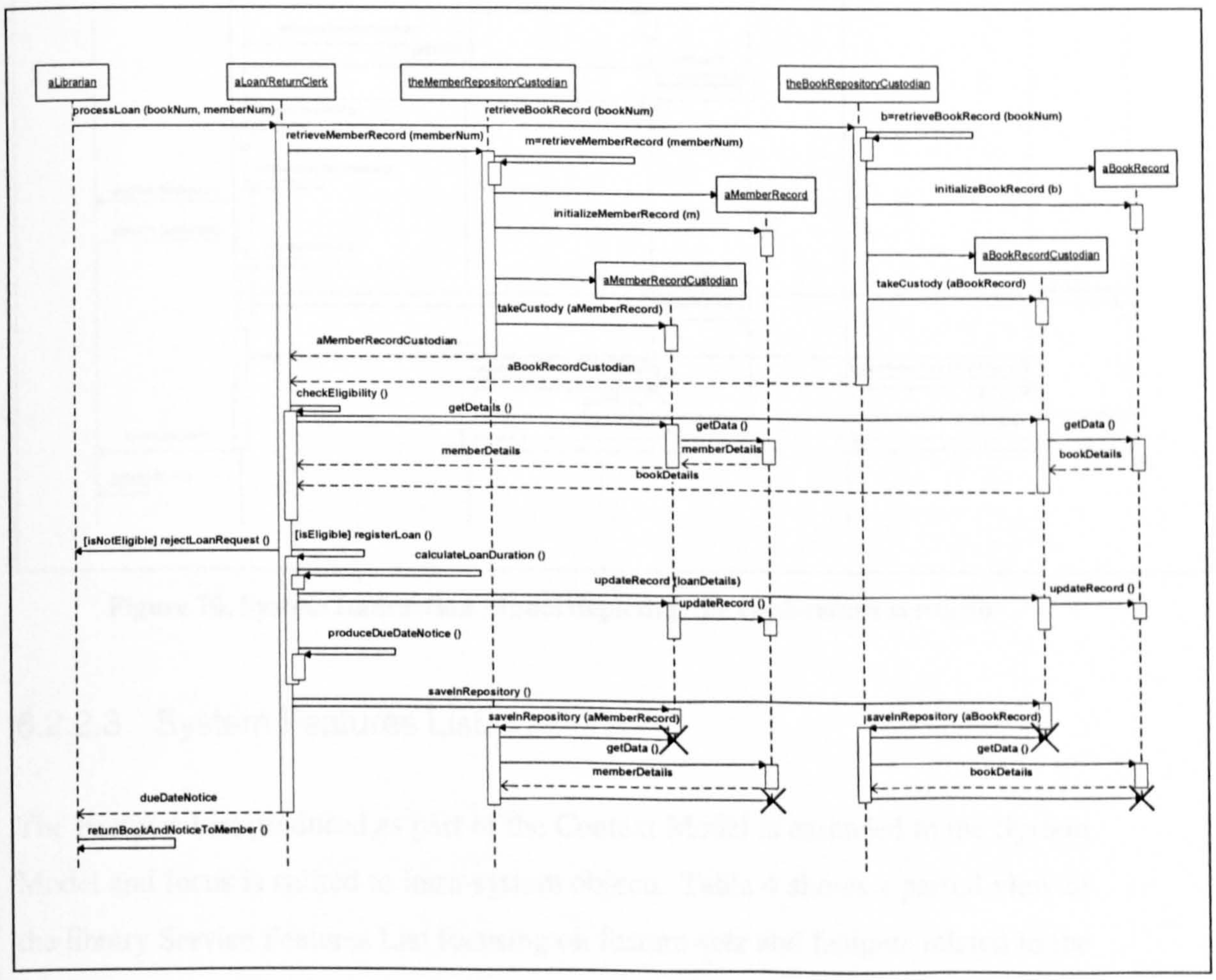

Figure 69. System Interaction Model depicting the *book-borrow* scenario

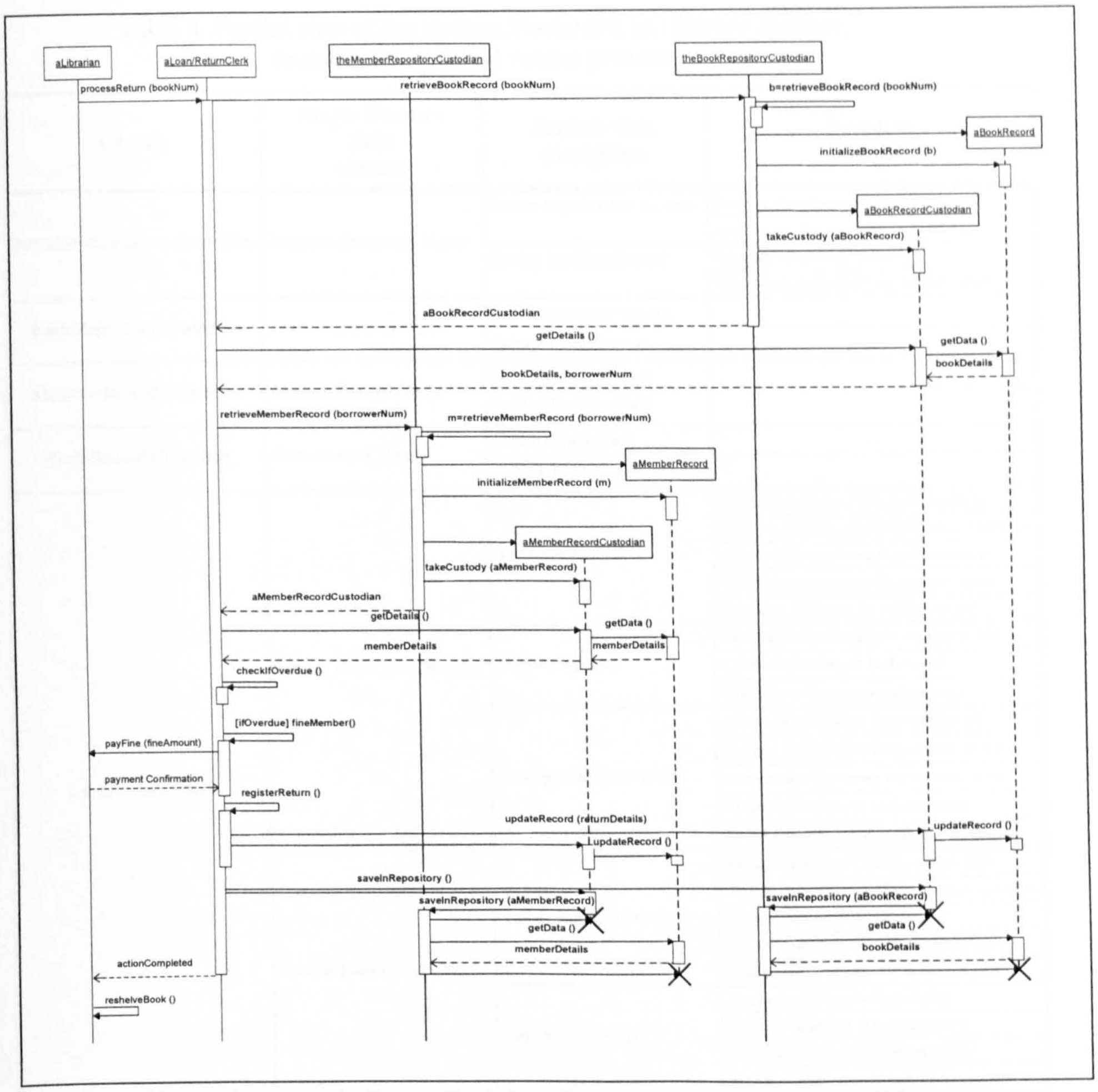

Figure 70. System Interaction Model depicting the *book-return* scenario

6.2.2.3 System Features List

The Features List produced as part of the Context Model is extended in the System Model and focus is shifted to intra-system objects. Table 4 shows a partial view of the library Service Features List focusing on feature-sets and features related to the *book-borrow* and *book-return* processes. Distribution of features among system objects is emphasized, with custodians providing access/update features and the clerk object undertaking the main processing tasks. Data and repository objects lack functionality and are therefore devoid of features.

Table 4. Partial view of the System Features List (library system), focusing on loan and return processes

Object	Major Feature Sets (*Areas*)	Feature Sets (*Activities*)	Features (*Steps*)
Member-Repository Custodian	Member-Repository Mgmt	Retrieving Member Record	Retrieve data of *member-record*
			Assign custodian to *member-record*
		Saving Member Record	Save data of *member-record*
			Disassociate custodian from *member-record*
Book Repository Custodian	Book-Repository Mgmt	Retrieving Book Record	...
		Saving Book Record	...
Member-Record Custodian	Member-Record Mgmt	Getting Member Data	...
		Setting Member Data	...
Book-Record Custodian	Book-Record Mgmt	Getting Book Data	...
		Setting Book Data	...
Loan/Return Clerk	System Book-Loan Mgmt	Verifying eligibility	Request *book-record* from *book-repository*
			Request *member-record* from *member-repository*
			Lookup loan specifications in *book-record*
			Lookup loan permissions in *member-record*
			Determine eligibility for *member-record*
		Registering loan	Determine loan duration for *member-record*
			Update loan history in *book-record*
			Update loan history in *member-record*
		Issuing Due-Date Notice	Retrieve specifications from *book-record*
			Retrieve specifications from *member-record*
			Generate due-date notice for *member-record*
			Return *book-record* to *book-repository*
			Return *member-record* to *member-repository*
	System Book-Return Mgmt	Calculating fine amount	Request *book-record* from *book-repository*
			Request *member-record* from *member-repository*
			Get loan history from *book-record*
			Get loan specifications from *member-record*
			Determine fine payable for *member-record*
		Registering return	Update loan history in *book-record*
			Update loan history in *member-record*
			Return *book-record* to *book-repository*
			Return *member-record* to *member-repository*

6.2.3 Software Model

The Software Model components presented in this section include Software Object Models and Software Interaction Models. As was the case with the System Model, detail has been limited to *book-borrow* and *book-return* processes.

Software Object Models are produced through iterative application of patterns to System Object Models. The Software Interaction Models depict the cooperation among software objects for performing the *book-borrow* and *book-return* processes.

6.2.3.1 Software Object Models

Figure 71 shows the System Object Model from the previous section along with the patterns that are applied in order to produce the Software Object Model. The sequence of pattern application conforms to that prescribed in the methodology, i.e. redistribution patterns take precedence, with refactoring patterns complementing them where needed.

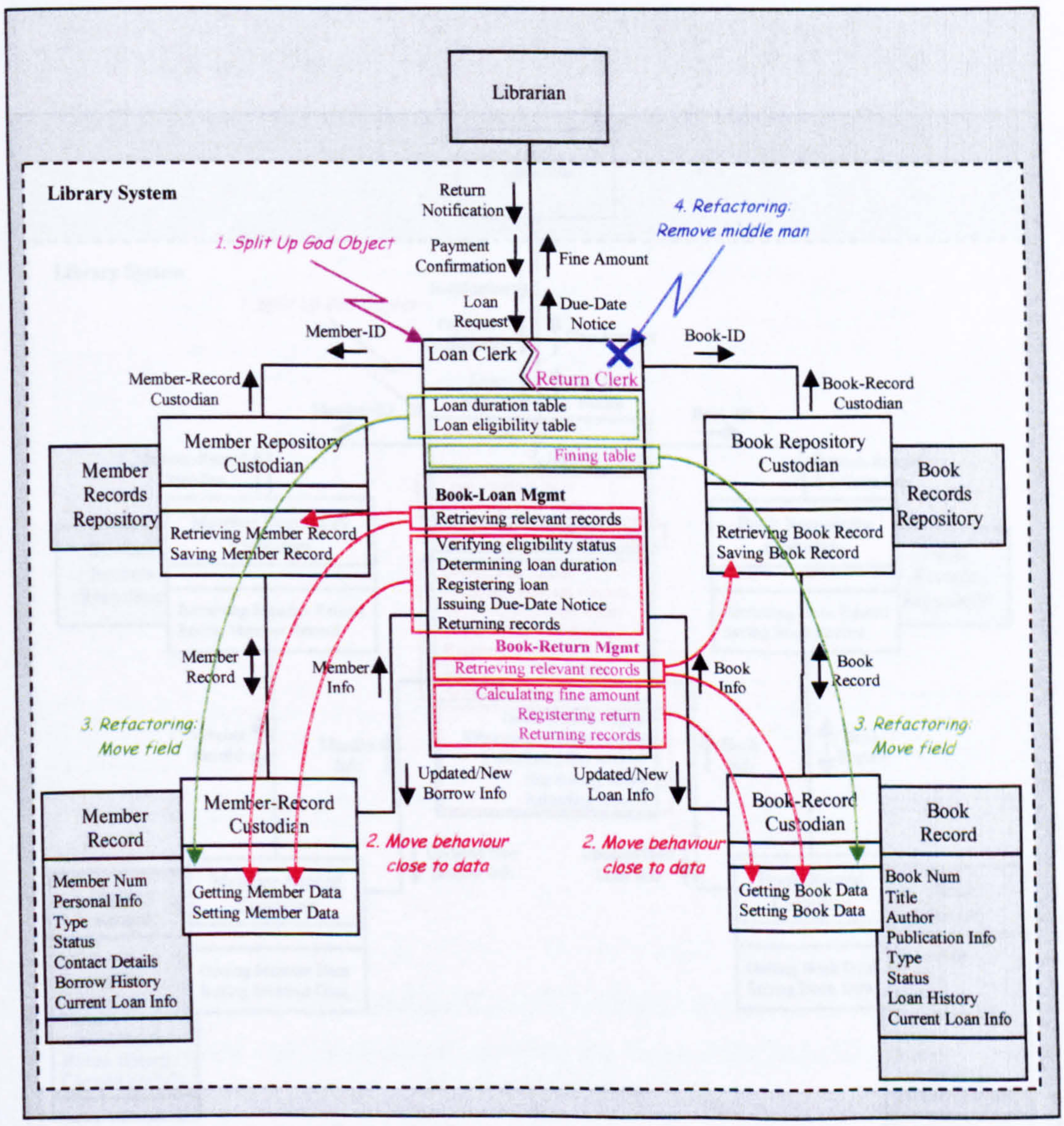

Figure 71. Object Model depicting the major patterns applied to convert the System Object Model into the Software Object Model

Other figures in this section depict the step-by-step process of pattern application, showing how each pattern is applied and the resulting changes in the Object Model. Transformations are highlighted in order to emphasize the effects of each pattern on the model.

Figure 72 shows the results of applying the *Split-Up-God-Class(Object)* pattern to the System Object Model. The *Loan/Return-Clerk* object is thus split up into two objects: the Loan-Clerk and the Return-Clerk. The features and data fields are moved to their corresponding object.

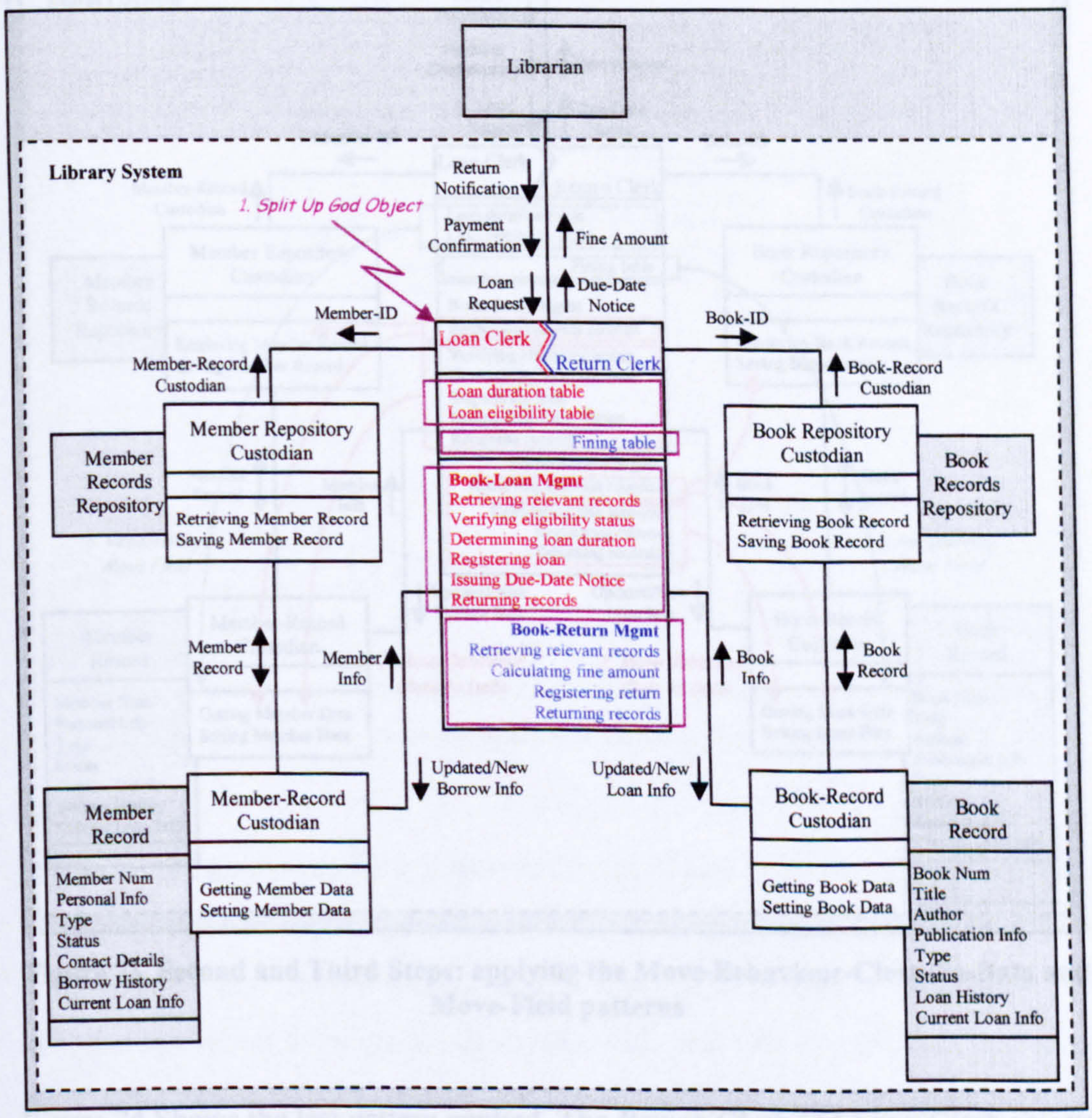

Figure 72. First Step: applying the Split-Up-God-Object pattern

Figure 73 shows the way the second and third patterns are applied. The *Move-Behaviour-Close-to-Data* pattern is first applied to move the relative behaviour to custodians. The *Move-Field* refactoring pattern is then applied to do the same with data fields. As a result, the two clerk objects lose their structure and behaviour and become simple intermediaries between the Librarian outside the system and the custodians.

