Process-Centered Review of Object Oriented Software Development Methodologies

RAMAN RAMSIN and RICHARD F. PAIGE

University of York

3

We provide a detailed review of existing object-oriented software development methodologies, focusing on their development processes. The review aims at laying bare their core philosophies, processes, and internal activities. This is done by using a process-centered template for summarizing the methodologies, highlighting the activities prescribed in the methodology while describing the modeling languages used (mainly diagrams and tables) as secondary to the activities. The descriptions produced using this template aim not to offer a critique on the methodologies and processes, but instead provide an abstract and structured description in a way that facilitates their elaborate analysis for the purposes of improving understanding, and making it easier to tailor, select, and evaluate the processes.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.9 [**Software Engineering**]: Management—*Life cycle*; K.2 [**History of Computing**]:—*Software*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management—*Systems analysis and design; systems development*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development; Software process*

General Terms: Design, Management

Additional Key Words and Phrases: Object-oriented methodologies, seminal methodologies, agile methods, integrated methodologies, methodology engineering

ACM Reference Format:

Ramsin, R. and Paige, R. F. 2008. Process-centered review of object oriented software development methodologies. *ACM Comput. Surv. 40,* 1, Article 3 (February 2008), 89 pages. DOI = 10.1145/1322432.1322435 http://doi.acm.org/10.1145/1322432.1322435

1. INTRODUCTION

The study of software engineering has been evolving over the past thirty years, but it has never completely solved the software crisis. As an integral part of the discipline of software engineering, software development methodologies have also evolved, from the 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 methodologies have managed to survive, and indeed, evolve.

Authors' addresses: R. Ramsin, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran; email: ramsin@sharif.edu; R. F. Paige, Department of Computer Science, University of York, York, YO10 5DD, UK; email: paige@cs.york.ac.uk.

Object-oriented software development methodologies have been around for close to two decades, but many of the problems associated with these methodologies at the beginning still remain unresolved. There is still a proliferation of methodologies, and it is difficult to choose from among them. Methodologies are treated—in some cases, sensibly—as products; many are even marketed as such (e.g., RUP [Kruchten 2003] and DSDM [DSDM Consortium 2003]). Treating methodologies as merchandise can result in redundant decorative clutter, attractive yet obscuring wrappings, and uninformative, sometimes even advertisement-like descriptions, which are useful when attempting to sell the methodology but which distract from explaining and understanding its underlying process. Furthermore, methodologies are inherently complex; even methodologists who 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. And often, methodologists are not objective and impartial towards their own creations (and should not be expected to); features stressed by methodologists are often not the essential ones for solving the problems of the methodology's domain, but are those that the methodologist sees as important or unique. All of these further add to the confusion of the developer planning to use the methodology.

This article presents a comprehensive review of object-oriented software development methodologies, but from the perspective of process. There are existing reviews of methodologies from the perspective of modeling languages available in the literature, but there is little up-to-date synthesis material available that focuses on process. The review is carried out for the following motivations:

—to help users of methodologies—for example, project managers, development teams, and auditors—understand the processes, phase interleavings, and expected interactions that a methodology implicitly or explicitly requires;

—to provide information that may be useful for users of methodologies who wish to tailor or select processes for their particular project or context;

—to help users value processes with respect to their projects and contexts.

It is inevitable that, in such a review, issues related to modeling languages will be touched upon, but we attempt to abstract away from these details in order to focus on process; standardization efforts for the UML are of course essential and helpful here.

We summarize and review the methodologies using a process-centered template, highlighting the activities prescribed in each methodology while keeping the description and discussion of the languages used (mainly diagrams and tables) as secondary to the activities. The description produced using this template offers little critique on the methodologies—indeed that is not our goal—yet abstracts and structures them in a way that enables elaborate analysis of individual methodologies, as well as comparative analysis of collections of them. 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:

   —details of the activities performed in the subprocess and the order in which they are performed;

   —a concise description of 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 methodology and obscure the process. In a complete description of the methodology, this subsection would provide a thorough overview of the languages used, but we shall deemphasise this subsection in this review.

The structure of this article is as follows. We commence with basic definitions and a brief history of object-oriented software development methodologies, charting the appearance of hybrid methodologies, the move to seminal methodologies, and the development of integrated (third-generation/heavyweight) methodologies and their agile (lightweight) counterparts. We then provide a comprehensive review of the processes of a selection of seminal, integrated, and agile methodologies, as well as process patterns and process metamodels, before concluding with a discussion on engineering the processes of next-generation methodologies. The methodologies, process patterns and process metamodels reviewed in this article are:

(1) *Methodologies*:
  (1.1) *Seminal*: Shlaer-Mellor, Coad-Yourdon, RDD, Booch, OMT, OSA, OOSE, BON, Hodge-Mock, Syntropy, Fusion;
  (1.2) *Integrated*: OPM, Catalysis, OPEN, RUP/USDP, EUP, FOOM;
  (1.3) *Agile*: DSDM, Scrum, XP, ASD, dX, Crystal, FDD;
(2) *Process Patterns*: Ambler;
(3) *Process Metamodels*: OPF (as part of the OPEN methodology), SPEM

In a review article like this, it is inevitable that some methodologies will be missed or deprecated; we have tried to keep this to a minimum, and any omissions or deprecations should not be viewed as a value judgment. The volume of methodologies available and the commonalities among some of them make it difficult to provide complete coverage.

2. 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 2003b], 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:
  —provides guidance as to the order of the activities;
  —specifies what artifacts should be developed using the *modeling language*;
  —directs the tasks of individual developers and the team as a whole; and
  —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, behavioral component of the methodology, governing the technical development 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. It has also been suggested that every methodology also has an underlying *philosophy*: the set of assumptions and beliefs of the authors of the methodology that are reflected in the methodology, and which support and rationalize the use of the methodology in its intended context [Avison and Fitzgerald 2003a].

It should be noted here that there is a strong argument against using the term *methodology* in this context, the proponents of which point out that methodology is the *study* of methods, not a type of method as used here, and therefore suggest the use of the general term *method* instead [Brinkkemper 1996; Graham 2001]. Although this point is lexically correct, we have chosen to use the term methodology because of its widespread adoption in the software engineering community, and also because of the generality of the term method; whereas method is used in software engineering for referring to various concepts and constructs at different levels of abstraction and diverse degrees of granularity, methodology is unequivocally used in the sense cited here.

3. OBJECT-ORIENTED METHODOLOGIES: A BRIEF HISTORY

Over the years, software development methodologies have evolved from shallow and informal "in-house" methodologies of the 1960s to the object-oriented methodologies of the 90s and the new millennium. Object-oriented software development methodologies (OOSDM) are specifically aimed at viewing, modeling and implementing the system as a collection of interacting objects, using the 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 introduced. A brief description of the categories of OOSDMs and their trends of evolution will help further clarify the domain.

3.1. 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 methodologies were hence categorized as *transformative* [Monarchi and Puhr 1992]. The methodologies 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 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. The so-called "methodology war," during which dozens of methodologies competing for a share in the software development industry, occurred during this period.

The sheer number of methodologies introduced became so prohibitive that choosing the right methodology for a software project was a major endeavour. 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 [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 1 shows the methodologies developed during the period from 1986 to 1996 [Webster 1996]. It also shows the evolution timeline and genealogical relationships among the methodologies, emphasizing the influences and the contributions.