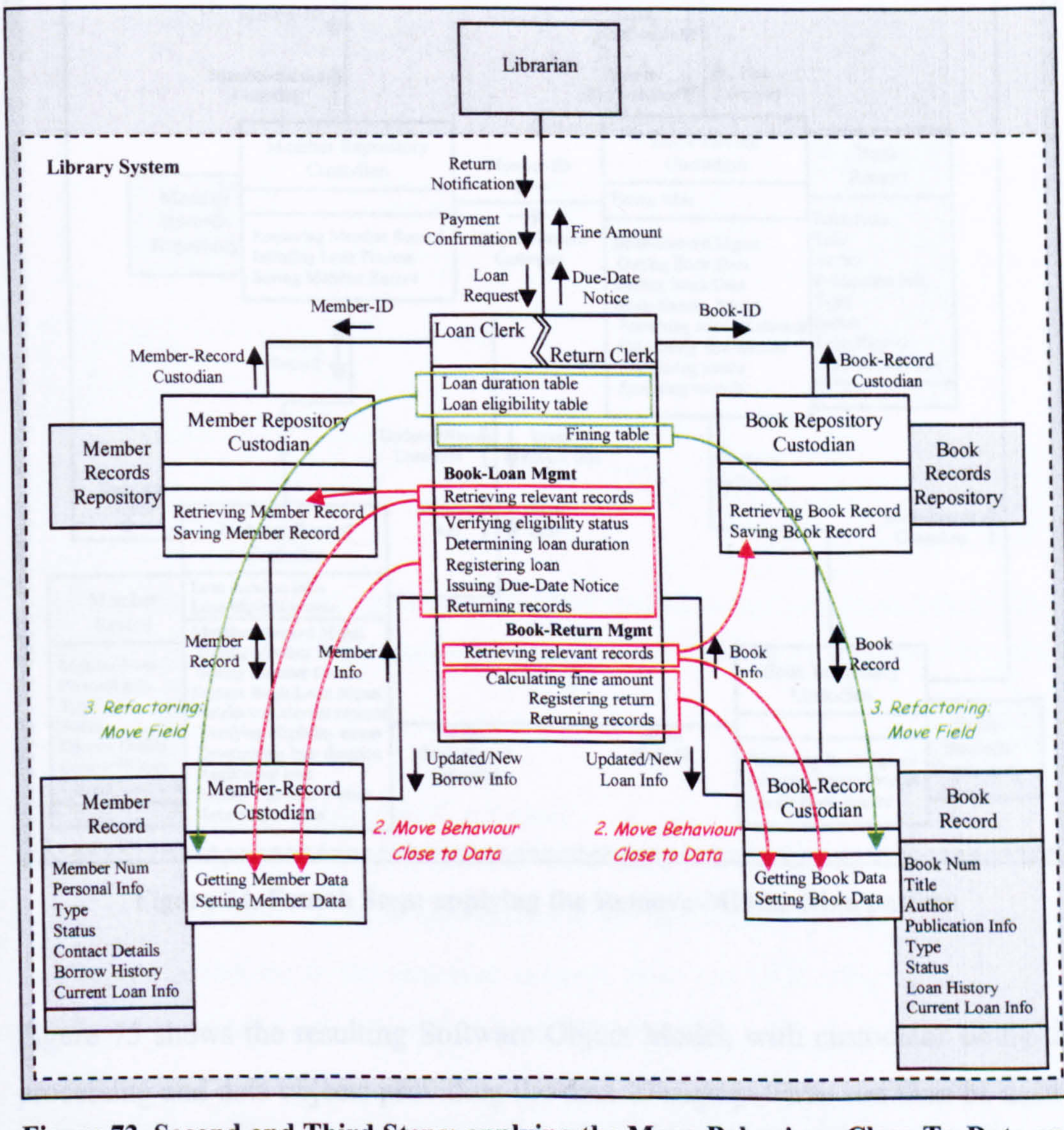

Figure 73. Second and Third Steps: applying the Move-Behaviour-Close-To-Data and Move-Field patterns

Figure 74 Shows the last pattern applied. The *Return-Clerk* and *Loan-Clerk* objects are now little more than empty shells. The *Remove-Middleman* refactoring pattern is hence applied to establish direct links between the custodians and the external Librarian. Alternatively, the *Poltergeist* antipattern can be used with the exact same effect.

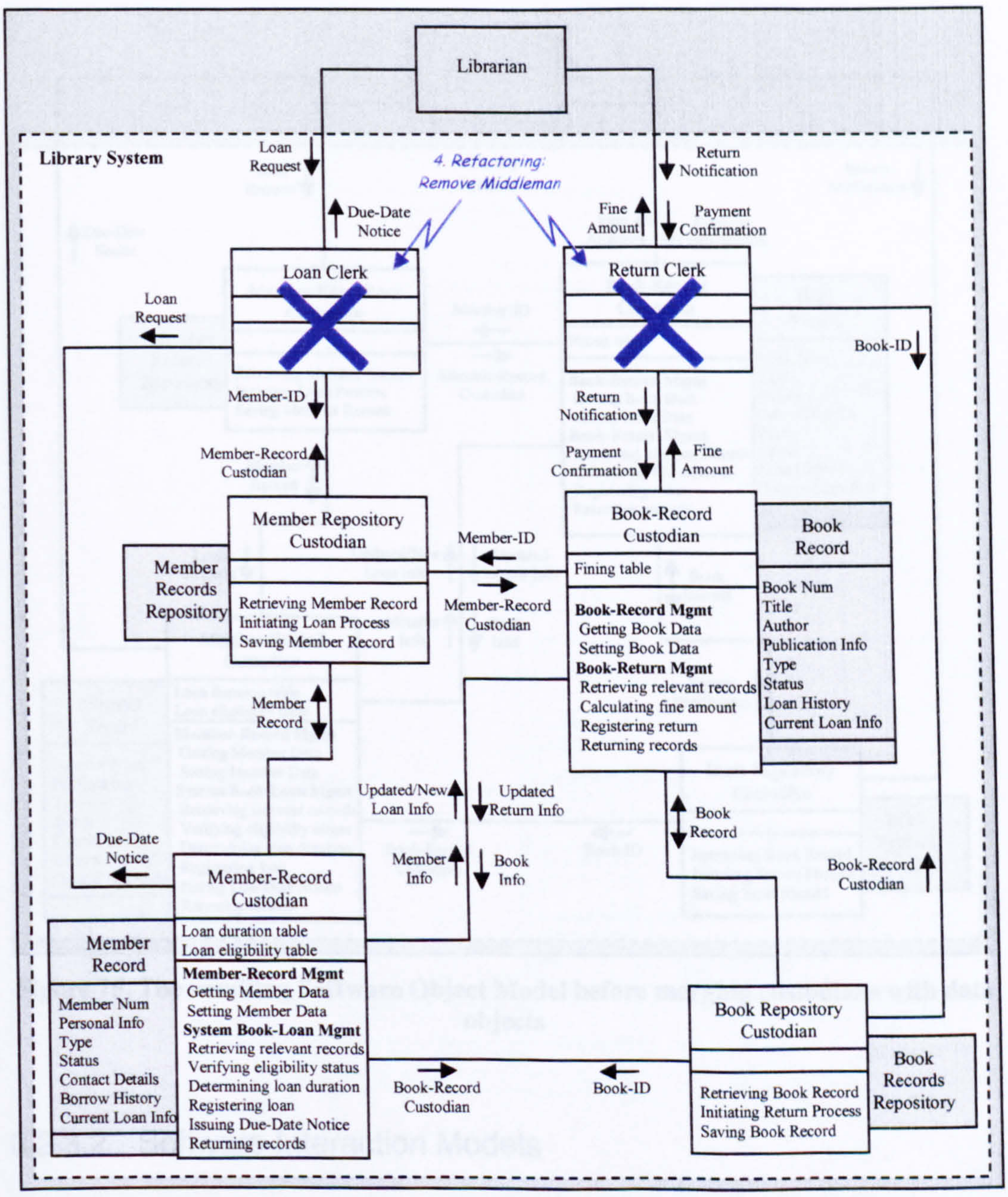

Figure 74. Fourth Step: applying the Remove-Middle-Man pattern

Figure 75 shows the resulting Software Object Model, with custodians doing the processing and data objects providing the data. Design patterns can then be used – if applicable – to introduce specialized structure and behaviour in the models. The next and last step is to merge the custodians with their data objects, thus producing objects encapsulating both state and behaviour. Software Class Models can then be produced, highlighting classifications of objects and their relationships, especially aggregation and generalization/specialization.

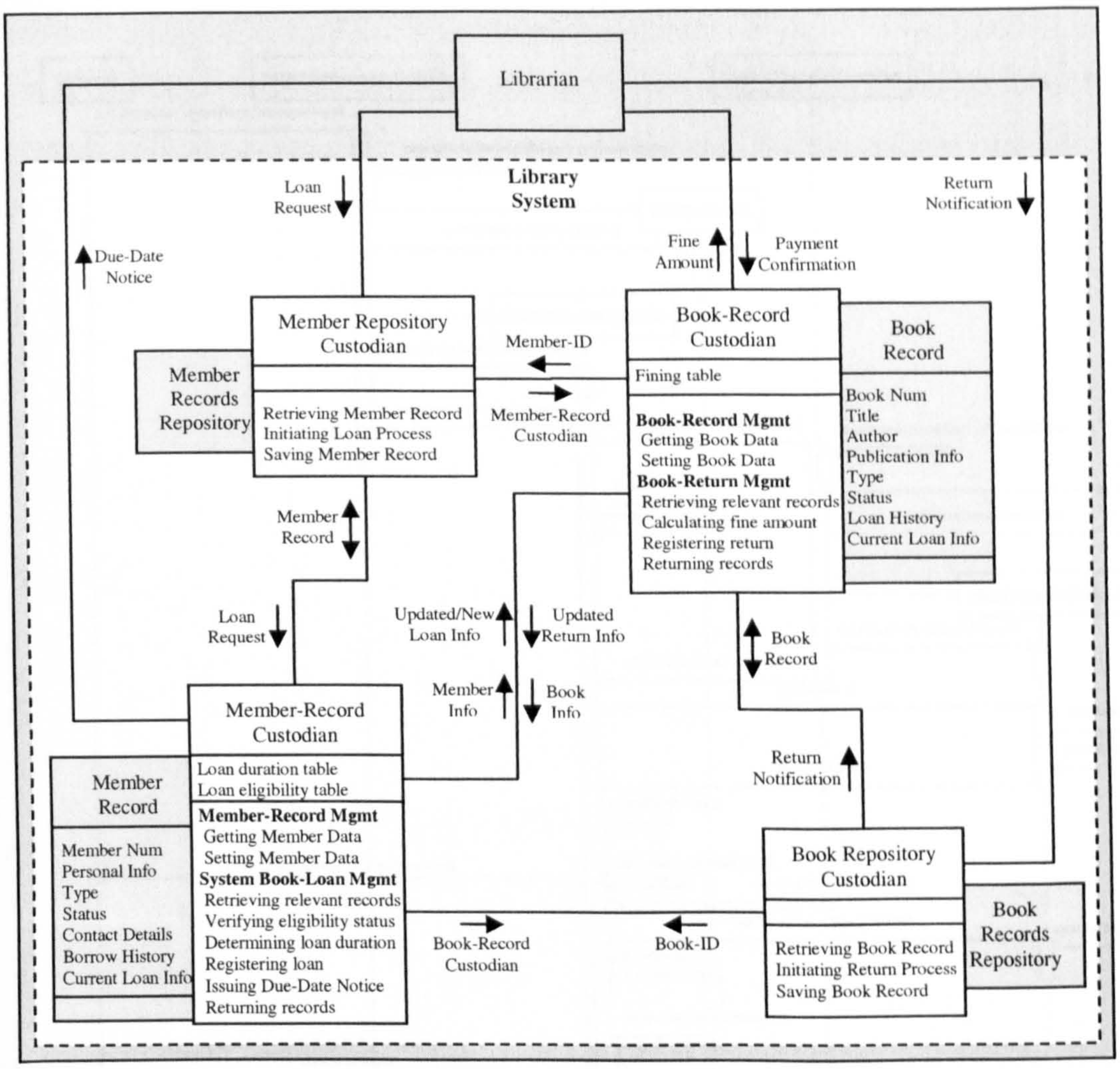

Figure 75. The resulting Software Object Model before merging custodians with data objects

6.2.3.2 Software Interaction Models

Figure 76 and Figure 77 show the scenarios for performing the *book-borrow* and *book-return* processes in the software system. Here too UML sequence diagrams are used with interactions depicted as message passing.

As these figures show, repository custodians are in charge of initiating the loan and return processes. Control is then passed to record custodians to implement the main functionality.

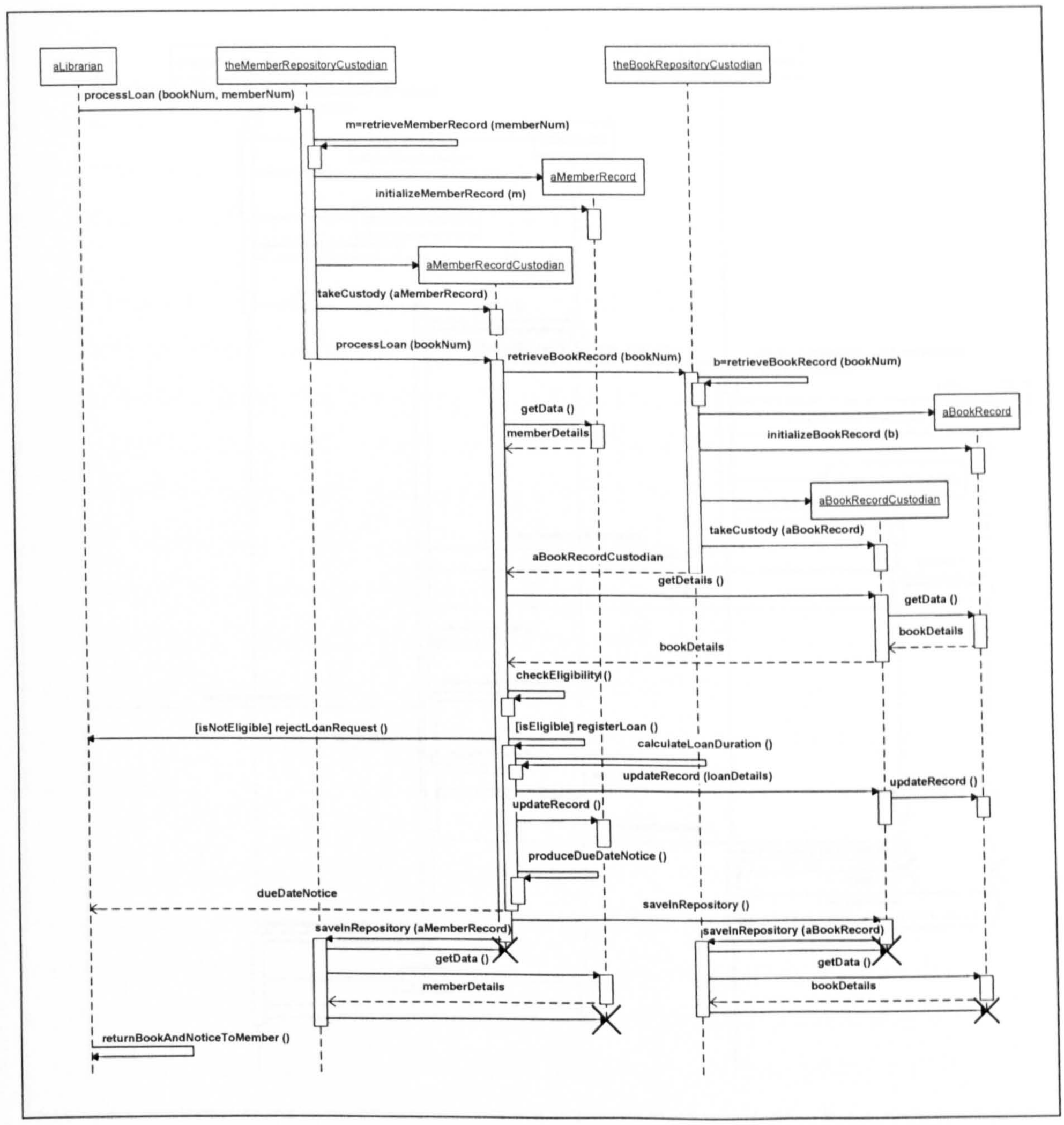

Figure 76. Software Interaction Model depicting the *book-borrow* scenario

Software Interaction Models will later be used alongside Software Object/Class Models to produce class- and method prologues during iterative design and implementation. Due to the feature-driven nature of models and development tasks, feature-based interaction scenarios are preserved as bases throughout the development process (as observed in the Library examples above), to ultimately influence class- and method prologues, and the final software code.

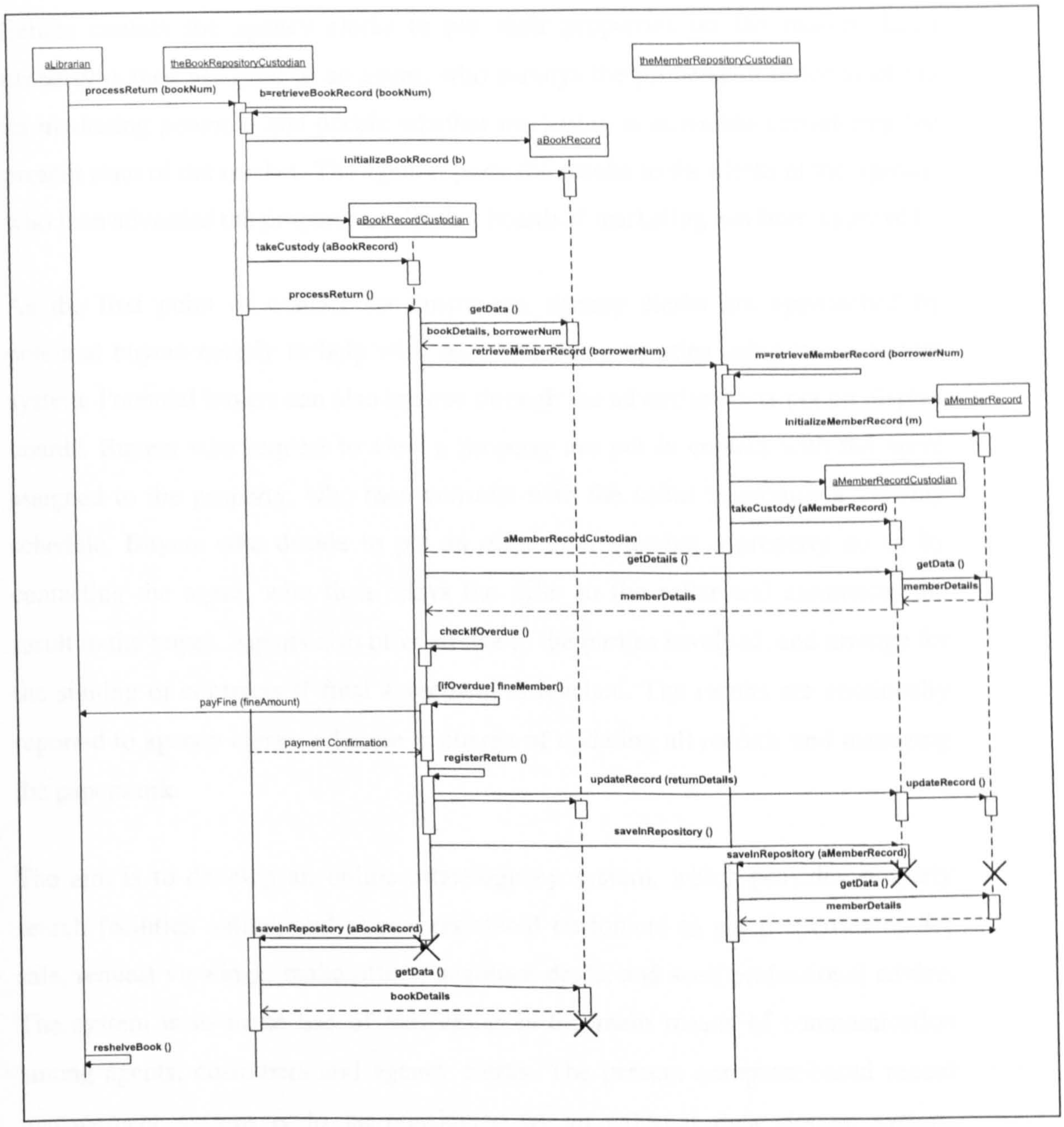

Figure 77. Software Interaction Model depicting the *book-return* scenario

6.3 Case Study 2: Estate Agency System

The estate agency system targeted in this case study is currently a computer-based system, with potential *buyers* and *sellers* visiting the premises of the agency in person or contacting agency *clerks* by phone in order to obtain information about properties on the market, put new properties up for sale, request viewings, or make an offer on a property. Information about properties, customers and transactions is stored in a database and maintained via an existing computer-based *record management system*.

Sellers contact the agency clerks to put their properties on the market. Each property is then assigned to an *agent*, who surveys the property in order to assess its marketing potential and decide whether marketing is advisable considering the present state of the market. The agent reports the results to the clerks at the agency, who then advertise the property on display boards if marketing has been approved.

As the first point of contact for customers, agency clerks are approached by potential buyers mainly to help with searching for properties using the computer system. Potential buyers can also browse through the advertisements put on display boards. Buyers who request to view a property are put in contact with the agent assigned to the property, who then consults with the seller to arrange a viewing schedule. Buyers who decide to put an offer after viewing a property do so by contacting the agent, who then relays the offer to the seller and announces the result to the buyer. Agents also offer advice to the parties involved, and arrange for the signing of contracts if final agreement is reached. The results are continually reported to agency clerks, who are in charge of updating all records and managing the paperwork.

The aim is to develop an online estate-agency system, which provides property search facilities online, and allows registered customers to put properties up for sale, request viewings, make offers, negotiate deals, and seek professional advice. The system is to make use of messaging as the main means of communication among agents, customers and agency clerks. The present computer-based record management system is to be considered as an external data storage system, interfaced in order to provide database management facilities to the online system.

The following sections contain the results of the modeling activities performed on the system through the application of the development methodology. For sake of brevity, when modeling detailed aspects of the system, focus has been limited to the three basic functions of *putting a property up for sale*, *arranging a viewing*, and *making an offer on a property*.

As the main purpose in conducting this case study has been to verify the applicability of the methodology and the pattern-based model-transformation approach to problem domains that already contain computerised elements, emphasis has been put on model conversion; Object Models have therefore taken

precedence, and some less relevant behavioural models have consequently been omitted from the results. Verification using this case study confirmed the findings reported in Section 6.2.

6.3.1 Context Model

The Context Model components presented in this section include Context Object Models, Context Interaction Models, the Context Features List, and a partial Glossary of Terms.

The Context Object Models show a representation of the estate agency problem domain as encountered in the real world, with the target system then added as a problem domain object. The Context Interaction Models depict the cooperation among problem domain objects for performing the *put-property-up-for-sale*, *viewing*, and *make-an-offer* processes. Context Interaction models come in two versions: the first versions model the real world, and the latter ones depict the interactions after the target system is added as a problem domain object and is involved in inter-object cooperation. The Context Features List shows the major feature sets (*areas*), as well as their constituent feature sets (*activities*) and bottom-level features (*steps*), with the detail mostly confined to features pertaining to *put-property-up-for-sale*, *viewing*, and *make-an-offer* processes. The list also shows the assignment of feature sets and features to problem domain objects.

6.3.1.1 Context Object Models

Figure 78 shows the real-world Context Object model of the estate agency problem domain. The diagram also shows the assignment of feature-sets and features to objects. Figure 79 shows the Context Object Model after the target system is added as a problem domain object. Feature sets have been redistributed and new feature-sets have been added to the system.

6.3.1.2 Context Interaction Models

Figure 80, Figure 81, and Figure 82 show the scenarios for performing the *put-property-up-for-sale*, *viewing*, and *make-an-offer* processes in the real-world estate agency. UML activity diagrams are used at this stage with swimlanes depicting the

active objects participating in the processes. As seen in the figures, active objects cooperate to perform the *put-property-up-for-sale*, *viewing*, and *make-an-offer* scenarios. Storage objects (such as *ad-boards*) are not modeled as participating objects due to their passive roles in the system, nevertheless references to their usage by active objects can be seen in activity descriptions.

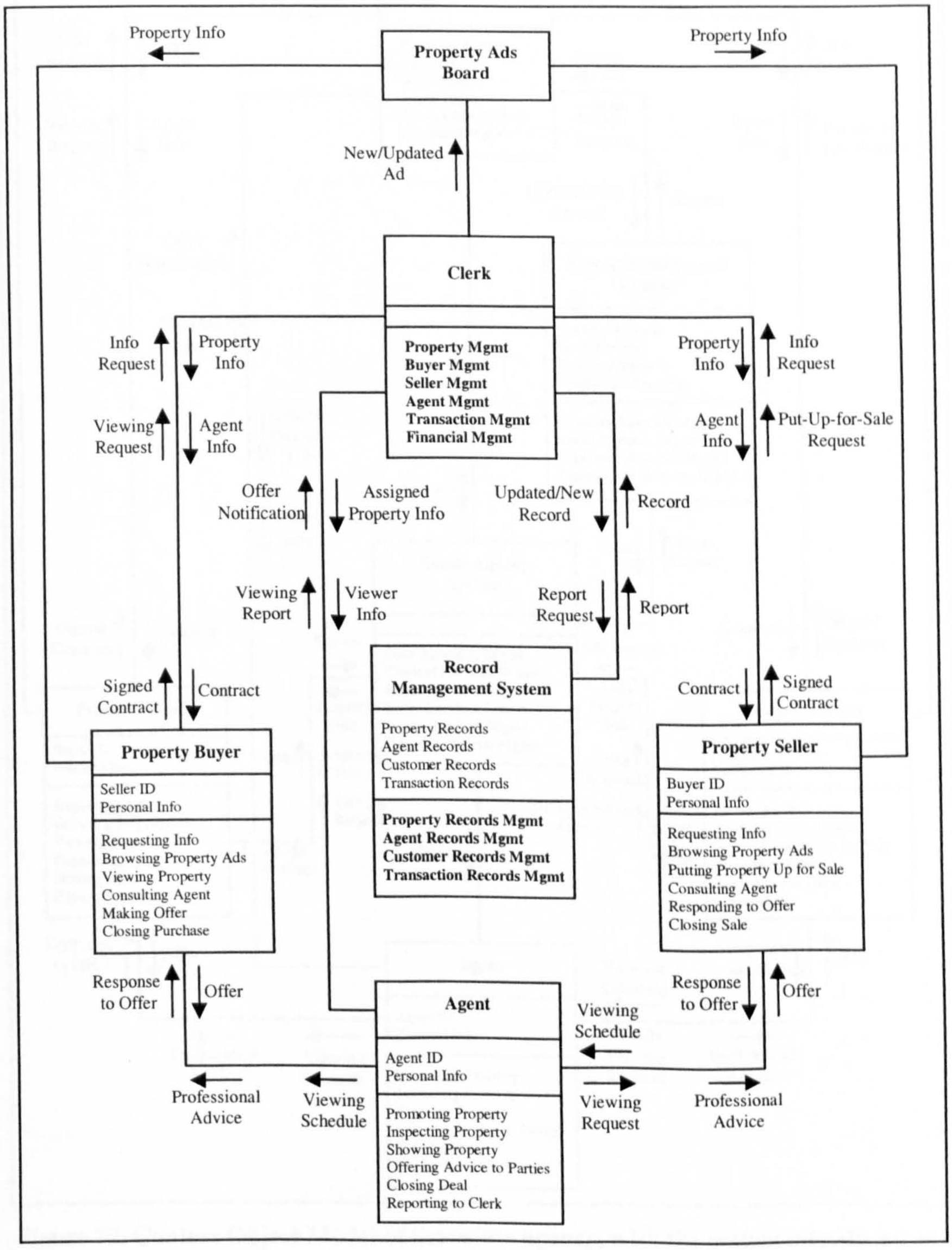

Figure 78. Context Object Model of the Estate Agency problem domain, depicting problem domain objects, feature sets and inter-object data flows

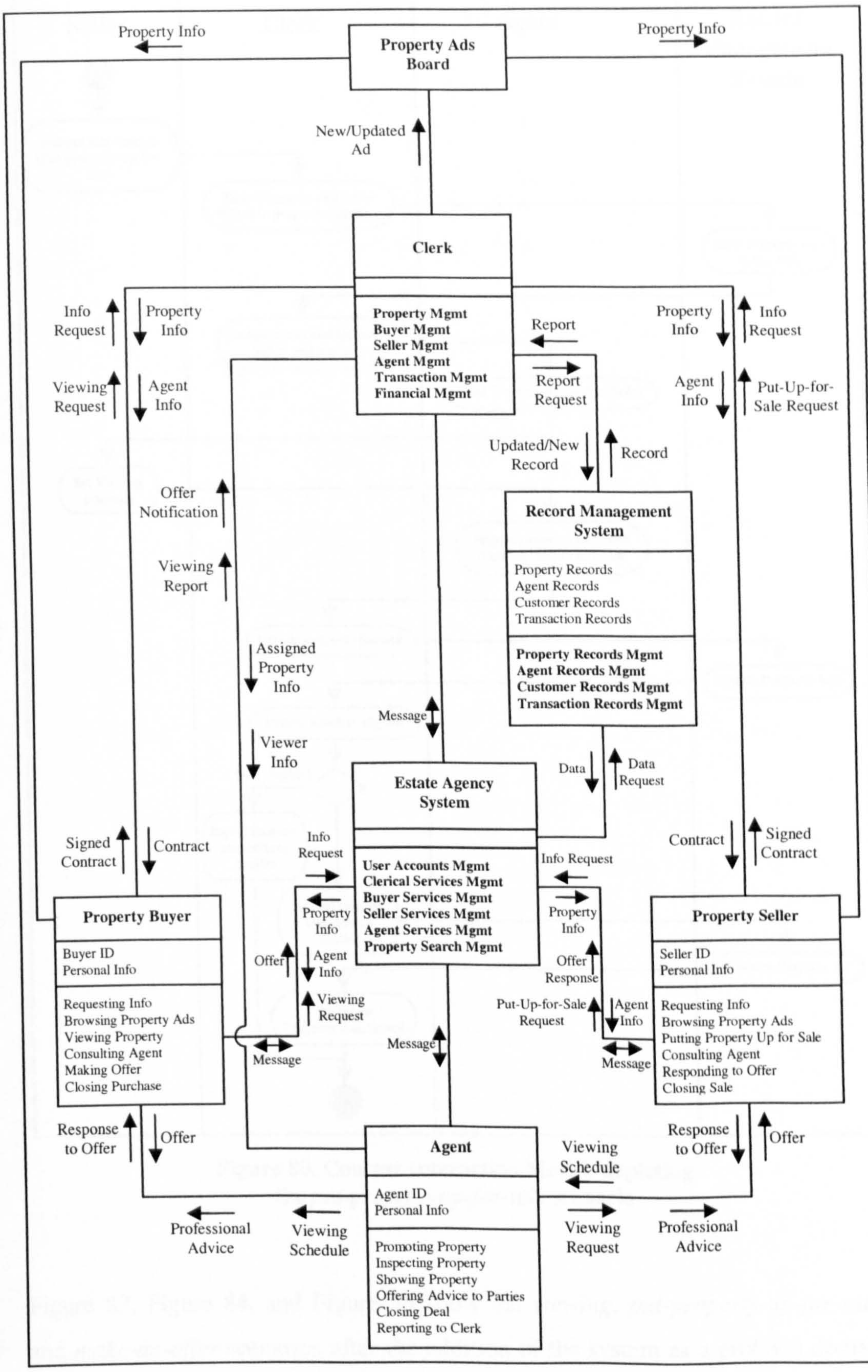

Figure 79. Context Object Model of the estate agency, with the system introduced as a problem domain object

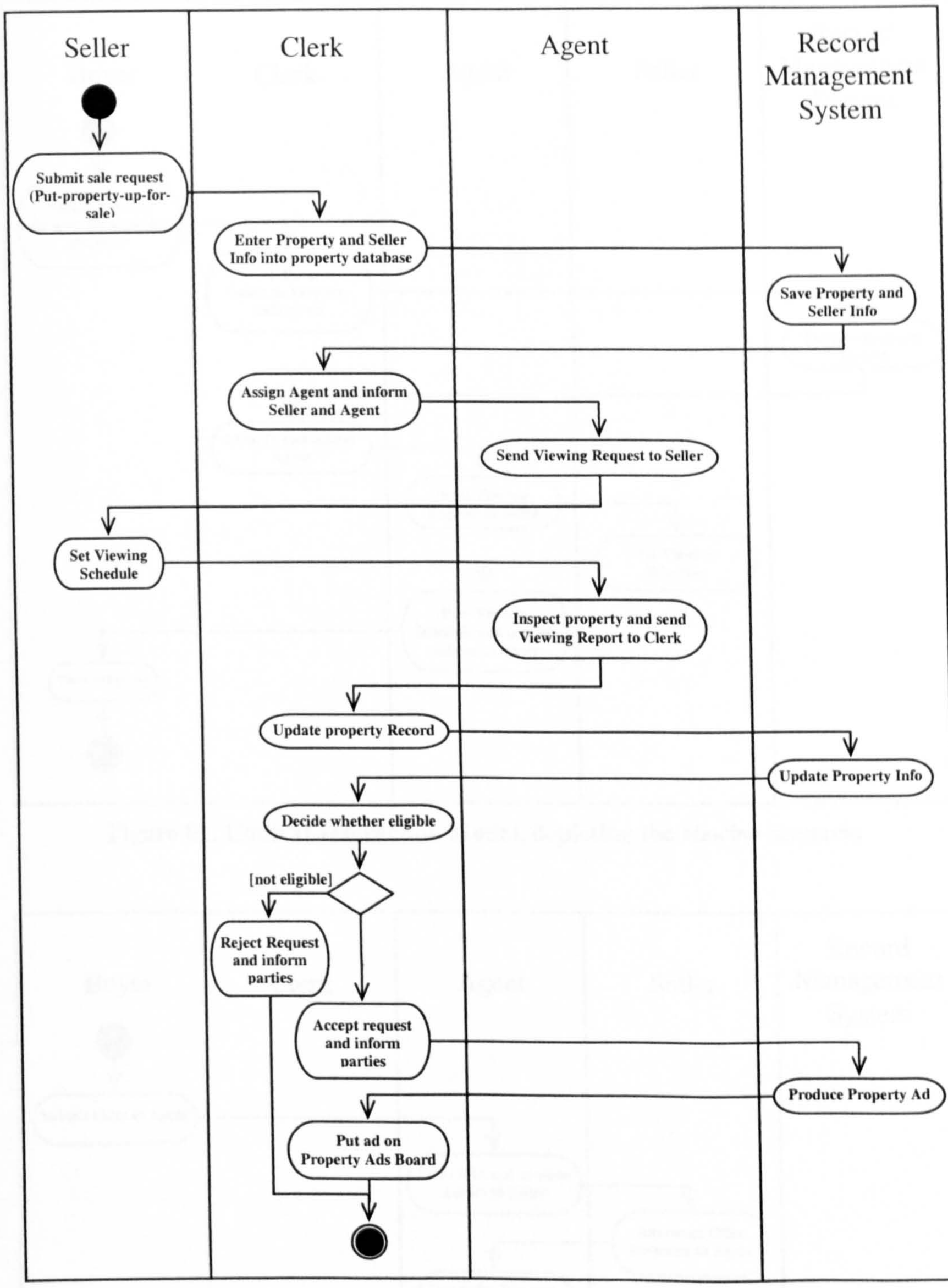

Figure 80. Context Interaction Model, depicting the *put-property-up-for-sale* scenario

Figure 83, Figure 84, and Figure 85 show the *viewing*, *put-property-up-for-sale*, and *make-an-offer* scenarios after the addition of the system as a problem domain object. The system has been assigned a separate swimlane, and activities and functionalities have been redistributed.