3.2. The Unified Modeling Language (UML)

UML 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 in essence identical. 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 it should be process-independent 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, 2004]. Yet the opposite is not true: processes do tend to become dependent on the modeling language they adopt. This 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, Ambler et al. 2005, Shoval 2007].

The original developers of UML were Rumbaugh, Jacobson, and Booch [Booch et al. 1999, 2005]; 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 and is constantly reviewed and revised under OMG's supervision.

3.3. Integrated Methodologies: Third Generation

Methodologies in this category are results of integrating seminal methodologies and are characterized by their process-centered attitude towards software development, typically targeting a vast variety of applications. Integrations have resulted in huge 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

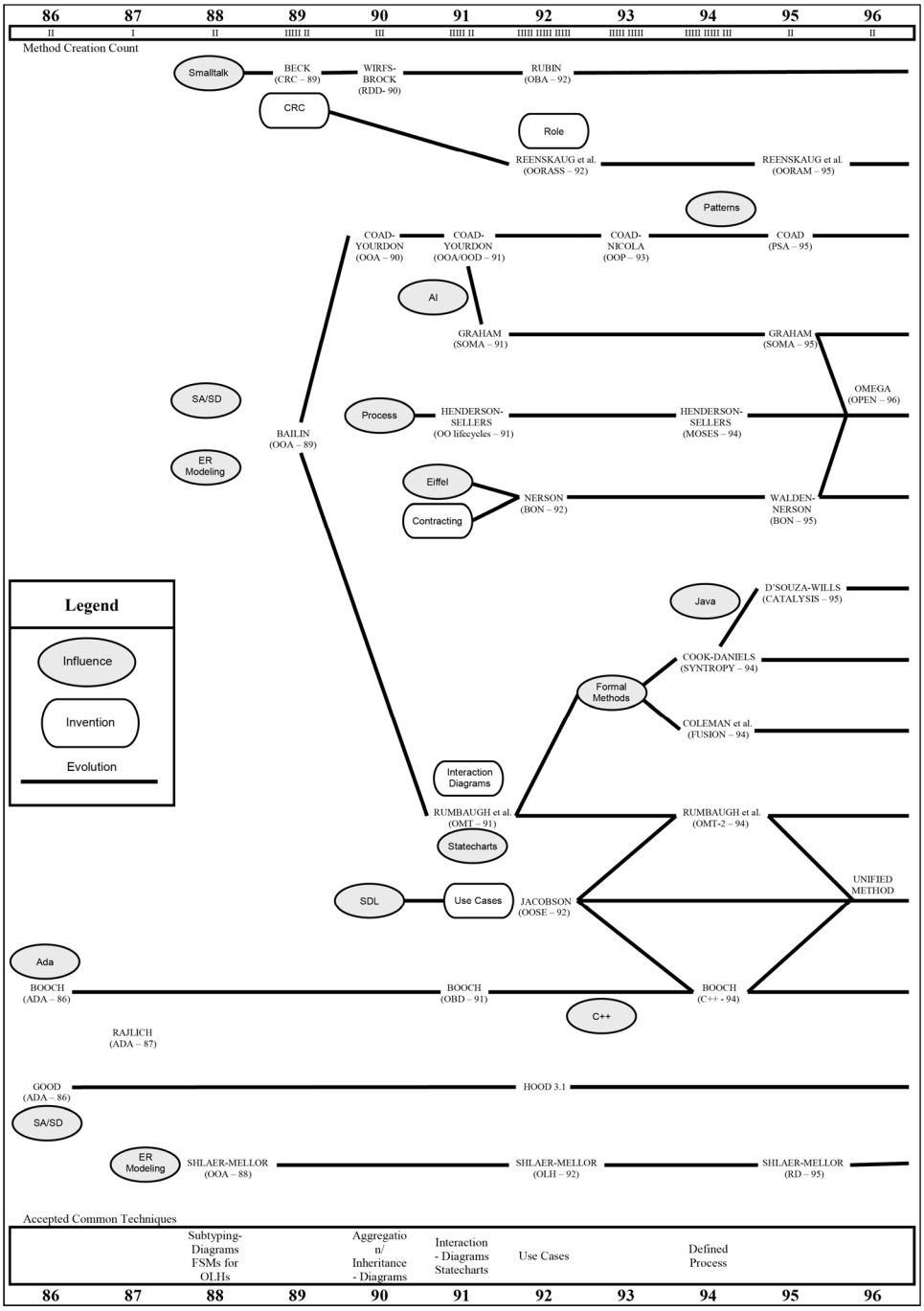

Fig. 1. The evolution timeline of object-oriented methodologies up to 1996—adapted from Webster [1996].

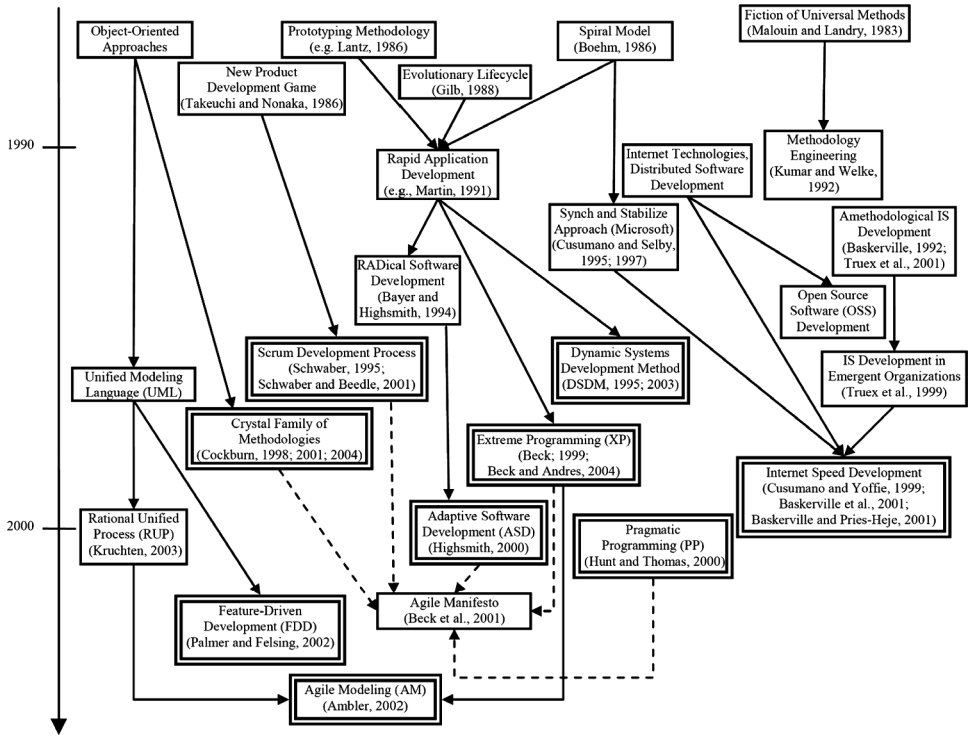

Fig. 2. The evolution map of agile methodologies—adapted from Abrahamsson et al. [2003].

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). 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 approaches.

3.4. 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-and-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 2 shows an evolution map for a number of agile methodologies, emphasizing the ways previous methodologies and practices have influenced them.

Agile software development has also been influenced by the Patterns movement of the 1990s [Coplien 2006]. The overall attitude of agile methodologies towards software development has been summarized in the "Agile Manifesto," agreed upon by major agile methodologies (Figure 3).

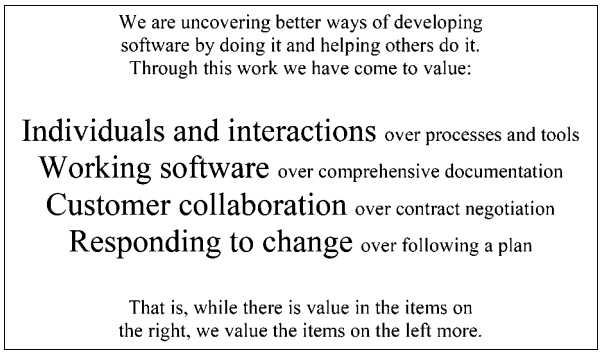

Fig. 3. The Agile Manifesto [Beck et al. 2001].

3.5. 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. 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]. 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].

4. SEMINAL METHODOLOGIES

Due to the large number of these methodologies, we are examining only the most renowned and influential: methodologies that, according to the evolution timeline of Figure 1, either 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 use; notational conventions, however, are beyond the scope of this article; the interested reader is referred to Wieringa [1998] and Graham [2001] for information on notations, as well as the literature cited for each methodology in the following subsections.

4.1. 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], 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 behavior, 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) partition the system into domains according to the four domain types defined in the methodology; the partitions practically divide the structure, functionality,

and behavior of the software system into four tiers: problem domain, application-independent services, physical architecture, and physical implementation;
(2) analyze the application (problem) domain;
(3) confirm the analysis through static and dynamic verification;
(4) extract the requirements for the application-independent service domains that support the application domain;
(5) analyze the service domains;
(6) specify the components of the architectural domain (physical configuration of the software);
(7) build the architectural components;
(8) translate (implement) the analysis models of relevant domains into the architectural components.

These steps are briefly described in the following sections.

4.1.1. 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; for example, 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*).

4.1.2. Analyzing the application domain (Shlaer-Mellor). The next step involves applying object-oriented analysis (OOA) 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, which defines the objects of the domain, and the relationships between them.
(2) State Models are built, which show the life cycle (behavior) of each object.
(3) Action Specification Models are built, which depict the processing taking place in the state models. Usually there is one action specification for each state in each object's life cycle. 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 among subsystems within a domain. These models are:

(1) Subsystem Relationship Model: showing the structural relationships among objects in different subsystems;
(2) Subsystem Communication Model: showing event communications among objects in different subsystems;
(3) Subsystem Access Model: showing data accesses among objects in different subsystems.

The methodology also prescribes the production of a number of *derived models* for each of the subsystems as listed here.

(1) Object Communication Model: showing the event communications among objects;
(2) Event List: showing events being sent within or among state models;
(3) Object Access Model: showing the data accesses among 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.

4.1.3. 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 behavior 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 behavior 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 results expected according to the desired behavior.

4.1.4. 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.

4.1.5. 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 behavior 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).

4.1.6. 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 in a layered structure to model the design of an object oriented program, library, or environment:

—Inheritance Diagrams, which show the inheritance relationships among classes;

—Dependency Diagrams, showing usage (client/server) and friend relationships among 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.

4.1.7. 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.

4.1.8. 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 in 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.

4.2. 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

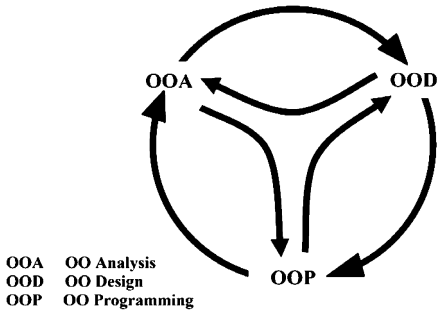

Fig. 4. The Coad-Yourdon model for software development—adapted from Coad and Nicola [1993].

is shown in Figure 4 (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.

4.2.1. Analysis (Coad-Yourdon). The analysis (OOA) part of the methodology consists of five principal activities:

(1) finding *classes* (abstract classes) and *class-and-objects* (concrete classes);
(2) identifying *structures* (generalization-specialization and whole-part relationships among classes);
(3) identifying *subjects* (partitions/subsystems);
(4) defining *attributes*, and *instance-connections* (association relationships among 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-and-Object Diagram that is the pivotal model of the system. In accordance to these major activities, the resulting class-and-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-and-object layer*: showing the abstract and concrete classes of the system;
(3) *structure layer*: which shows the generalization-specification and whole-part relationships among the classes;
(4) *attribute layer*: showing the attributes of the classes and the association relationships among classes;
(5) *service layer*: which shows the operations of the classes and the potential message-passing among the objects (even the sequence of the messages can be modeled).

The class-and-object diagram is supplemented by various behavioral diagrams produced during the identification of the operations and the message-connections (activity 5 of the analysis phase). Typically, the dynamic behavior 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.

4.2.2. 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, for example, 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 intertask 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-and-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 conveyed by class-and object diagrams.

4.3. 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 requirements specification of the system has already been provided. This means that certain requirements typical analysis activities, including requirement 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; and
(2) the actions an object can perform.

The process is divided into two phases: the *Exploratory Phase* and the *Analysis Phase* (Figure 5). A brief overview of each phase is given in the next sections.

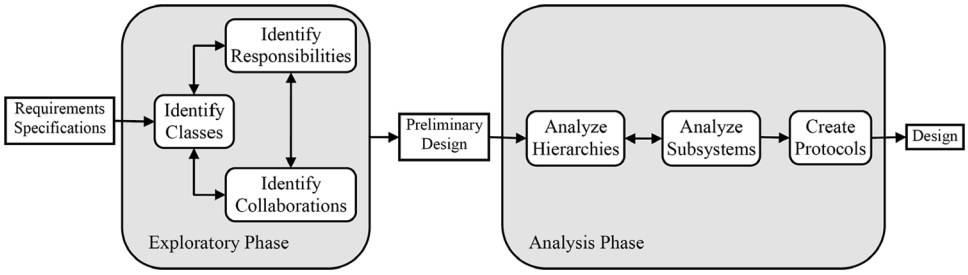

Fig. 5. The RDD process—adapted from Wirfs-Brock et al. [1990].

4.3.1. Exploratory Phase (RDD). The major tasks in this phase are to:

(1) discover the classes required to model the application;

(2) determine what behavior the system is responsible for and assign these responsibilities to specific classes; and

(3) determine what collaborations must occur among 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 Class-Responsibility-Collaborator (CRC) 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.

4.3.2. 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, the list of requests that a client can make of the class; a class may support numerous contracts, showing different behavior to different clients;

(2) identifying possible subsystems of objects and modeling the collaborations among 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 among 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 the 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.

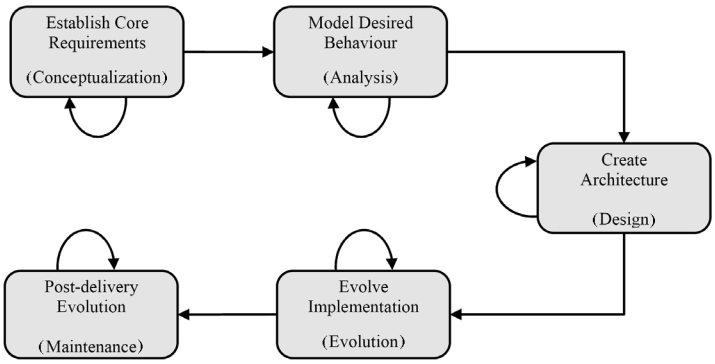

Fig. 6. The Macro Process of the Booch methodology—adapted from Booch [1994].