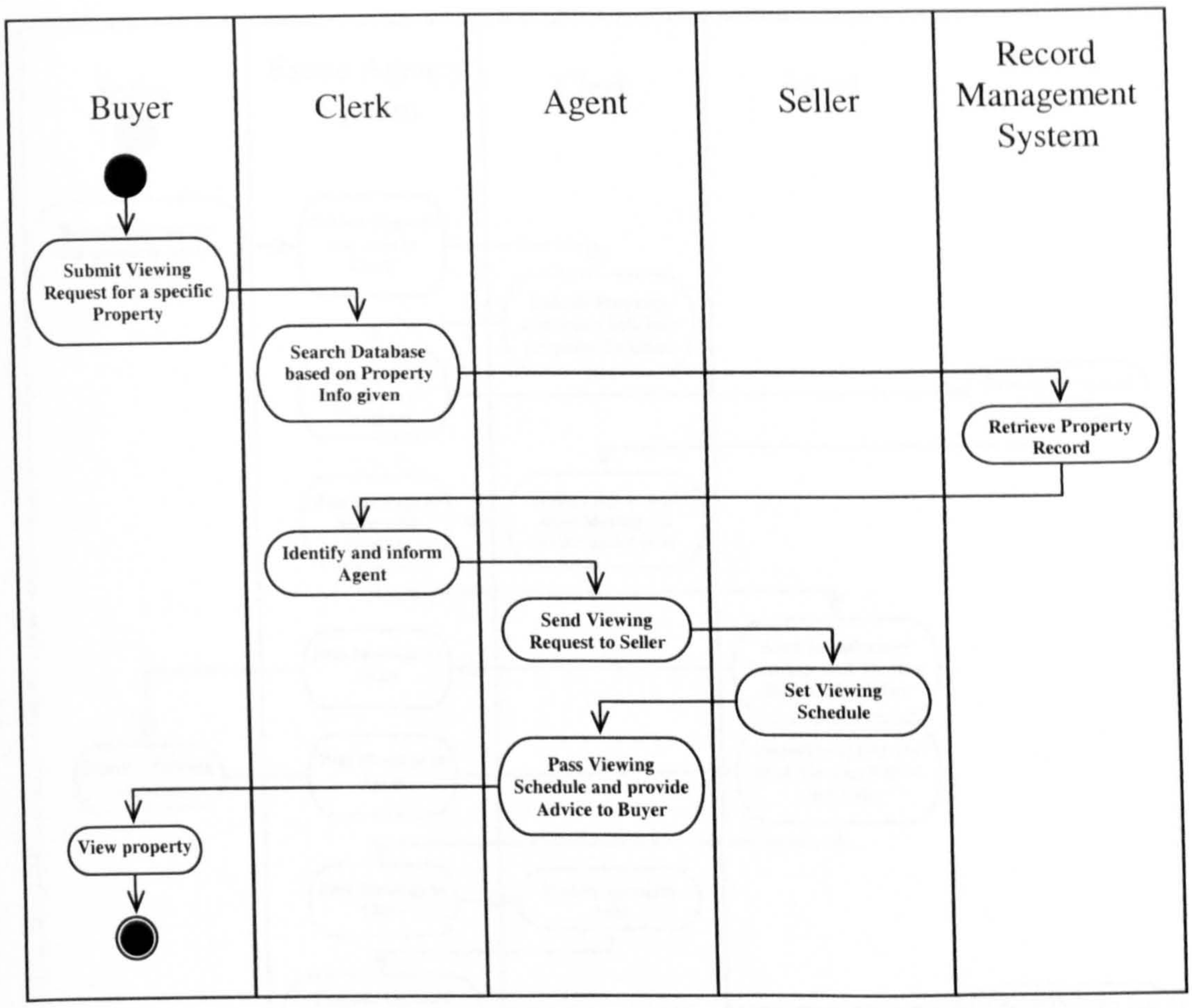

Figure 81. Context Interaction Model, depicting the *viewing* scenario

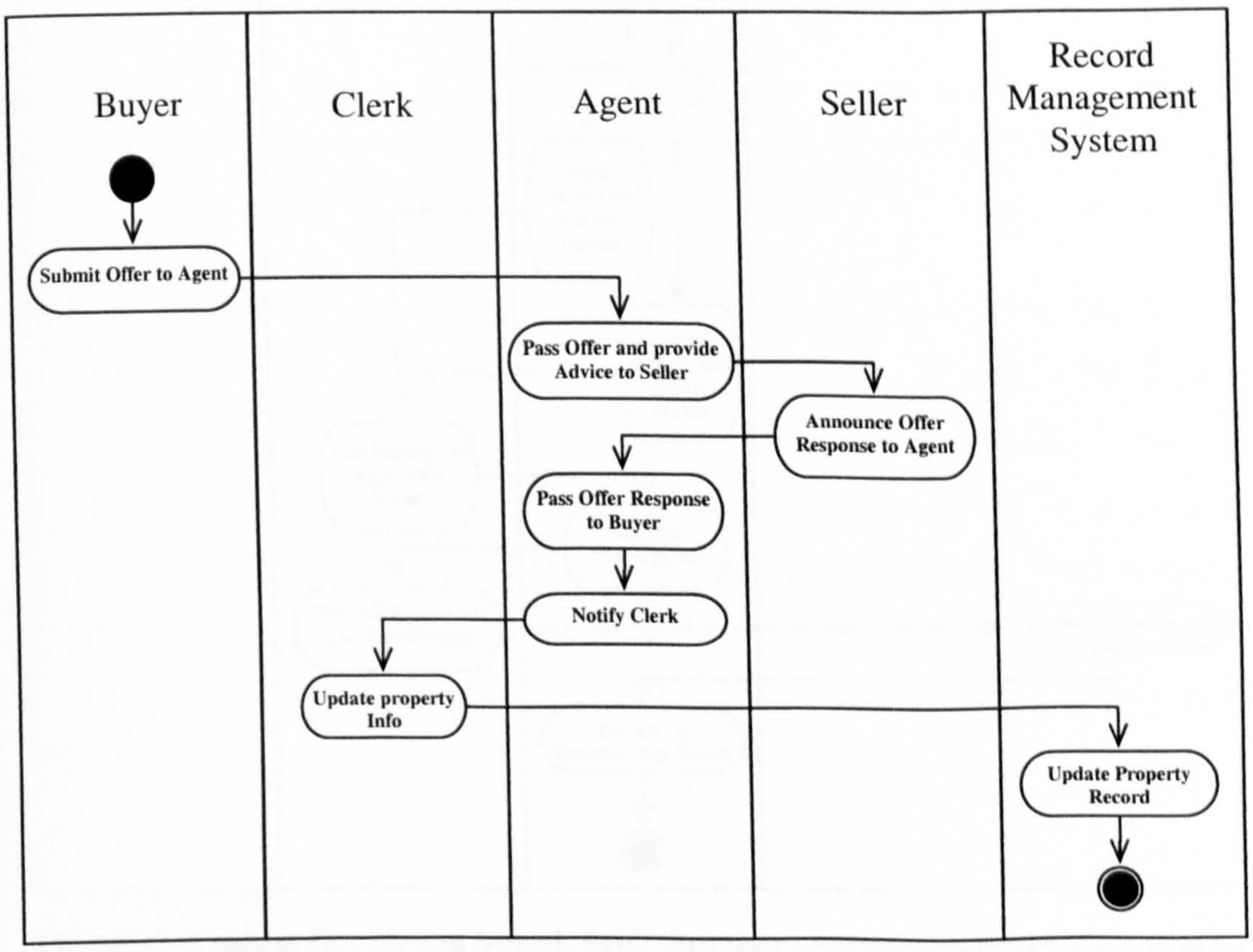

Figure 82. Context Interaction Model, depicting the *make-an-offer* scenario

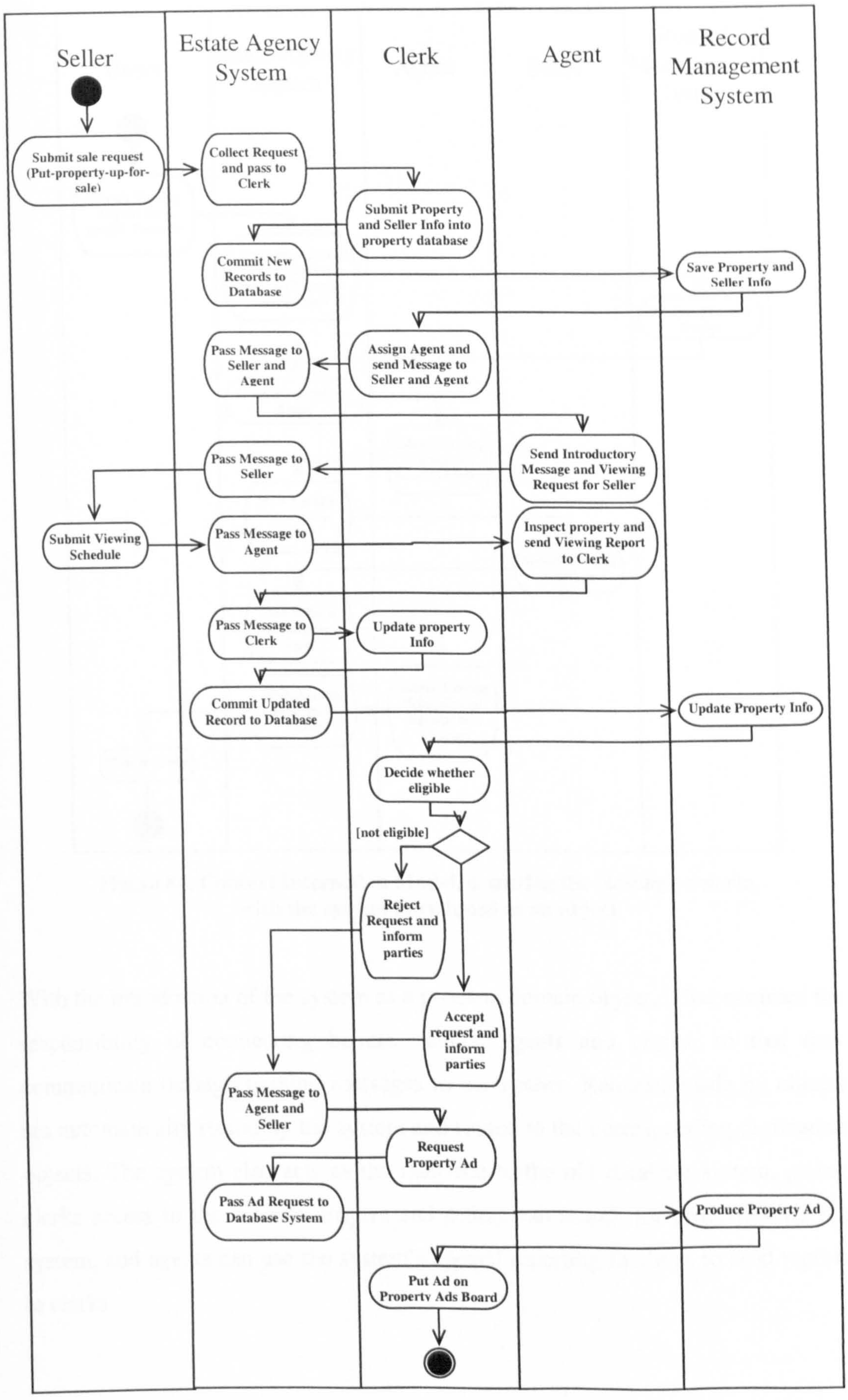

Figure 83. Context Interaction Model, depicting the *put-property-up-for-sale* scenario, with the system introduced as an object

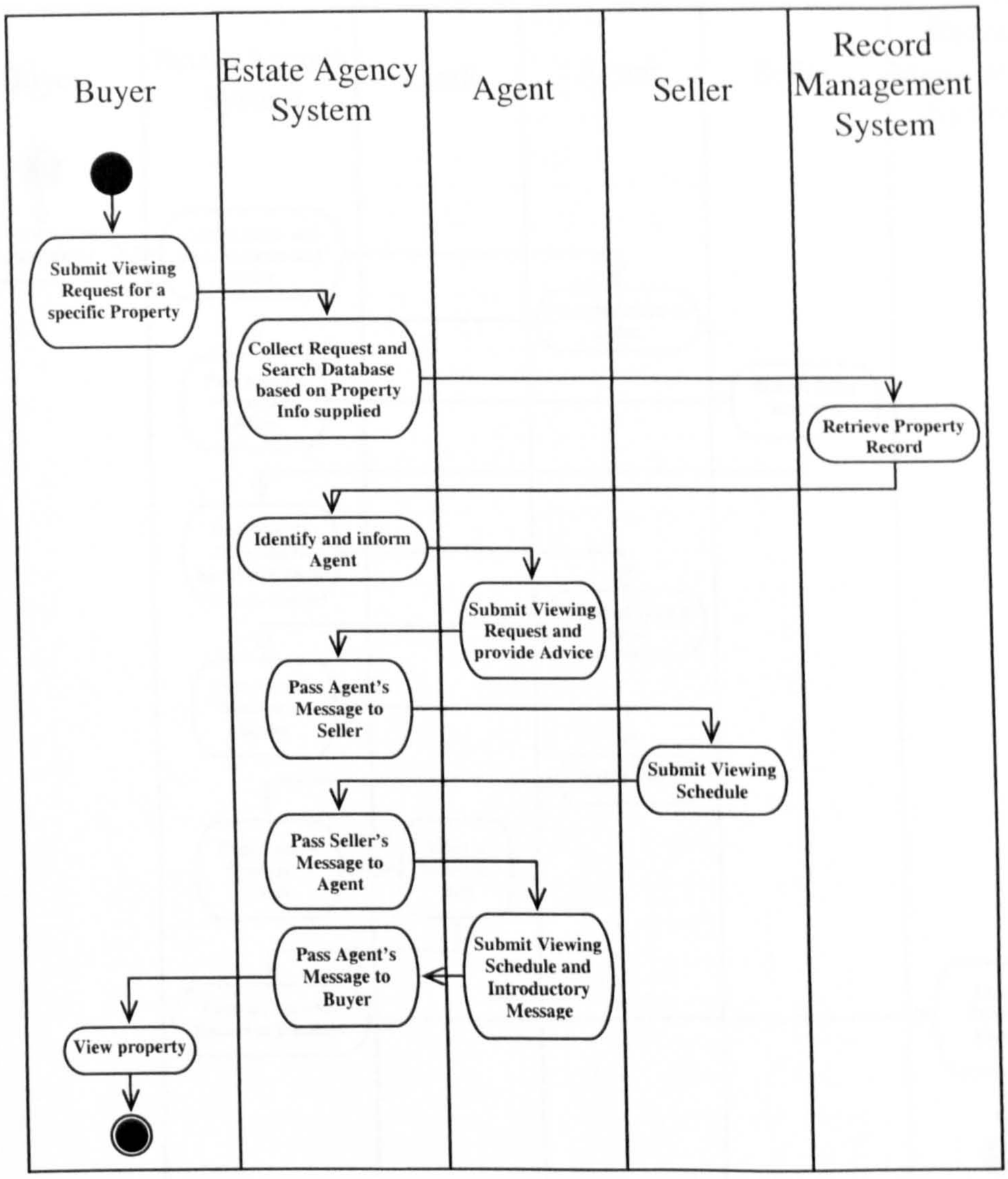

Figure 84. Context Interaction Model, depicting the *viewing* scenario, with the system introduced as an object

With the introduction of the system as a problem domain object, it has assumed the responsibility of connecting buyers, sellers, agents and clerks, so that they communicate through passing messages to each other. Requests made by objects are automatically stored by the system and routed to the corresponding destination objects. The system also acts as the interface to the old database system, giving clerks access to the records. Buyers and sellers can search for properties via the system, and agents can use the system's special reporting facilities to send reports to clerks.