4.4. 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 6 shows the macro process as prescribed by Booch (the self-iterations represent the micro process).

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 7 shows the various tasks involved in the micro process.

These two processes are further described in the next sections.

4.4.1. 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 behavior (analysis);
(3) create an architecture for the implementation (design);
(4) evolve the implementation through successive refinements (evolution);
(5) manage post-delivery evolution (maintenance).

4.4.2. Micro Process (Booch). The micro process tends to cycle through the following activities [Booch 1994]:

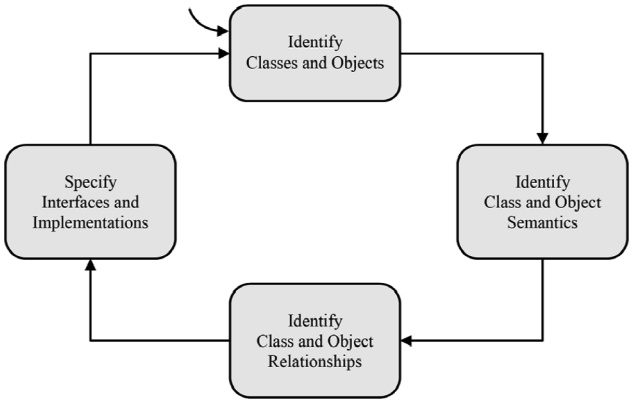

Fig. 7. The Micro Process of the Booch Methodology—adapted from Booch [1994].

(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 through 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 4.3.1) 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 in order to establish the meanings of the classes and objects identified in the previous step, with the emphasis chiefly on the behavior of the system and its constituents, rather than the structure. This is done by establishing the behavior 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 behavior. 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 among the objects are shown; the arrows and sequence numbers, which determine the behavioral 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 behavioral detail during the *second* activity in later iterations.

(3) Identify the relationships among classes and objects: once behavior 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 that 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 latter 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.

4.5. OMT (1991)

Object Modeling Technique (OMT) was introduced by 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) 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) indicates the dynamics of the objects and their changes in state. It captures the essential behavior of the system by exploring the behavior 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) is a hierarchical set of data flow diagrams (DFD) 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 that 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.

A use-case-driven version of OMT, coined OMT-2, was proposed by Rumbaugh [1994]; in OMT-2, use case diagrams and object interaction diagrams replace DFDs as constituents of the functional model.

The OMT process consists of five phases, as shown in Figure 8. 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.

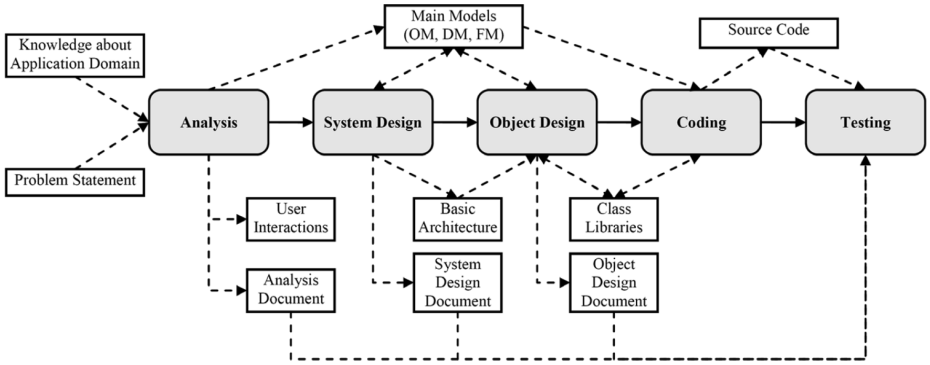

Fig. 8. The OMT process and its deliverables—adapted from Derr [1995].

4.5.1. 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) build the object model, including a class diagram, depicting the classes of the system and their relationships, and a data dictionary;
(2) develop the dynamic model, including state transition diagrams and global event-trace diagrams; OMT identifies the following steps in constructing the dynamic model:
  (2.1) identify patterns of system usage and prepare scenarios of typical interaction sequences;
  (2.2) identify events among objects and prepare an event-trace diagram for each scenario;
  (2.3) prepare an event-trace diagram for the system, showing events flowing at the boundary of the system;
  (2.4) develop state transition diagrams for classes with important dynamic behavior;
  (2.5) check for consistency and completeness of events shared among the state transition diagrams;
(3) construct the functional model, including data flow diagrams and constraints;
(4) verify, revise, and refine the three models.

4.5.2. 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.

4.5.3. 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.

4.6. OSA (1992)

Object-oriented systems analysis (OSA) 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 prespecified 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 be considered a methodology at all. 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 behavior, 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-behavior 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.

4.6.1. 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 behavior. 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.

4.6.2. Object-Behavior Model—OBM (OSA). The OBM describes the behavior of objects in a system. It consists of a collection of *state nets*, each of which defines the behavior 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.

4.6.3. Object-Interaction Model—OIM (OSA). An OIM captures information about interactions among objects. OIM components include objects, interactions, and various types of constraints. High-level interactions are those described by more detailed OIM diagrams.

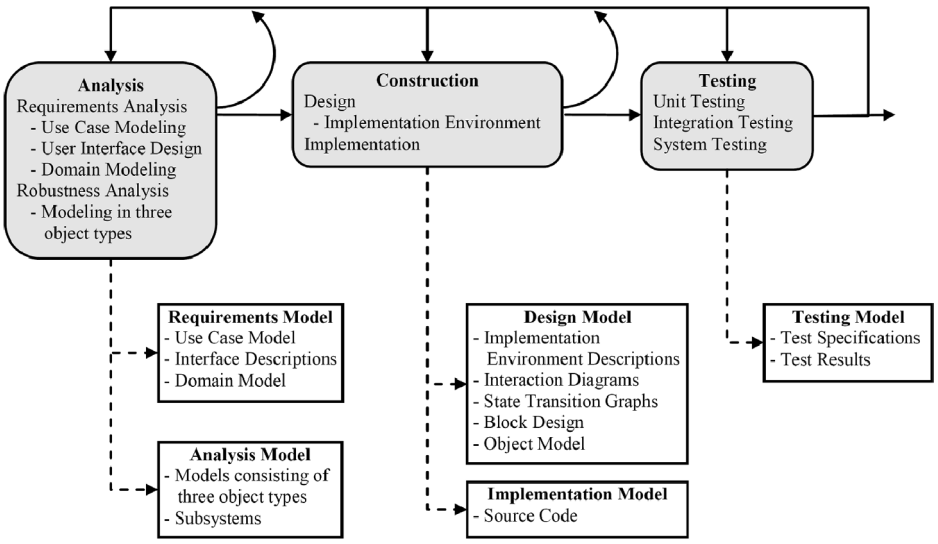

Fig. 9. The OOSE process and the models produced—adapted from Jacobson et al. [1992].

4.7. OOSE (1992)

Object-oriented software engineering [OOSE] [Jacobson et al. 1992] 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*: focuses on understanding the system and creating a conceptual model of it. This phase consists of two non-sequential, iterative subphases:
   (1.1) *requirements analysis*, aims at eliciting and modeling the requirements of the system. A requirements model is produced as a result of this activity.
   (1.2) *robustness analysis,* aims 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*: focuses on creating a blueprint of the software and producing the code. This phase consists of two subphases:
   (2.1) *design*, aims at modeling the run-time structure of the system, and also the interobject as well as intraobject behavior necessary to realize the requirements. A design model is produced as the result of this activity.
   (2.2) *implementation*, aims at building the software. An implementation model (including the code) is produced as the result of this activity.

(3) *testing* focuses on verifying and validating the implemented system. A test model is produced during this phase.

Figure 9 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.

4.7.1. 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.

4.7.1.1. 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* delimits the system and describes the functional requirements from the user's perspective. The use case model specifies the complete functional behavior 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 among the use cases themselves: a use case may *extend* another use case's behavior, or *use* another use case in order to perform its own functionality.

—A *domain object model* consists of objects representing entities derived from the problem domain, and their inheritance, aggregation, and association relationships.

—*Interface descriptions* 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.

4.7.1.2. 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 and typically outlive 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 behavior 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 among 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.

4.7.2. 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 behavior of the system, and producing the final code. The two subphases closely correspond to the generic lifecycle activities of the same names.

4.7.2.1. 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 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 behavior 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 behavior of each block.

4.7.2.2. 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.

4.7.3. 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.

4.8. BON (1992, 1995)

The BON Methodology was first introduced in a paper by 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

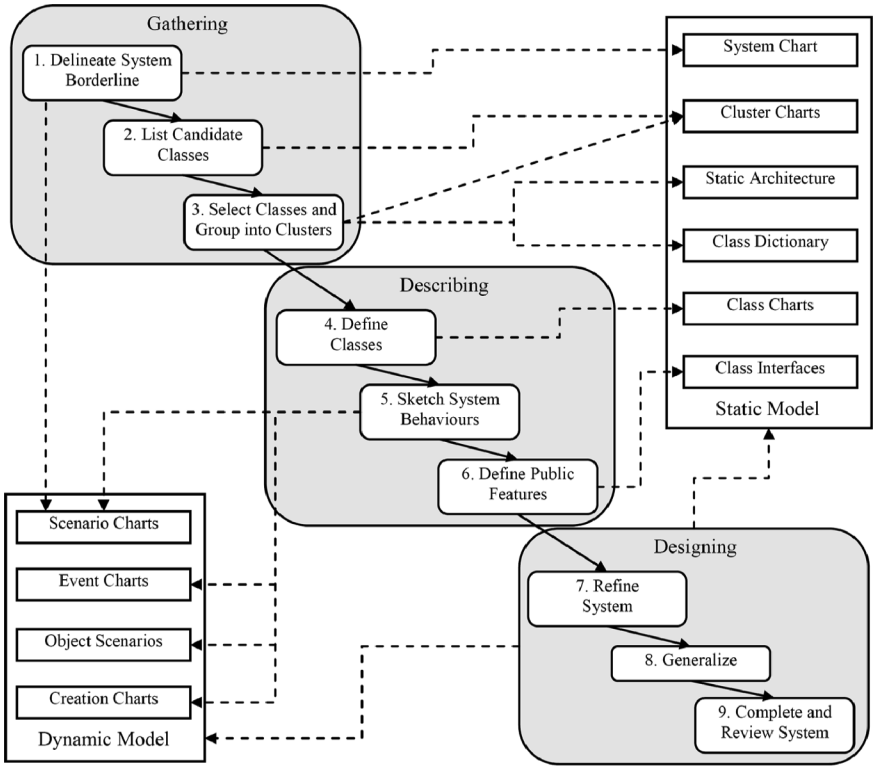

Fig. 10. The BON process: the tasks and their deliverables—adapted from Walden and Nerson [1995].

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* (Figure 10). 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.

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 depicted in Figure 10. 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 among 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, the object types responsible for the creation of other objects, and a number of system execution scenarios representing typical types of system usage.

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 model.

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.

4.8.1. 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 subclusters. 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 behavior 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 which belong 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 internal or external stimuli which shows the resulting events in the order they occur. Some interesting system scenarios are usually collected to illustrate important aspects of the overall system behavior. A description of the scenarios, depicting the actions fulfilled in each, is then tabulated as scenario charts.

4.8.2. 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, cluster charts will be refined and completed during the BON process and will ultimately contain descriptions of the classes and subclusters 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.

4.8.3. Selecting Classes and Grouping into Clusters (BON). In this task, beginning with the list of candidates produced in task 1. 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 among 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.

4.8.4. 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 behavior (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.

4.8.5. Sketching System Behavior (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 among 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.

4.8.6. 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 among classes, which are also modeled in the charts. The static architecture is updated to reflect the refinements done in this task.

4.8.7. 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. 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.

4.8.8. Generalizing (BON). This task concerns improving the inheritance hierarchy of the classes by factoring common state and behavior into deferred (abstract) superclasses. The relevant diagrams and tables—including the static architecture, class interfaces, and the class dictionary—are updated accordingly.

4.8.9. 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.

4.9. Hodge-Mock (1992)

The methodology introduced by Hodge and Mock [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. 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*: Focuses on refining the requirements and identifying the scope, structure and behavior of the system. This phase in turn consists of four subphases:
  (1.1) *requirements analysis*: Focuses on eliciting the requirements of the system.
  (1.2) *information analysis*: Focuses on determining the classes in the problem domain, their interrelationships, and the collaborations among their instances.
  (1.3) *event analysis*: Focuses on identifying the behavior 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*: Focuses on providing a more detailed view of the collaborations among objects.

(2) *system design*: Focuses on adding design classes to the class structure of the system and refining the external behavior of each of the classes.

(3) *software design*: Focuses on adding implementation-specific classes and details to the class structure of the system, and specifying the internal structure and behavior of each class.

(4) *implementation*: Focuses on coding and unit testing.

(5) *testing* focuses on system-level verification and validation.

Figure 11 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,

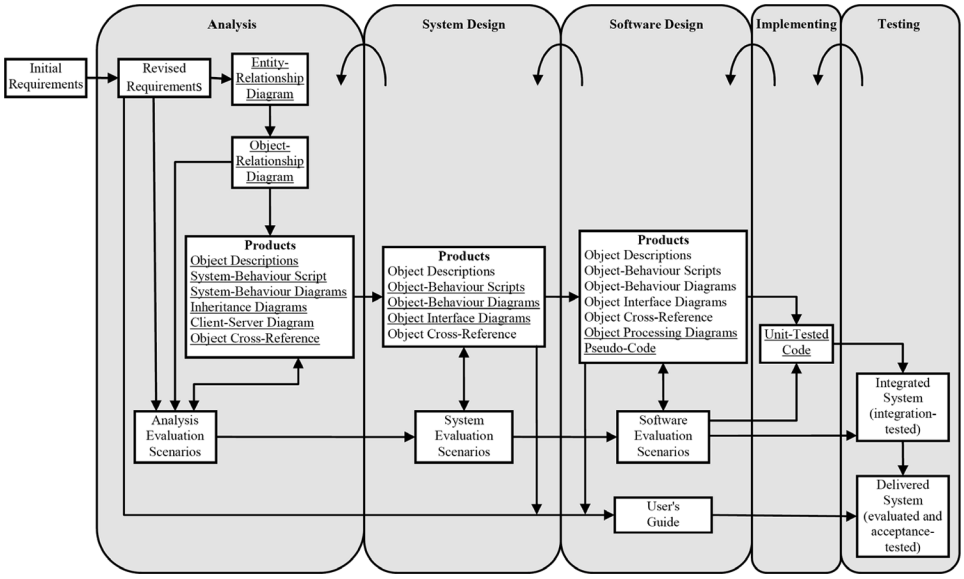

Fig. 11. The Hodge-Mock process: the phases and their deliverables—adapted from Hodge and Mock [1992].