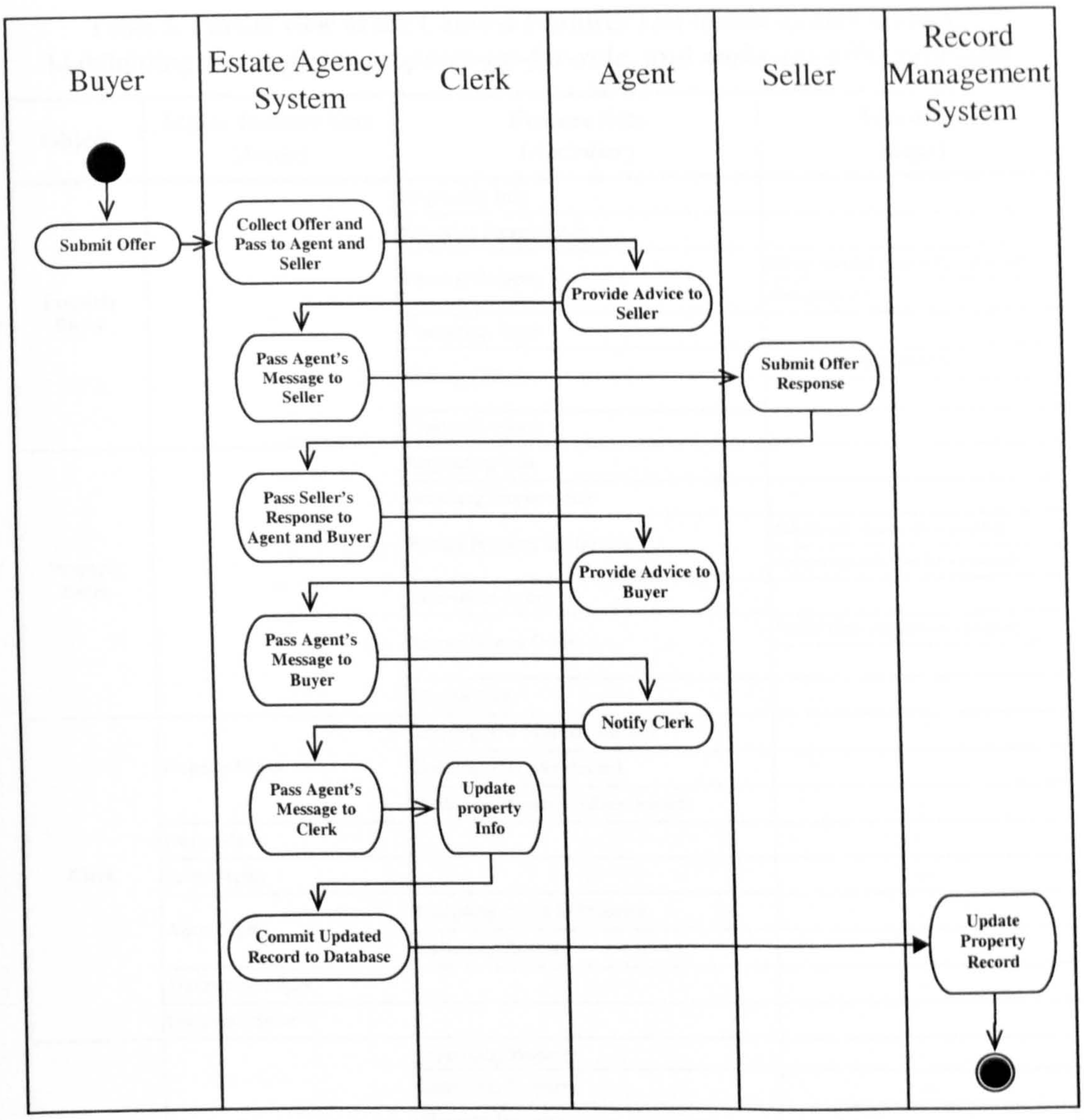

Figure 85. Context Interaction Model, depicting the *make-an-offer* scenario, with the system introduced as an object

6.3.1.3 Context Features List

In the initial version of the estate agency Context Features List, the functions that each active object of the problem domain performs are listed and expressed as higher-level feature sets (*areas* and *activities*) and low-level features. When the target system is added as a problem domain object, responsibilities are redistributed and feature sets and features redefined. Table 5 shows a partial view of the estate agency Context Features List after the system has been added, highlighting feature-sets and features related to the *viewing*, *put-property-up-for-sale*, and *make-an-offer* functionalities.

Table 5. Partial view of the Context Features List (estate agency system), highlighting *viewing*, *put-property-up-for-sale*, and *make-an-offer* processes

Object	Major Feature Sets (*Areas*)	Feature Sets (*Activities*)	Features (*Steps*)
Property Buyer	–	Requesting Info	…
		Browsing Property Ads	…
		Viewing Property	Submit viewing request for a *property*
			View *property*
		Consulting Agent	…
		Making Offer	Submit offer on a *property*
			…
		Closing Purchase	…
Property Seller	–	Requesting Info	…
		Browsing Property Ads	…
		Putting Property Up for Sale	Submit sale request for a *property*
			Set viewing schedule for a *property*
		Consulting Agent	…
		Responding to Offer	Submit offer response on a *property*
			…
		Closing Sale	…
Clerk	Property Mgmt	Creating new Property Record	…
		Updating Property Record	…
		Producing Property Advertisement	…
	Buyer Mgmt	…	…
	Seller Mgmt	…	…
	Agent Mgmt	Assigning Agent to Property	…
		Collecting Reports	…
	Transaction Mgmt	…	…
	Financial Mgmt	…	…
Agent	–	Promoting Property	…
		Inspecting Property	…
		Showing Property	…
		Offering Advice to Parties	…
		Closing Deal	…
		Reporting to Clerk	…
Record Management System	Property Records Mgmt	…	…
	Agent Records Mgmt	…	…
	Customer Records Mgmt	…	…
	Transaction Records Mgmt	…	…
Estate Agency Web-Based System	User Accounts Mgmt	…	…
	Clerical Services Mgmt	Providing Database Management Facilities	…
		Providing Messaging Facilities	…
	Buyer Services Mgmt	Registering/Processing Viewing Requests	…
		Registering/Processing Offers	…
		Providing Messaging Facilities	…
	Seller Services Mgmt	Registering/Processing Sale Requests	…
		Registering/Processing Offer Responses	…
		Providing Messaging Facilities	…
	Agent Services Mgmt	Registering/Processing Viewing Requests	…
		Providing Reporting Facilities	…
		Providing Messaging Facilities	…
	Property Search Mgmt	…	…

Estate Agency

Problem Domain Objects

Property Buyer	A person intending to buy a property; customers visiting the agency are initially considered potential *buyers* and are allowed to search for properties via browsing ads or requesting information from clerks. A *buyer* will have to be registered with the system in order to request a viewing or place an offer on a property. Registration involves opening a *buyer* account for the person and storing their particulars in the database. Registered *buyers* will be assigned a unique *buyer-ID*, which will be used in all their transactions with the agency. A registered *buyer* will have to reregister as a *seller* if ever intending to sell a property.
Property Seller	A person intending to sell a property; a *seller* will have to be registered with the system in order to put a property up for sale. Registration involves opening a *seller* account for the person and storing their particulars in the database. Registered *sellers* will be assigned a unique *seller-ID*, which will be used in all their transactions with the agency. A registered *seller* will have to reregister as a *buyer* if ever intending to buy a property.
Clerk	A person in charge of clerical services at the estate agency offices. As the first point of contact with customers, a *clerk*'s office duties include: providing information on properties, registering customers, entering information into the database, producing property ads, assigning *agents* to properties, and taking care of all paperwork related to transactions.
Agent	A person acting as the representative of the estate agency in performing surveys and valuations, inspecting and promoting properties that have been put up for sale, showing properties to potential *buyers*, acting as mediator and adviser to *sellers* and *buyers*, and arranging the finalization of transactions. *Agents* should report all their activities to the *clerks* involved.
Record Mgmt System	The Computer-based Database Management System maintaining records of properties, customers, agents and transactions. The system also produces reports of the data upon request, including property ads.
Property Ads Board	A Display board for displaying property advertisements.

Flowing Data (or Objects)

Property Ad	A descriptive advertisement of a property detailing the specifications of the property and the price as indicated by the *seller*. *Property Ads* are produced by the *Record Mgmt System* upon request by the *clerks*, who then put them on display on *property ads boards*.
Viewing Request	A request made by a *buyer* to view a specific property. The request is passed to the *agent* assigned to the property, who then contacts the *seller* of the property and arranges for a *viewing schedule*.
Put-Up-for-Sale Request	A request made by a *seller* to the estate agency, giving permission to the agency to act as sale representative on behalf of the *seller* for the promotion and sale of a specific property owned by the *seller*. The request is passed to a *clerk* who assigns an *agent* to the property. The *agent* then inspects and evaluates the property. If confirmed as eligible for sale, the property is then registered in the database and advertised for sale, with the *agent* acting as mediator in all transactions.
Offer	A price offered by a *buyer* on a specific property.
Response to Offer	An acceptance or rejection response by a *seller* to an *offer* made by a *buyer*.
Info Request	A request for *property info* which is submitted to a *clerk* and typically includes values for search criteria – such as location, price range, number of bedrooms, or type.
Property Info	Detailed information on a *property*, which results from a search in the *property ads board* or an *info request* from a *clerk*.
Viewing Schedule	A schedule set by a *seller* for the viewing of a property by a *buyer*, or inspection by an *agent*.
Contract	Legally binding agreement denoting a transaction between *buyers* and *sellers* over a specific *property*. A contract is prepared by a *clerk* and signed by the parties involved.

Figure 86. Partial view of the glossary of terms for the estate agency case study

6.3.1.4 Glossary of Terms

Figure 86 shows a partial view of the Glossary of Terms in the estate agency's Context Model, focusing on problem domain objects and flowing data. Descriptions of typical activities and interactions, and detailed structural information have been left out for sake of brevity.

6.3.2 System Model

The System Model components presented in this section consist of System Object Models; System Interaction Models and the System Features List have been left out for sake of brevity. The System Object Models show the internal structure of the target system, designed as an extension to the existing estate agency structure, i.e. as a separate section consisting of service-attendants and information storage facilities found in conventional offices. Detail has been limited to *put-property-up-for-sale*, *viewing*, and *make-an-offer* processes.

Since System Model components focus on the three system functionalities of *put-property-up-for-sale*, *viewing*, and *make-an-offer*, a more restricted view of the latest version of the Context Object Model has been presented in Figure 87 as the basis for System Object Models.

As already mentioned, the System Model is developed through designing the system as an extension to the existing estate agency. Figure 88 shows an example of one such design. The shaded area in Figure 88 shows the target system, with the objects colour-coded in order to be easily distinguished.

Figure 89 shows an initial version of the System Object Model, with service attendants in charge of interacting with external objects. Custodians have been clearly shown as objects in charge of providing access to passive data objects. Features of the objects – as pertaining to the *put-property-up-for-sale*, *viewing*, and *make-an-offer* functionalities – have been clearly depicted.

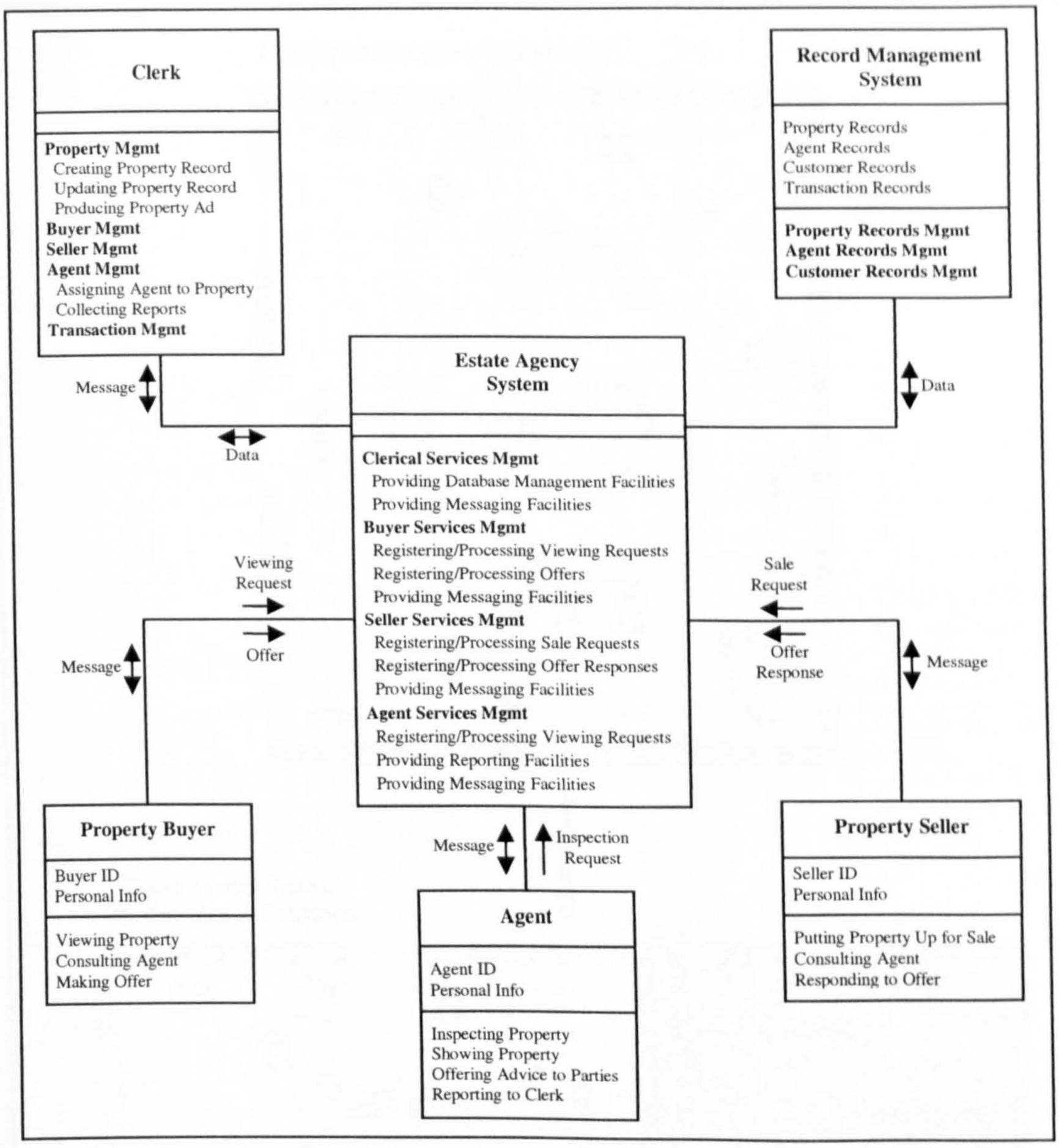

Figure 87. Context Object Model, focusing on the *Put-Up-for-Sale*, *Request-Viewing*, and *Make-Offer* functionalities

As seen in Figure 89, service attendants act as interfaces to the system, interacting with buyers, sellers, agents, and agency clerks. Record-Management-Clerks have been put in charge of retrieving and updating records through interacting with the old database system. A message repository, with its own custodian, has been set up to store the messages through which external objects communicate with each other. Clerical-Services-Attendant objects not only act as interfaces between agency clerks and the system, but also act as intra-system controllers, in that they control access from other attendants to the message repository and Record-Management-Clerks.