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 11).

4.9.1. 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 nonfunctional 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 together classes of close functional or structural relationships.

(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 behavior (services) of individual classes, generalization-specialization (isa) relationships existing among 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 Behavior Script (SBS).

(3.2) The external behavior of the system is captured in a system behavior diagram (SBD), which is a state transition diagram showing the states the system can be in and the state transitions triggered by external stimuli (events).

(3.3) Based on system behavior determined in previous tasks (stimuli and activities), data elements and objects required to provide the behavior 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 behavior of the system. Results are used for verifying and 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 within 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.

4.9.2. 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 among instances of the individual class interact 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 behavior of each class, an object behavior 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 behavior. Analogous to the SBD and following the same notation, this class-level state transition diagram is called the object behavior diagram (OBD).

(4) Class definitions in tables and diagrams are refined in order to include the detailed signatures 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.

4.9.3. 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 nonfunctional requirements are the major issues necessitating additions and refinements to the models.

(2) The internal structure and behavior 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 pseudocode 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 to 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.

4.10. Syntropy (1994)

Syntropy, introduced by 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 to 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 behavior 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 behavioral 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 behavioral view by describing the possible states for each object type 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): is solely used in the implementation model for describing the flow of messages among objects in response to stimuli.

Syntropy supports the notion of *domain*: a subsystem 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.

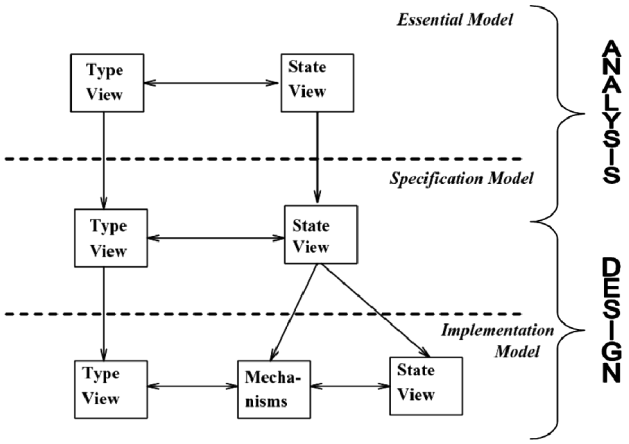

Fig. 12. Models, views and their interdependencies, showing the implicit syntropy process—adapted from Cook and Daniels [1994].

Figure 12 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 behavior); likewise, the generic concept of system design is seen in the remaining part of the specification model (design of state-charts and interactions among 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.

4.10.1. Essential Model (Syntropy). In the essential model, the problem domain is modeled 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.

4.10.2. Specification Model (Syntropy). The specification model describes the possible states for the software, 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: 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 behavior of the software objects in response to stimuli, either external or issued by other objects. External stimuli are simultaneously observable by all the objects in the model. 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 (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.

4.10.3. 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 interrelationships (links), and the sequence of the messages passed among 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.

4.11. 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 in which 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:
   —classes and objects of interest found in the application domain, and the relationships that exist among these classes and objects,
   —the operations that are to be performed by the system; and
   —the proper ordering of these operations.

(2) design is the focus 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:
   —realization of system operations in terms of cooperating objects;
   —how these objects are linked together;
   —how the classes to which the objects belong, are specialized and refined (the inheritance structure of the classes); and
   —detailed particulars of each class's attributes and methods.

(3) implementation is 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.

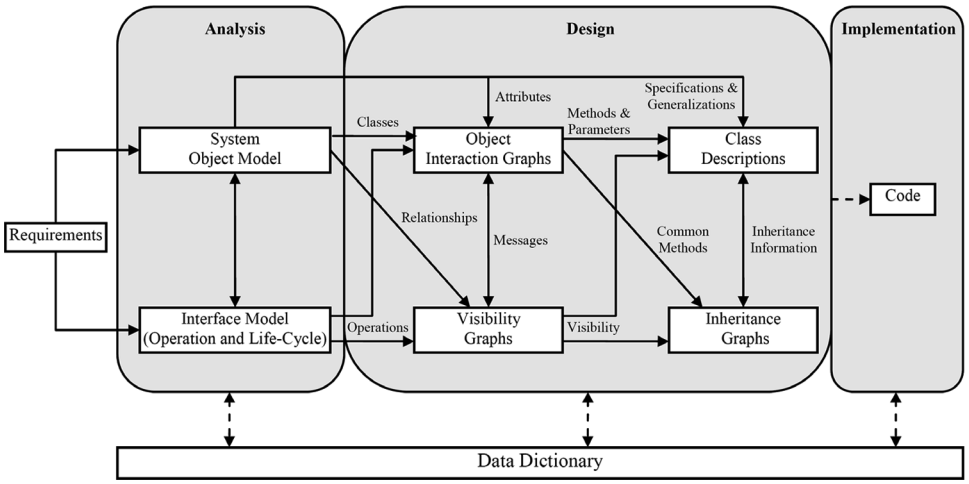

Fig. 13. The Fusion process and its deliverables—adapted from Lano et al. [2000].

Figure 13 shows the Fusion process, the models produced, and the interdependencies among the models, describing what they 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 subphases. A brief description of each phase, the subphases, and the models produced, is given in the next sections.

4.11.1. 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.

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.

4.11.2. 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 among classes.

The design phase consists of the following steps:

(1) Develop the object interaction graphs, which are used to develop the 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, which describe the server objects that a client object needs to reference and specify the kind of references that are 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, which 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 behavior in order to increase reusability and maintainability. The result is summarized in inheritance graphs.

4.11.3. 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.

5. INTEGRATED METHODOLOGIES

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 megamethodologies. Although many of these methodologies have adopted UML as their modeling language, they share little else, particularly as pertaining to process and modeling approach.

5.1. OPM (1995, 2002)

Object-Process Methodology (OPM) was introduced by Dori [1995], primarily as a novel approach to analysis modeling that advocated combining the classic process-oriented modeling approach with object-oriented modeling techniques. 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 behavior 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 behavioral 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 object-process language (OPL) 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 [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 behavioral 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 OPM process.

The OPM process consists of three high-level subprocesses:

(1) initiating focuses on preliminary analysis of the system, determining the scope of the system, the required resources, and the high-level requirements.
(2) developing focuses on detailed analysis, design, and implementation of the system.
(3) deploying focuses 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 these subprocesses.

5.1.1. Initiating (OPM). The following activities are performed during this subprocess:

(1) identifying the needs and/or opportunities justifying the development of the system;
(2) conceiving the system 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.

5.1.2. Developing (OPM). The following activities are performed during this subprocess:

(1) analyzing, which is 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, which is 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 semiautomatic 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.