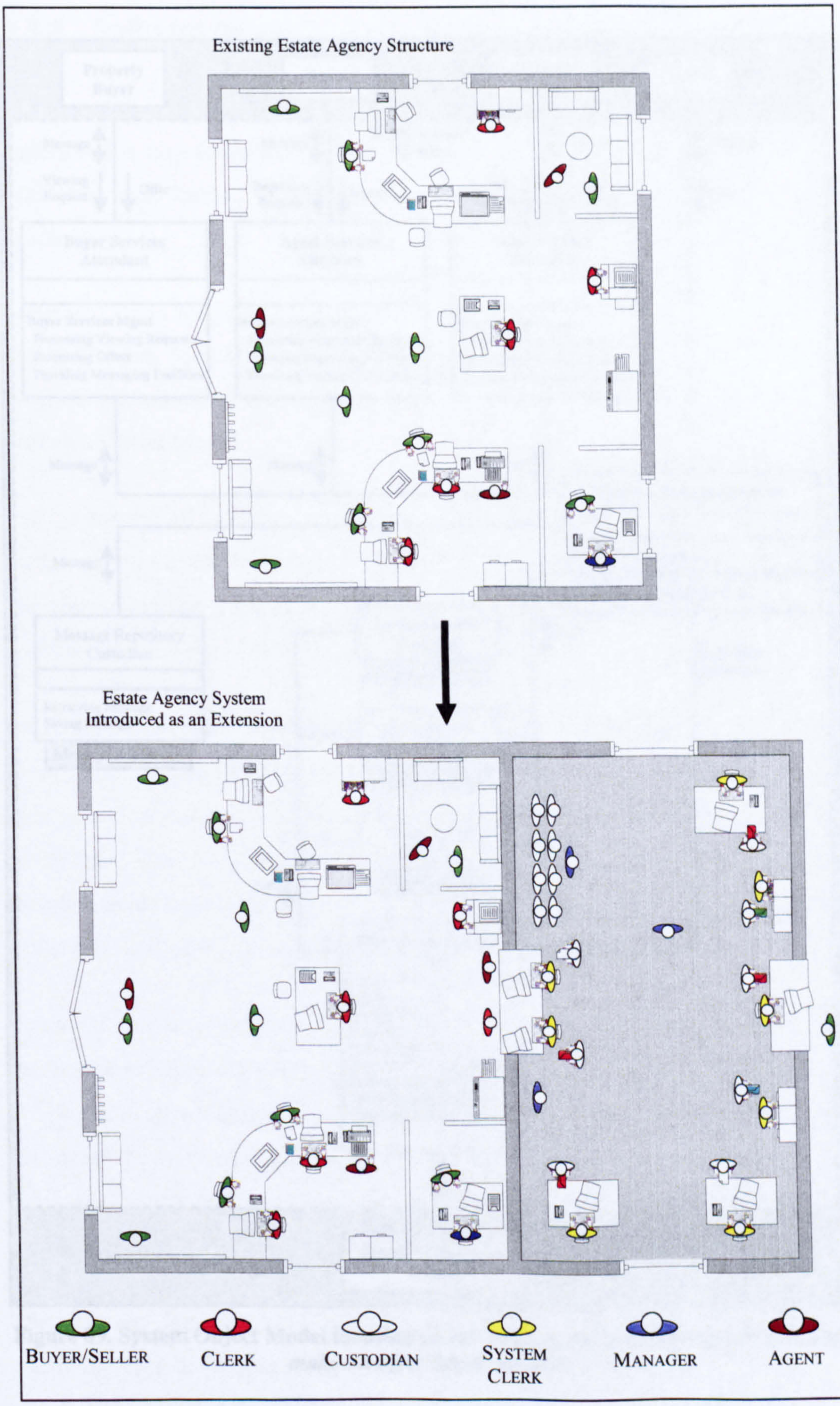

Figure 88. Designing the estate-agency system as an extension to the existing structure

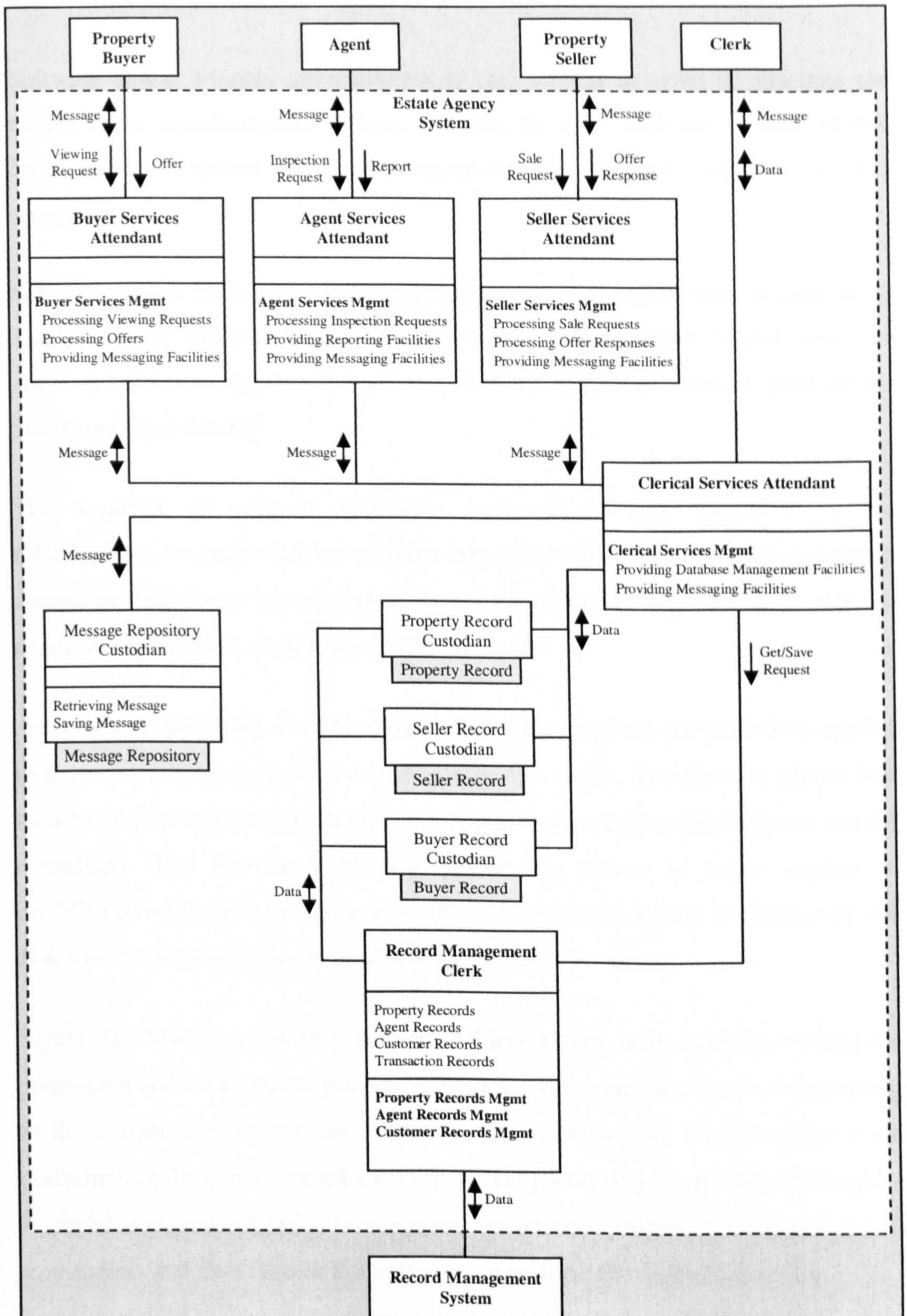

Figure 89. System Object Model focusing on the *viewing*, *put-property-up-for-sale*, and *make-an-offer* functionalities

6.3.3 Software Model

Software Object Models are presented in this section in order to illustrate the pattern-based transformation process. As was the case with the System Model, detail has been limited to *put-property-up-for-sale*, *viewing*, and *make-an-offer* processes.

Software Object Models are produced through iterative application of patterns to System Object Models. Figure 90 shows the System Object Model from the previous section along with the patterns that are applied in order to produce the Software Object Model.

The sequence of pattern application conforms to that prescribed in the methodology, i.e. redistribution patterns take precedence, with refactoring patterns complementing them where needed. Transformations are highlighted in order to emphasize the effects of each pattern on the model.

As shown in this figure, the *Move-Behaviour-Close-to-Data* pattern is first applied to move the relative behaviour to custodians. As a result, the attendant objects lose their behaviour and become simple intermediaries between external objects and the custodians. The *Remove-Middleman* refactoring pattern is hence applied to establish direct links between the custodians and external objects. Alternatively, the *Poltergeist* antipattern can be used with the exact same effect.

Figure 91 shows the resulting Software Object Model, with custodians doing the processing and data objects providing the data. The main functions– as pertaining to the *put-property-up-for-sale*, *viewing*, and *make-an-offer* functionalities – are performed by Property-Record-Custodians, yet processing is typically initiated by Record-Management-Clerks, who retrieve the relevant property record, assign it to a custodian, and then request the custodian to perform the required function.

The next and last step is to merge the custodians with their data objects, thus producing objects encapsulating both state and behaviour. Software Class Models can then be produced, highlighting classifications of objects and their relationships, especially aggregation and generalization/specialization.

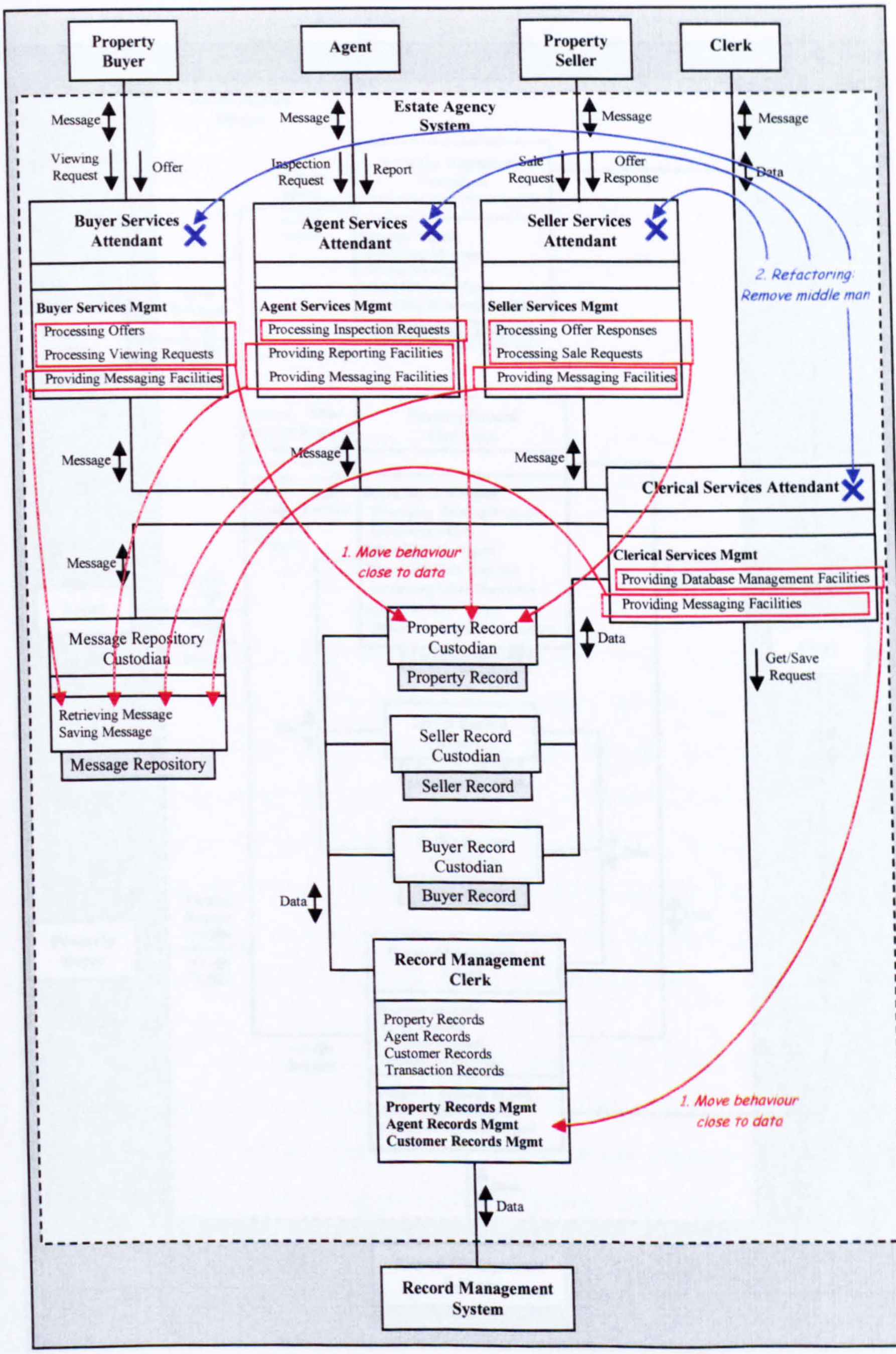

Figure 90. Object Model depicting the major patterns applied to convert the System Object Model into the Software Object Model

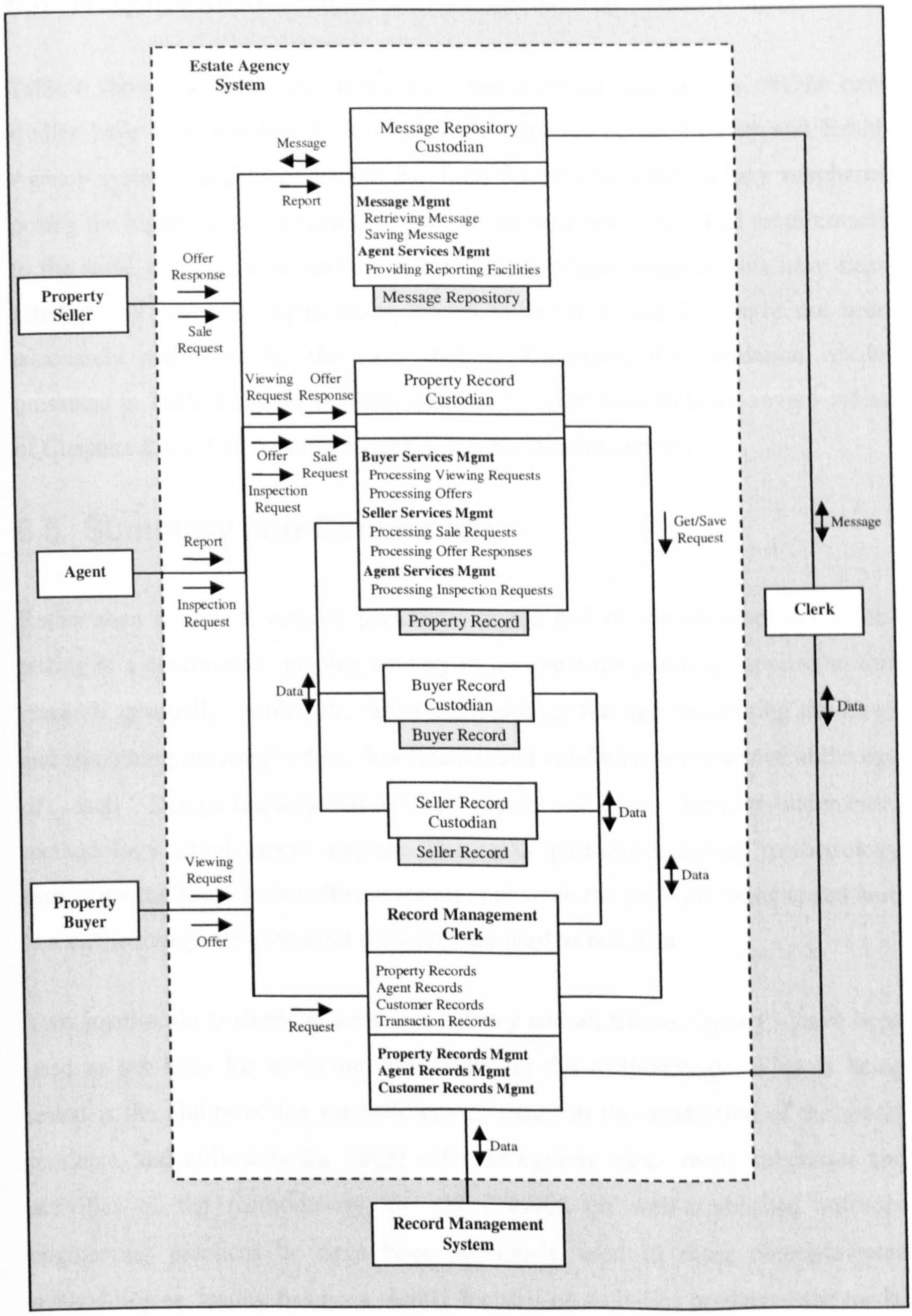

Figure 91. The resulting Software Object Model before merging custodians with data objects

6.4 Requirements-Based Review of the Test Phase

Table 6 shows the final validation results based on the case studies. As the case studies have been conducted on small-scale versions of the Library and Estate Agency systems, and since testing has been focused on methodology subphases posing the highest development risk, test results have not targeted all requirements to the same extent; for instance, while modeling-related requirements have been satisfactorily covered, requirements related to umbrella activities have not been adequately addressed by the case studies. Therefore, the validation results presented in Table 6 should be complemented by requirements-based review tables of Chapters 4 and 5 in order to provide a comprehensive picture.

6.5 Summary and Conclusion

Rather than a one-off activity performed at the end of the development effort, testing is a continuous, ongoing activity in the meta-methodology applied in this research, gradually shaping the target methodology through uncovering the flaws and smoothing the rough edges. Verification and validation is conducted at the end of each Design-Implementation-Test cycle of the iterative-incremental methodology development engine. The tasks performed during methodology testing are the same as in software testing, but since the software being tested here is a methodology, development situations are used as test data.

Two small-scale business systems – a Library and an Estate Agency – have been used as test-beds for verifying and validating the methodology. What is being tested is the ability of the methodology to result in the production of the work-products, and ultimately the target software system. Since many subphases and activities of the methodology are either based on well-established software engineering practices or have been previously used in other object-oriented methodologies, testing has been mainly focused on activities producing the model chain, which due to the novel methods and techniques used in its production, poses the highest development risk.

Verification results have been reported in this chapter as case studies, showing that enacting the final version of the methodology does indeed result in successful

production of design models, which can then be enriched with architectural design detail and class- and method prologues, and ultimately converted into the software system. Validation results, however, are not comprehensive, mainly due to the limited scale and focus of the testing conducted. Validation results should therefore be complemented with the requirements-based review results reported in Chapters 4 and 5.