5.1.3. Deploying (OPM). The following activities are performed during this subprocess:

(1) assimilating, which is 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, by 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 nonfunctional 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, which is concerned with declaring the current system as dead, applying the usual postmortem procedures, and prompting the generation of a new system.

5.2. Catalysis (1995, 1998)

Catalysis was introduced by D'Souza and Wills [1995], originally as a component-based formalization of OMT deeply influenced by Fusion, Objectory, Booch and Syntropy. A UML-based, refined version of the methodology appeared in 1998 [D'Souza and Wills 1998]. Catalysis has greatly influenced and inspired object-oriented component-based development processes, one of the most notable results of which is the UML Components methodology [Cheesman and Daniels 2001].

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 at hand. However, it does propose a specific process for developing business systems, as shown in Figure 14. 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. The main point of interest is the seamless fractal modeling approach, which is a major contribution of Catalysis. The last section contains a description of the process patterns proposed by the methodology.

5.2.1. Business Systems Development Process (Catalysis). This process gradually moves 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]. It consists of the following activities:

(1) identify and model the requirements, which focuses on exploration and modeling of the problem domain and the requirements of the system;
(2) develop the system specification, which focuses 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 specifications describing the screens, dialog flows across windows, information presented and required, and reports;
(3) develop the architectural design, which focuses on designing the internal component (logical) architecture of the system, as well as the technical (physical) architecture

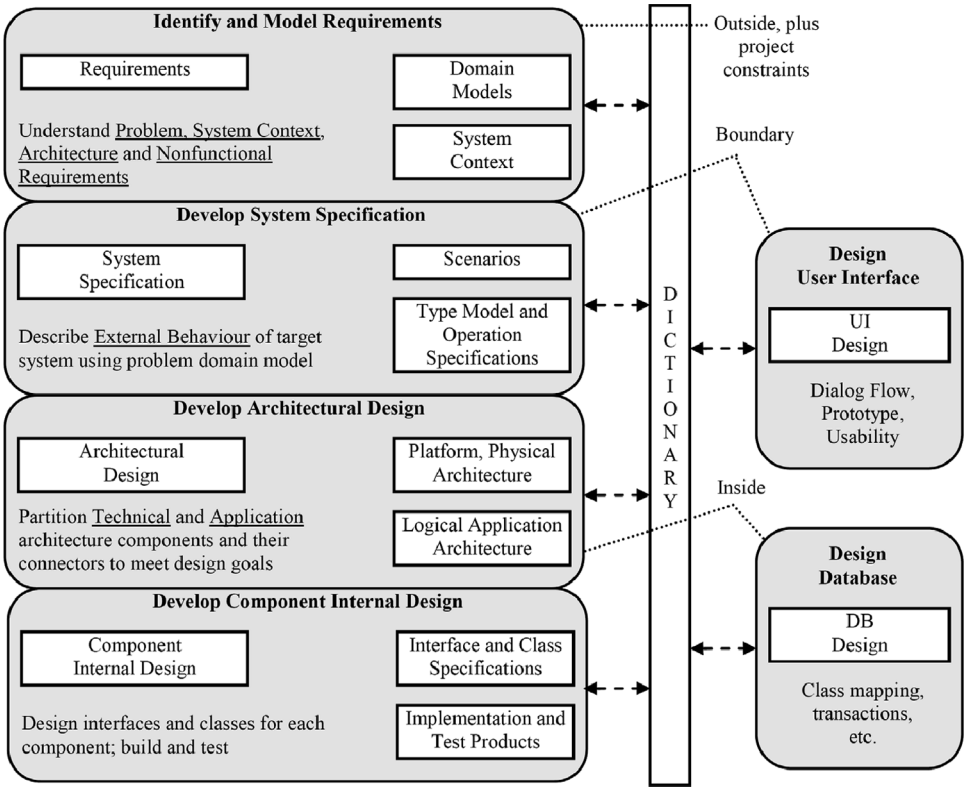

Fig. 14. The Catalysis process for developing typical business systems—adapted from D'Souza and Wills [1998].

defining the domain-independent parts of the system, such as the hardware and software platforms; 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, which focuses on designing the internal details of the components, which are then implemented and tested.

The following sections include brief descriptions of these activities.

5.2.1.1. Identify and Model the Requirements (Catalysis). The following tasks are performed during this activity:

(1) Explore the problem domain and construct the business model, which 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 nonfunctional 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 among 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.

5.2.1.2. 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 behavior of the system is usually captured in statecharts.

5.2.1.3. 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 interrelationships. 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 interrelationships. Interactions are shown in collaboration diagrams.

5.2.1.4. 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 preexisting 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 interrelationships. Interactions are shown by sequence and collaboration diagrams.

5.2.2. Overall Approach (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 behavior are carefully modeled. The focus is then shifted onto the system itself, modeling it as a collection of components, again with their own interrelationships 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 interrelationships and interactions. Catalysis provides specifics as to how the diagram types used at each level

are linked, thus enriching the modeling language with semantics that facilitate the production of highly cohesive models.

5.2.3. Process Patterns (Catalysis). This sort of gradual refinement is an essential practice in Catalysis. So is the recursive (fractal) modeling approach: applying the same view (constituents, their interrelationships, 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].

5.3. OPEN (1996)

Object-oriented process, environment, and notation (OPEN) 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 [Walden and Nerson 1995] 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 [Henderson-Sellers and Unhelkar 2000]. 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) [Firesmith and Henderson-Sellers 2001]. OPF is a process metamodel defining five classes of components and guidelines for constructing customized OPEN processes (Figure 15). 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 at hand [Henderson-Sellers et al. 1998]. The OPF component classes and the instantiation method for constructing OPEN processes are discussed in the following sections.

5.3.1. OPF Component Classes (OPEN). As depicted in Figure 15, 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;

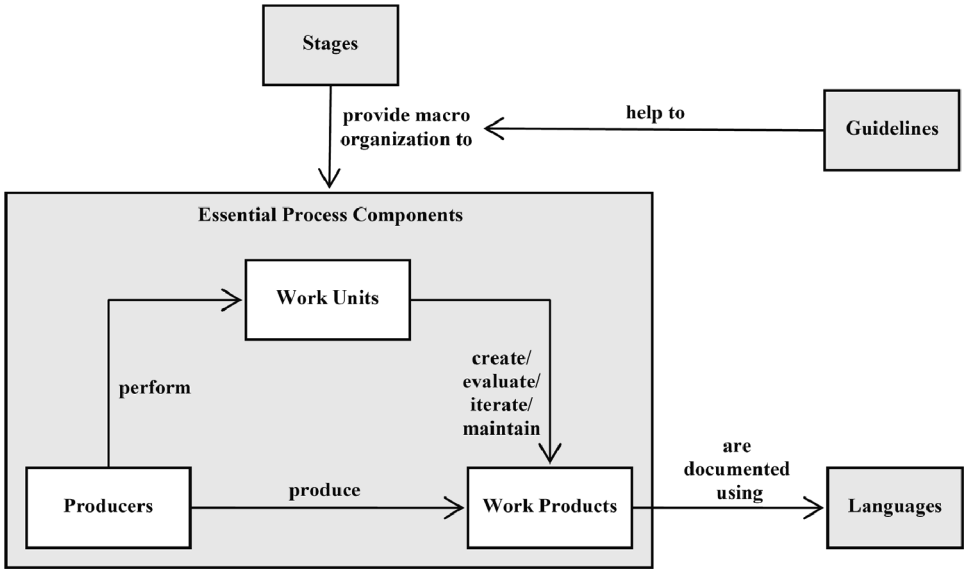

Fig. 15. OPF components (OPEN)—adapted from OPEN Consortium [2000].

(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:
  —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);
  —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;
  —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:
  —milestone (instantaneous stage): a point in time marking the occurrence of an event.
  —stage with duration: the high-level periods during which work units are performed. There are seven significant types:
    (5.1) project: covering a single individual project;
    (5.2) 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:
      —project development lifecycle: the duration over which the project is conceived and products are constructed;

—project lifecycle: covering the project development lifecycle and the maintenance stage;
—delivery lifecycle: focusing on the repetitive delivery of product versions;
—enterprise lifecycle: in which business modeling and business reengineering occur;
—program lifecycle: larger in scale than the project lifecycle, this is a cycle related to a program 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;

(5.3) 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 lifecycles; seven phases have been defined in OPF:
—inception: during which the development is started and appropriate preparations are made;
—construction: during which the work products are developed and prepared for release;
—usage: during which the work products are released to the user organization and put into service;
—retirement: when the software is withdrawn from service;
—strategy: in which cross-project considerations at the business level are analyzed;
—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);
—business reengineering: in which the processes in the business are analyzed and reconsidered;

(5.4) 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;

(5.5) build: during which tasks are undertaken; builds are the only kinds of stages that occur within the inception phase; in other phases, they are generally complemented by releases, deployments, and milestones;

(5.6) release: in which the results of a build are delivered to the user;

(5.7) deployment: when the user receives the product and puts it into service.

5.3.2. Process Instantiation (OPEN). 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.

5.4. RUP/USDP (1998, 1999, 2000, 2003)

Rational unified process (RUP) 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 nonproprietary, 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; Kroll and Kruchten 2003]:

(1) *inception*: focused on defining the objectives of the project, especially the business case;
(2) *elaboration*: focused on capturing the crucial requirements, developing and validating the architecture of the software system, and planning the remaining phases of the project;
(3) *construction*: focused on implementing the system in an iterative and incremental fashion based on the architecture developed in the previous phase;
(4) *transition*: focused 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 realize the 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.

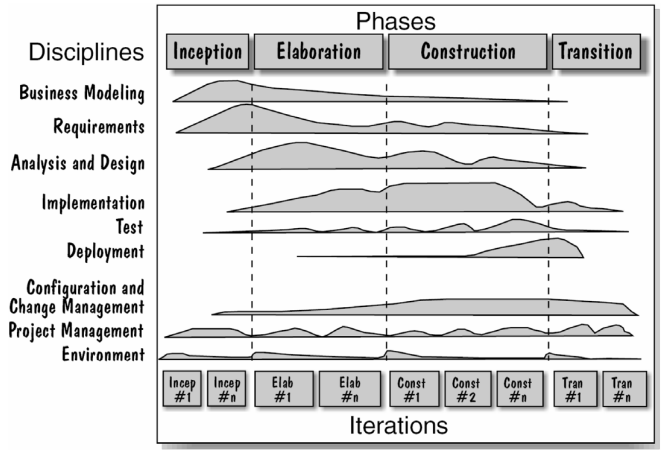

Fig. 16. A typical RUP lifecycle model—adapted from Kroll and Kruchten [2003].

(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.

The disciplines do not have equal emphases 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 requirements management take a lot of emphasis during earlier phases, whereas during later phases, most of the effort is put into deployment and testing. Figure 16 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 artifacts (work products), activities (units of work on the artifacts), and roles (responsibilities taken on by development team members).

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

5.4.1. 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 artifacts are usually produced during this phase:

—a vision document: a general description of the project's core 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.

5.4.2. 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 artifacts 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 nonfunctional 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 a 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).

5.4.3. 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 artifacts are usually produced during this phase:

—the software product;
—the user manuals;
—a description of the current release.

5.4.4. 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 artifacts 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.

5.5. EUP (2000, 2005)

Enterprise unified process (EUP) was introduced by Ambler and Constantine in 2000 as an extended variant of RUP [Ambler and Constantine 2000a]. 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:

—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 17. 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 on modeling is also somewhat different from RUP. Whereas RUP advocates adherence to UML, EUP also makes use of some older modeling notations. 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.

5.5.1. New Phases (EUP). The two new phases that EUP has added to RUP are:

—*production*: added as the fifth phase, its focus 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; it 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];

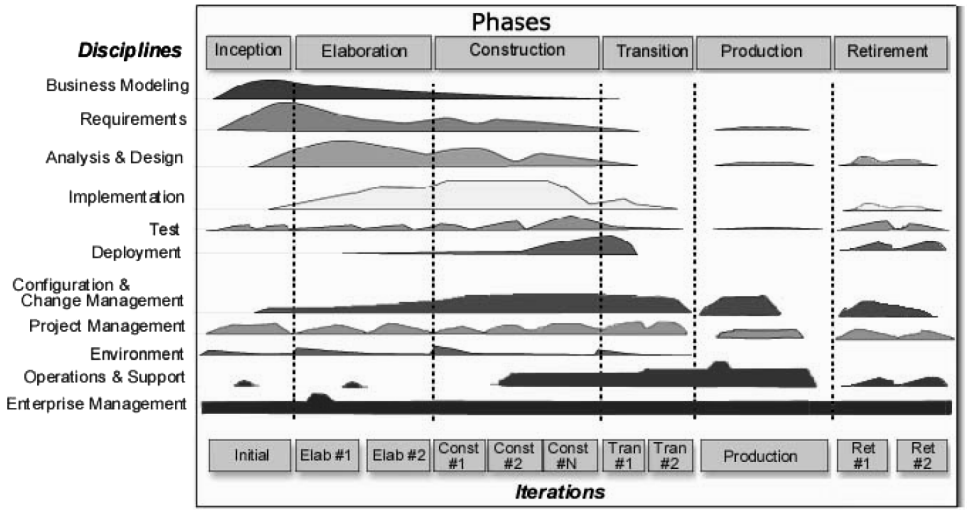

Fig. 17. A typical EUP lifecycle model [Ambler and Constantine 2000a].

—*retirement*: added in 2002 as the sixth phase, its focus is on the careful removal of a system from production, either because it is no longer needed or is being replaced; this typically involves [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;
—archiving of data previously maintained by the system, but which 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.

5.5.2. 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 artifacts 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 artifacts, such as the organization-wide models (requirements and architecture), software process, standards, guidelines,

and reusable artifacts [Ambler and Constantine 2000a, 2000b, 2000c]; 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.

5.5.3. 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, 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].

5.6. FOOM (2001, 2007)

Introduced by Shoval and Kabeli [2001], functional and object-oriented methodology (FOOM) is an object-oriented variant of Shoval's ADISSA methodology of 1988 [Shoval 1988]. Architectural design of information systems based on structured analysis (ADISSA) 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. A refined version of FOOM appeared in 2007 [Shoval 2007].

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 behavioral 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 intrasystem 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 behavior*: 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 behavioral modeling during systems analysis. Targeting data-intensive systems has also resulted in a slack attitude towards behavioral design of the system; many of the activities prescribed in the design-of-system-behavior 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 behavioral specifications), using any common