Table 6. Validation results (continued on next page)

REQUIREMENT		VALIDATION RESULTS	COMMENTS
Process	Clarity, rationality, accuracy, consistency of definition	➢ Flaws detected in modeling activities in the user guide. Instructions refined and perfected in the final version of the user guide.	➢ Also refer to the results of the requirements-based review tabulated at the end of Chapter 5.
	Coverage of generic development lifecycle activities	➢ Analysis and preliminary design activities, especially as pertaining to modeling, refined and improved. ➢ Other activities not addressed by the case studies.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Support for umbrella activities: Risk management	Validation not addressed by the case studies.	➢ Refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Support for umbrella activities: Project management	Validation not addressed by the case studies.	➢ Refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Support for umbrella activities: Quality assurance	➢ Traceability through *features* validated and improved.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Seamlessness and smoothness of transition between phases, stages and activities	➢ Supported throughout the model chain and the relevant activities. Feature-lists are essential for keeping the model conversion activity on track. ➢ Application of design patterns not addressed by the case studies.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Basis in the requirements	➢ The feature-driven approach, starting from the real-world model of the problem domain, validated as capturing the requirements.	
	Testability and Tangibility of artefacts, and traceability to requirements	➢ Fractal modeling, basis in real-world modeling, and seamlessness were validated as enhancing tangibility and traceability. ➢ Tangibility, traceability and the feature-driven nature of artefacts were validated as enhancing testability through improving observability, understandability and simplicity (see [Pressman 2004]).	
	Encouragement of active user involvement	Validation not addressed by the case studies.	➢ Refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Practicability and practicality	➢ Analysis and preliminary design validated as practicable and practical. ➢ Validation of the rest of the methodology has not been addressed by the case studies.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Manageability of complexity	➢ Methodology instructions and guidelines validated as sufficient, simple and easy to follow.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Extensibility / Configurability / Flexibility / Scalability	Validation not addressed by the case studies.	➢ Refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Application scope (*Information Systems*)	➢ Applicability to the two business systems is validated insofar as modeling is concerned. Further validation through enactment in large scale development situations, preferably in an industrial context, is strongly advised. ➢ Applicability to other kinds of information systems has not been addressed by the case studies.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.

Table 6. Contd.

REQUIREMENT			VALIDATION RESULTS	COMMENTS
Modeling Language	Support for object-oriented modeling	Structural – Functional – Behavioural	➢ Structural, functional and behavioural aspects of the systems analyzed and designed in the case studies have been adequately captured; detail has been enough to enable model transformation whilst moving forward along the model chain.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
		Logical to Physical	➢ Analysis and preliminary-design modeling validated as practical. ➢ Architectural and detailed design modeling not addressed by the case studies.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
		At different levels of granularity	➢ Fractal modeling validated as enabling modeling at different levels of granularity.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
		Formal and Informal features	➢ Informal features have been implemented through UML and validated throughout the model chain. ➢ Validation of formal features not addressed by the case studies.	➢ Also refer to the results of requirements-based reviews tabulated at the end of Chapters 4 and 5.
	Provision of strategies and techniques for tackling inconsistency and complexity		➢ Detailed specification of dependencies and consistency guidelines in the user guide has been validated as enhancing simplicity as well as useful for detecting inconsistencies.	

Chapter 7

Conclusion

This chapter presents a summary of the results and contributions of the development effort reported in previous chapters, together with an analysis of the achievements and shortcomings. Several suggestions for furthering this research have also been provided.

7.1 A Summary of Research Results

The following are the main results and contributions of this thesis:

1. A proposed object-oriented software development methodology addressing some of the problems found in existing methodologies; the following are the major contributions of this methodology:
    1.1. A model-based approach to the development of business systems integrating the agile feature-driven merits of the FDD methodology [Palmer and Felsing 2002] with design-based features of third-generation OOSDMs, particularly Catalysis [D'Souza and Wills 1998].
    1.2. A novel modeling approach built into the methodology providing seamless and smooth transition from real-world models of the problem domain to system models and ultimately to design models, using fractal modeling and pattern-based transformation
2. A proposed *meta*-methodology for developing object-oriented software development methodologies based on a software engineering approach; the following are the major contributions of this meta-methodology:
    2.1. An iterative-incremental lifecycle based on the generic activities of software development

2.2. A process-centred template for describing OOSDMs, and the results of applying this template to 24 prominent object-oriented methodologies, process patterns and process metamodels

2.3. A criteria-based analysis method for identifying strengths and weaknesses in object-oriented processes (and thereby defining a set of requirements for the target OOSDM), plus the results of applying this method to the 24 methodologies, process patterns and process metamodels mentioned above

2.4. An iterative-incremental requirements-based design method for producing the blueprint of the target methodology; the method has been designed in such a way as to provide flexible use of a multitude of design approaches.

2.5. A User Guide template for providing a pragmatic description of object-oriented software development methodologies

7.2 Objectives Achieved

As expected in any software engineering effort, the objectives of this methodology development effort are manifested in the set of requirements identified through problem-domain analysis (reported in Chapter 3). Identifying the features that address the requirements is facilitated by the fact that the list of requirements also provides extensive coverage of methods and techniques found in existing methodologies that can be adopted to satisfy the requirements. By focusing on requirements and integrating ideas from existing methodologies, a methodology has been produced that is relatively simple, makes use of established techniques in a coherent and intuitive fashion, and addresses key issues of practicality, tangibility and seamlessness, all of which contribute to its usability by practitioners.

Table 7 shows the requirements addressed in the production of the target methodology, detailing the methodology features through which the requirements have been met. As shown in this table, some requirements (i.e. *extensibility* and *application scope*) have been satisfied partially, and will therefore be further discussed in the next section.

Table 7. Methodology requirements that have been addressed

REQUIREMENT			DETAILS OF HOW THE REQUIREMENT HAS BEEN ADDRESSED
Process	Clarity, rationality, accuracy, consistency of definition		➢ Addressed through the structure of the User Guide template used for defining the methodology, which provides a means for defining the methodology in a top-down fashion, guiding and focusing the definition process on specifying appropriate levels of detail as relevant to the three basic, complementary views of the methodology: *Process-Based*, *Work-Product(Artefact)-Based*, and *Role(Producer)-Based*.
	Coverage of generic development lifecycle activities		➢ Addressed at the lifecycle, subprocess and task levels ➢ Maintenance has not been covered in full in order to avoid commitment to a specific maintenance strategy, allowing an iteration of the development process if appropriate. Maintenance Planning, however, is conducted during the Transition subprocess.
	Support for umbrella activities	Risk management	➢ Addressed through iterative-incremental development, preliminary analysis, risk-based planning, continuous verification and validation, regular product/plan reviews, and continuous integration.
		Project management	➢ Addressed through project planning, scheduling and control activities incorporated in the subprocesses of the methodology, and provisions for continual review and revision of the plans throughout the process.
		Quality assurance	➢ Addressed through regular technical reviews, continuous verification and validation during iterative development, requirements traceability incorporated in the feature-based model chain, and the feature-driven nature of activities and tasks.
	Seamlessness and smoothness of transition between phases, stages and activities		➢ Addressed through the artefact chain and the iterative-incremental development engine: fractal modeling, system design through homogeneous extension, and pattern-based transformation provide a seamless and smooth modeling process – having the same effect on the subprocesses built around the artefact chain, while plan-based feature-driven cyclic development provides smooth transition between the design and build activities of the iterative development engine.
	Basis in the requirements		➢ Addressed through *features*: i.e. the feature-based description of the requirements, and the feature-driven approach governing all development activities throughout the process.
	Testability and tangibility of artefacts, and traceability to requirements		➢ Addressed via basis in real-world modeling, fractal modeling, gradual seamless transformation of artefacts through analysis and design, initial design of the system as a homogeneous extension to the problem domain, and the feature-based nature of artefacts throughout the process.
	Encouragement of active user involvement		➢ Addressed through constant participation of user representatives throughout the process, a feature adapted from agile development.
	Practicability and practicality		➢ Addressed through avoiding complexity at all levels, adhering to risk-based development, incorporating project management activities, making use of methods and techniques already tested in existing methodologies, and using techniques and strategies for focusing the development (e.g. feature-driven model chain).
	Manageability of complexity		➢ Addressed through the hierarchical structure of the methodology definition, and also via keeping subprocesses, activities and tasks cohesive and easy to understand.
	Scalability / Extensibility		➢ Scalability was addressed through plan-based, model-driven and architecture-centric process. ➢ Extensibility was addressed through keeping the process as a cohesive core organized around a model chain. Further work is required on defining extension points and extension mechanisms.
	Application scope (*Information Systems*)		➢ Partially addressed through concentrating on business systems as commonly encountered information systems. ➢ Applicability to other kinds of information systems has not been explored.
Modeling Language	Support for object-oriented modeling	Structural – Functional – Behavioural	➢ Addressed through using appropriate UML-based diagrams at different levels: structural modeling is addressed through the use of Object Models and Class Models, functional features are modeled in data-flow-oriented Object Models, and Activity Diagrams and Interaction Diagrams are used for modeling behavioural features.
		Logical to Physical	➢ Addressed through the model chain, starting at the problem-domain level and proceeding to detailed design.
		At different levels of granularity	➢ Addressed through fractal modeling at different granularity levels (Enterprise level – System level – Subsystem/Package level – Inter-object level – Intra-object level).
	Provision of strategies and techniques for tackling inconsistency and complexity		➢ Inconsistency prevention and resolution is implemented through detailed specification of dependencies and consistency guidelines. ➢ Complexity is tackled through fractal modeling, the layered structure of the models, and UML's complexity management features.

The hybrid iterative process used for designing the methodology is based on prioritizing the requirements, both as a complexity management technique and a flexibility enhancement feature (as reported in Chapter 4). As a result, requirements relevant to the core target areas in object-oriented methodologies where improvement is needed (*Compactness*, *Extensibility*, *Traceability to Requirements*, *Consistency*, *Testability of Artefacts*, *Tangibility of Artefacts*, and *Visible Rationality*; as listed in Chapter 1) have been assigned higher priority during the first few crucial iterations of the design engine. This explains why fractal modeling and the seamless model chain, for example, have been assigned such a pivotal role in the methodology.

7.3 **Shortcomings**

A number of requirements have not been addressed, mainly due to the need for further practical experience with the current version of the methodology, or in order to avoid undue complexity. Table 8 shows the requirements that have not been addressed in the current version of the methodology, along with details of why implementation has not been considered and ideas for future enhancements to the methodology in order to satisfy the requirements.

Table 8. Methodology requirements that have not been addressed in the current version of the methodology

REQUIREMENT			DETAILS	IDEAS FOR IMPLEMENTATION
Process	Configurability		➢ Practical application of the methodology to industrial projects, i.e. based on industrial scenarios and taking place in an industrial context, is required in order to identify potentials for variation, and thereby determine ways to parameterize the process; therefore, configurability has not been incorporated at this stage.	➢ In cases where FDD has been shown to be operationally useful, an imploded version of the implemented methodology with the model chain deemphasized seems to be of practical merit. The model chain is therefore a potential area for parameterization.
	Flexibility		➢ Although useful methods for implementing flexibility already exist, it has been decided to keep the methodology in its present form until data from practical experience with the current version of the methodology has been gathered.	➢ Implementing flexibility (dynamic configurability) via incorporating process review sessions (similar to those seen in Crystal-Clear [Cockburn 2004]) seems a promising method.
Modeling Language	Support for object-oriented modeling	Formal features	➢ Formal features not considered in the present version in order to keep the methodology definition as simple as possible.	➢ Incorporating UML/OCL.

In addition to requirements not addressed in the current version of the methodology, there are also requirements that admittedly have not been fully met. As shown in Table 7, these requirements include:

- *Extensibility*: extension points and extension mechanisms have not been defined, mainly due to the need for further practical experience with the methodology in order to identify potential ways of extending the process. Exploring the applicability of the methodology to various types of systems, at different levels of criticality and scale, is necessary in order to gain adequate knowledge of extensions required and the feasibility of their incorporation into the methodology. However, designing the methodology as a core – avoiding complexity and undue commitment to unessential and complementary methods and techniques at all levels – is definitely instrumental in allowing extensibility to be implemented in future versions of the methodology.
- *Application Scope*: although the methodology is targeted at Information Systems, the focus has been mostly limited to Business Systems; hence, behavioural modeling features are lacking. As an indicative example, state-dependent behaviour – which is the distinguishing feature of real-time aspects of information systems – is not captured. Improving the methodology in this regard requires the incorporation of State-Transition modeling into the modeling process, and emphasizing the mechanisms already available in UML for expressing timing constraints.

7.4 Suggestions for Further Research

There are several potential courses for furthering or complementing the research reported in this dissertation, some of which are listed below:

- **Engineering variants of the methodology targeting other types of systems, e.g. safety-critical:** the pattern-based model transformation approach can potentially be used for developing system types other than Business systems, yet the current model chain lacks formal modeling features, and its applicability to systems requiring more precise behavioural modeling is not certain. Variants of the methodology should also address

the use of specialized sets of design patterns for model transformation, especially for introducing context-specific structure and behaviour in the models.

- **Applying the methodology to case studies of larger scope:** the sample systems used for the verification and validation of the methodology have been intentionally selected to be small in scope, and the focus has mostly been on the practicability of the modeling approach. In order to test the scalability of the methodology, larger systems should be considered for testing.
- **Expressing the methodology and meta-methodology processes in a Process Modeling Language (PML):** this will be especially useful for static verification of the methodology and/or enactment in a Process-centred Software Engineering Environment (PSEE) [Ambriola et al. 1997, Barthelmess 2003].
- **Empirical analysis of the usability of the methodology:** an empirical analysis will also complement the testing results already obtained: although many components of the methodology have already been used in existing methodologies, the overall effectiveness of many aspects of the methodology – especially the umbrella activities and the actual development tasks of the iterative-incremental engine – cannot be properly tested without enactment in actual development projects based on industrial scenarios and taking place in an industrial context. Practical experience will also help identify potential for making enhancements to the methodology, e.g. to implement features such as configurability and extensibility.
- **Comparison of the methodology to other OOSDMs:** the aim is to assess the contributions of the methodology using existing comparison frameworks. The criteria-based evaluation method presented in this thesis is of little use for this purpose, since it has already been used for defining methodology requirements, based on which the methodology was developed in the first place.
- **Application of the meta-methodology to the development of other methodology types:** the meta-methodology is general enough to be used for developing any type of methodology, provided that there are enough

base methodologies to provide the requirements and process components. Agent-oriented development, for example, is a suitable candidate, since the field is mature enough to provide established development methodologies for analysis and integration.

Abbreviations

ADFD	Action Data Flow Diagram
ADISSA	Architectural Design of Information Systems based on Structured Analysis
ASD	Adaptive Software Development
BON	Business Object Notation
CORBA	Common Object Request Broker Architecture
CRC	Class-Responsibility-Collaborator
DFD	Data Flow Diagram
DM	Dynamic Model
DMC	Data Management Component
DSDM	Dynamic Systems Development Method
ERD	Entity-Relationship Diagram
EUP	Enterprise Unified Process
FDD	Feature-Driven Development
FM	Functional Model
FOOM	Functional and Object-Oriented Methodology
HIC	Human Interaction Component
IDL	Interface Definition Language
JAD	Joint Application Development
MDA	Model-Driven Architecture
OBM	Object-Behaviour Model

OCL	Object Constraint Language
OIM	Object Interaction Model
OM	Object Model
OMG	Object Management Group
OML	OPEN Modeling Language
OMT	Object Modeling Technique
OO	Object-Oriented
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OODLE	Object Oriented Design LanguagE
OOP	Object-Oriented Programming
OOSDM	Object-Oriented Software Development Methodology
OOSE	Object-Oriented Software Engineering
OOSP	Object-Oriented Software Process
OPD	Object Process Diagram
OPEN	Object-oriented Process, Environment, and Notation
OPF	OPEN Process Framework
OPL	Object-Process Language
OPM	Object-Process Methodology
ORM	Object-Relationship Model
OSA	Object-oriented Systems Analysis
PDC	Problem Domain Component
PIM	Platform-Independent Model
PML	Process Modeling Language

PSEE	Process-centred Software Engineering Environment
PSM	Platform-Specific Model
Q/A	Quality Assurance
RAD	Rapid Application Development
RDD	Responsibility-Driven Design
RUP	Rational Unified Process
SA	Structured Analysis
SD	Structured Design
SDM	Software Development Methodology
SPEM	Software Process Engineering Metamodel
SQL	Structured Query Language
SSADM	Structured Systems Analysis and Design Method
TMC	Task Management Component
UI	User Interface
UML	Unified Modeling Language
USDP	Unified Software Development Process
USPM	Unified Software Process Metamodel
XP	eXtreme Programming

References

ABRAHAMSSON, P., SALO, O., RONKAINEN, J., AND WARSTA, J. 2002. *Agile Software Development Methods: Review and Analysis*. VTT Publications, Oulu, Finland.

ABRAHAMSSON, P., WARSTA, J., SIPONEN, M. T., AND RONKAINEN, J. 2003. New directions on agile methods: A comparative analysis. In *Proceedings of the International Conference on Software Engineering – ACM/ICSE 2003*, 244-254.

ALABISO, B. 1988. Transformation of dataflow analysis models to object oriented design. In *Proceedings of ACM/OOPSLA'88 Conference*, 335-353.

AMBLER, S. W. 1998a. *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, New York, NY.

AMBLER, S. W. 1998b. An introduction to process patterns. Published on the Web at: *http://www.ambysoft.com/processPatterns.pdf*, visited in April 2006.

AMBLER, S. W. 1999. *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. Cambridge University Press, New York, NY.

AMBLER, S. W. 2005. Introduction to the Enterprise Unified Process. Available at: *http://www.enterpriseunifiedprocess.info/downloads/eupIntroduction.pdf*, visited in April 2006.

AMBLER, S. W., AND CONSTANTINE, L. L. 2000a. *The Unified Process Inception Phase*. CMP Books, Gilroy, CA.

AMBLER, S. W., AND CONSTANTINE, L. L. 2000b. *The Unified Process Elaboration Phase*. CMP Books, Gilroy, CA.

AMBLER, S. W., AND CONSTANTINE, L. L. 2000c. *The Unified Process Construction Phase*. CMP Books, Gilroy, CA.

AMBLER, S. W., AND CONSTANTINE, L. L. 2002. *The Unified Process Transition and Production Phase*. CMP Books, Gilroy, CA.

AMBLER, S. W., NALBONE, J., AND VIZDOS, M. J. 2005. *The Enterprise Unified Process: Extending the Rational Unified Process*. Prentice-Hall, Englewood Cliffs, NJ.

AMBRIOLA, V., CONRADI, R., AND FUGGETTA, A. 1997. Assessing process-centered software engineering environments. *ACM Transactions on Software Engineering and Methodology 46,* 3 (July), 283–328.

AVISON, D. E., AND FITZGERALD, G. 2003. Where now for development methodologies? *Communications of the ACM 46*, 1 (January), 79-82.

BARTHELMESS, P. 2003. Collaboration and coordination in process-centered software development environments: A review of the literature. *Information and Software Technology 45*, 13, 911-928.

BECK, K. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Mass.

BECK, K., AND ANDRES, C. 2004. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, Reading, Mass.

BECK, K., ET AL. 2001. Manifesto for agile software development. Published on the Web at: *http://agilemanifesto.org*, visited in November 2004.

BOEHM, B. 2006. Some future trends and implications for systems and software engineering processes. *Systems Engineering 9*, 1 (Spring), 1-19.

BOEHM, B., AND TURNER, R. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, Reading, Mass.

BOEHM, B., AND TURNER, R. 2005. Management challenges to implementing agile processes in traditional development organizations. *IEEE Software* 22, 5 (September/October), 30-39.

BOOCH, G. 1986. Object-oriented development. *IEEE Transactions on Software Engineering 12*, 2 (February), 211-221.

BOOCH, G. 1991. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA.

BOOCH, G. 1994. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA.

BOOCH, G., MARTIN, R. C., AND NEWKIRK, J. 1998. *Object Oriented Analysis and Design with Applications*, 2nd ed. (Unpublished). Addison Wesley, Reading, Mass. The unpublished chapter on RUP and dX is available on the Web at: http://www.objectmentor.com/resources/articles/RUPvsXP.pdf.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. *Unified Modeling Language—User's Guide*. Addison-Wesley, Reading, Mass.

BRINKKEMPER, S. 1996. Method engineering: engineering of information systems development methods and tools. *Information and Software Technology 38*, 4, 275-280.

BROWN, W. J., MALVEAU, R. C., MCCORMICK, H., AND MOWBRAY, T. 1998. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York, NY.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern Oriented Software Architecture: A System of Patterns.* Wiley, New York, NY.

CAPRETZ, L. F. 2003. A brief history of the object-oriented approach. *ACM SIGSOFT Software Engineering Notes 28*, 2 (March).

COAD, P., LEFEBVRE, E., AND DE LUCA, J. 1999. *Java Modeling in Color with UML: Enterprise Components and Process.* Prentice-Hall, Englewood Cliffs, NJ.

COAD, P., AND YOURDON, E. 1991a. *Object-Oriented Analysis,* 2nd ed. Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ.

COAD, P., AND YOURDON, E. 1991b. *Object-Oriented Design.* Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ.

COCKBURN, A. 1998. *Surviving Object-Oriented Projects: A Manager's Guide.* Addison-Wesley, Reading, Mass.

COCKBURN, A. 2001. *Agile Software Development: Software through People.* Addison-Wesley, Reading, Mass.

COCKBURN, A. 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams.* Addison-Wesley, Reading, Mass.

COLEMAN, D., ARNOLD, P., BODOFF, S., DOLLIN, C., GILCHRIST, H., HAYES, F., AND JEREMAES, P. 1994. *Object-Oriented Development: The Fusion Method.* Prentice-Hall, Englewood Cliffs, NJ.

COLEMAN, D., JEREMAES, P., AND DOLLIN, C. 1992. *Fusion: A Systematic Method for Object-Oriented Development.* Hewlett Packard Laboratories.

COOK, S., AND DANIELS, J. 1994. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice-Hall, Englewood Cliffs, NJ.

COPLIEN, J. O. 1994. A development process generative pattern language. In *Proceedings of the First Annual Conference on Pattern Languages of Programming (PLoP)*.

CORAM, M. AND BOHNER, S. 2005. The impact of agile methods on software project management. In *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (Ecbs'05)*, 363-370.

CRYSTAL METHODOLOGIES ORGANIZATION. 2001. Adaptive Software Development process framework. PowerPoint presentation available on the Web at: *http://crystalmethodologies.org/processes/asd/asdprocess.ppt*, visited in January 2003.

D'SOUZA, D. F., AND WILLS, A. C. 1995. Catalysis – practical rigor and refinement: Extending OMT, Fusion, and Objectory. Available on the Web at: *http://www.catalysis.org/publications/papers/1995-catalysis-fusion.pdf*, visited in April 2006.

D'SOUZA, D. F., AND WILLS, A. C. 1998. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass.

DEMARCO, T. 1978. *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, NJ.

DEMEYER, S., DUCASSE, S., AND NIERSTRASZ, O. 2003. *Object-Oriented Reengineering Patterns*. Morgan-Kauffman, San Francisco, CA.

DERR, K.W. 1995. *Apply OMT: A Practical Step-by-step Guide to Using the Object Modeling Technique*. Cambridge University Press, New York, NY.

DORI, D. 1995. Object-process analysis: Maintaining the balance between system structure and behaviour. *Journal of Logic and Computation 5*, 2 (April), 227-249.

DORI, D. 2002a. *Object-Process Methodology: A Holistic Systems Paradigm.* Springer, Berlin-New York.

DORI, D. 2002b. Why significant UML change is unlikely. *Communications of the ACM 45*, 11 (November), 82-85.

DOWNS, E., CLARE, P., AND COE, I. 1988. *Structured Systems Analysis and Design Method: Application and Context.* Prentice-Hall International, UK.

DSDM CONSORTIUM. 2003. *DSDM: Business Focused Development*, 2nd ed. J. Stapleton, Ed. Addison-Wesley, Reading, Mass.

EMBLEY, D. W., KURTZ, B. D., AND WOODFIELD, S. N. 1992. *Object-Oriented Systems Analysis: A Model-Driven Approach.* Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ.

ENGELS, G., AND GROENEWEGEN, L. 2000. Object-oriented modeling: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering – ACM/ICSE 2000*, 103-116.

FINKELSTEIN, A., AND KRAMER, J. 2000. Software engineering: a roadmap. *Proceedings of the Conference on the Future of Software Engineering – ACM/ICSE 2000*, 3-22.

FIRESMITH, D., AND HENDERSON-SELLERS, B. 2001. *The OPEN Process Framework: An Introduction.* Addison-Wesley, Reading, Mass.

FOWLER, M. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, Mass.

FOWLER, M. 2004. Model Driven Architecture. Published on the Web at: *http://martinfowler.com/bliki/ModelDrivenArchitecture.html*, visited in April 2006.

FUGGETTA, A. 2000. Software process: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering – ACM/ICSE 2000*, 25-34.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, Reading, Mass.

GE, X., PAIGE, R. F., POLACK, F. A. C., CHIVERS, H., AND BROOKE, P. J. 2006. Agile development of secure web applications. In *Proceedings of the International Conference on Web Engineering (ICWE2006).*

GERVAIS, M. P. 2002. Towards an MDA-oriented methodology. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC2002)*, 265-270.

GRAHAM, I. 2001. *Object-oriented Methods: Principles and Practice,* 3rd ed. Addison-Wesley, Reading, Mass.

GRAHAM, I., HENDERSON-SELLERS, B., AND YOUNESSI, H. 1997. *The OPEN Process Specification.* Addison-Wesley, Reading, Mass.

HARMSEN, A. F. 1997. *Situational Method Engineering*. Moret Ernst & Young.

HENDERSON-SELLERS, B. 2003. Method engineering for OO systems development. *Communications of the ACM 46*, 10 (October), 73-78.

HENDERSON-SELLERS, B., AND GRAHAM, I. 1996. OPEN: Toward method convergence? *IEEE Computer 29*, 4 (April), 86-89.

HIGHSMITH, J. 1997. Messy, exciting, and anxiety-ridden: Adaptive software development. *American Programmer 10*, 4 (April), 23-29.

HIGHSMITH, J. 2000a. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House, New York, NY.

HIGHSMITH, J. 2000b. Retiring lifecycle dinosaurs. *Software Testing* and *Quality Engineering* 2, 4 (July/August), 22-28.

HIGHSMITH, J. 2002. *Agile Software Development Ecosystems*. Addison-Wesley, Reading, Mass.

HODGE, L. R., AND MOCK, M. T. 1992. A proposed object-oriented development methodology. *Software Engineering Journal 7*, 2 (March), 119-129.

ISODA, S. 2001. Object-oriented real-world modeling revisited. *Journal of Systems and Software 59*, 2 (November), 153-162.

JACOBSON, I. 1987. Object-oriented development in an industrial environment. In *Proceedings of ACM/OOPSLA'87*, 183-191.

JACOBSON, I., BOOCH, G., AND RUMBAUGH, G. 1999. *Unified Software Development Process*. Addison-Wesley, Reading, Mass.

JACOBSON, I., CHRISTERSON, M., JONSSON, P., AND ÖVERGAARD, G. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, Mass.

KABELI, J., AND SHOVAL, P. 2003. Software analysis process – which order of activities, is preferred? An experimental comparison using FOOM methodology. In *Proceedings of the IEEE International Conference on Software-Science, Technology and Engineering*, 111-122.

KARAM, G. M., AND CASSELMAN, R. S. 1993. A cataloging framework for software development methods. *IEEE Computer 26*, 2 (February), 34-45.

KROLL, P., AND KRUCHTEN, P. 2003. *The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process*. Addison-Wesley, Reading, Mass.

KRUCHTEN, P. 2001. A process engineering metamodel. Available on the Web at: *http://www.forsoft.de/zen/sdpp02/papers/Kruc01.pdf*, visited in April 2006.

KRUCHTEN, P. 2003. *Rational Unified Process: An Introduction*, 3rd ed. Addison-Wesley, Reading, Mass.

KUMAR, K., AND WELKE, R. J. 1992. Method engineering: a proposal for situation-specific methodology construction. In *Systems Analysis and Design: A Research Agenda*. Cotterman, W. W., and Senn, J. A. Eds. Wiley, 257-268.

LANG, N. 1993. Shlaer-Mellor object-oriented analysis rules. *Software Engineering Notes 18*, 1 (January), 54-58.

LANO, K., FRANCE, R., AND BRUEL, J. 2000. A semantic comparison of Fusion and Syntropy. *The Computer Journal 43*, 6, 451-468.

MEYER, B. 1997. *Object-oriented Software Construction*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ.

MOCK, M.T., AND HODGE, L. R. 1992. An exercise to prototype the object-oriented development process. *Software Engineering Journal 7*, 2 (March), 114 – 118.

MONARCHI, D. E., AND PUHR, G. I. 1992. A research typology for object-oriented analysis and design. *Communications of the ACM 35*, 9 (September), 35-47.

NERSON, J. 1992, Applying object-oriented analysis and design. *Communications of the ACM 35*, 9 (September), 63-74.

NERUR, S., MAHAPATRA, R., AND MANGALARAJ, G. 2005. Challenges of migrating to agile methodologies. *Communications of the ACM 48*, 5 (May), 73-78.

NUSEIBEH, B., AND EASTERBROOK, S. 2000. Requirements engineering: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering – ACM/ICSE 2000*, 35-46.

OMG. 2001. *Model Driven Architecture (MDA)*. Object Management Group (OMG).

OMG. 2002. *Software Process Engineering Metamodel Specification (v1.0)*. Object Management Group (OMG).

OMG. 2003. *Unified Modeling Language Specification (v1.5)*. Object Management Group (OMG).

OMG. 2004. *Unified Modeling Language Specification (v2.0)*. Object Management Group (OMG).

OPEN CONSORTIUM. 2000. What is OPEN? Published on the Web at: *http://www.open.org.au/Introduction/main.html*, visited in April 2006.

OSTERWEIL, L. J. 1987. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, 2-13.

OSTERWEIL, L. J. 1997. Software processes are software too, revisited: An invited talk on the most influential paper of ICSE 9. In *Proceedings of the 19th International Conference on Software Engineering*, 540-548.

PAIGE, R., AND OSTROFF, J. S. 2002, The single model principle. *Journal of Object Oriented Technology 1*, 5 (November-December), 63-81.

PALMER, S. R., AND FELSING, J. M. 2002. *A Practical Guide to Feature-Driven Development*. Prentice-Hall, Englewood Cliffs, NJ.

PRESSMAN, R. S. 2004. *Software Engineering: A Practitioner's Approach,* 6th ed. McGraw-Hill, New York, NY.

RALYTÉ, J., DENECKÉRE, R., AND ROLLAND, C. 2003. Towards a generic model for situational method engineering. In *Proceedings of CAiSE 2003 (LNCS 2681)*, 95-110.

RALYTÉ, J., ROLLAND, C., AND DENECKÉRE, R. 2004. Towards a meta-tool for change-centric method-engineering: A typology of generic operators. In *Proceedings of CAiSE 2004 (LNCS 3084)*, 202-218.

RAMSIN, R. 1995. Detailed Inspection and Evaluation of Object-Oriented Software Development Methodologies. MSc Thesis (in Persian). Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. Submitted in February 1995.

RAMSIN, R., AND PAIGE, R.F. 2004. Process-centred review of object-oriented software development methodologies. Technical Report YCS-2004-381. University of York, York, UK.

REENSKAUG, T., WOLD, P., AND LEHNE, O. 1996. *Working with Objects: The OOram Software Engineering Method.* Manning Publications, Greenwich, CT.

RUMBAUGH, J. 1994. Getting started: Using use cases to capture requirements. *Journal of Object-Oriented Programming 7*, 5 (September), 8-23.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ.

SCHUH, P. 2005. *Integrating Agile Development in the Real World.* Charles River Media, Hingham, Mass.

SCHWABER, K. 1995. SCRUM development process. In *Proceedings of the ACM/OOPSLA'95 Workshop on Business Object Design and Implementation*, also available on the Web at: http://jeffsutherland.com/oopsla/schwapub.pdf.

SCHWABER, K. 2004. *Agile Project Management with Scrum.* Microsoft Press, Redmond, WA.

SCHWABER, K., AND BEEDLE, M. 2001. *Agile Software Development with Scrum.* Prentice-Hall, Englewood Cliffs, NJ.

SEIDEWITZ, E., AND STARK, M. 1986. Towards a general object-oriented software development methodology. In *Proceedings of the First International Conference on Ada Programming Language Applications*, 1-14.

SHLAER, S., AND MELLOR, S. J. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data.* Prentice-Hall, Englewood Cliffs, NJ.

SHLAER, S., AND MELLOR, S. J. 1992. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, Englewood Cliffs, NJ.

SHLAER, S., AND MELLOR, S. J. 1996. The Shlaer-Mellor method. Published on the Web at: *http://www.nrt.se/nrt/PTpdf/smmethod.pdf*, visited in January 2003.

SHOVAL, P. 1988. ADISSA: Architectural design of information systems based on structured analysis. *Information Systems 13*, 2, 193-210.

SHOVAL, P., AND KABELI, J. 2001. FOOM: Functional- and object-oriented analysis and design of information systems: An integrated methodology. *Journal of Database Management 12*, 1 (January-March), 15-25.

SIEGEL, J., AND OMG. 2001. *Developing in OMG's Model Driven Architecture (MDA).* Object Management Group (OMG).

SOMMERVILLE, I. 2004. *Software Engineering*, 7th ed. Addison-Wesley, Reading, Mass.

THOMAS, D. 2004. MDA: Revenge of the modelers or UML utopia. *IEEE Software 21*, 3 (May/June), 22-24.

TURK, D., FRANCE, R., AND RUMPE, B. 2005. Assumptions underlying agile software-development processes. *Journal of Database Management 16*, 4 (October-December), 62-87.

WALDEN, K., AND NERSON, J. 1995. *Seamless Object-Oriented Software Architecture*. Prentice-Hall, Englewood Cliffs, NJ.

WALKER, I. J. 1992. Requirements of an object-oriented design method. *Software Engineering Journal 7*, 2 (March), 102-113.

WEBSTER, S. 1996. On the evolution of OO methods. Available on the Web at: *http://dec.bournemouth.ac.uk/staff/swebster/OO_meth_evol_complete.html*, visited in January 2003.

WELLS, D. 2003. Extreme programming: A gentle introduction. Published on the Web at: *http://www.extremeprogramming.org*, visited in April 2006.

WIRFS-BROCK, R., AND MCKEAN, A. 2002. *Object Design: Roles, Responsibilities and Collaborations*. Addison-Wesley, Reading, Mass.

WIRFS-BROCK, R., WILKERSON, B., AND WIENER, R. 1990. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ.

WORDSWORTH, J. 1992. *Software Development with Z: Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley, Reading, Mass.

YOURDON, E., AND CONSTANTINE, L. L. 1979. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ.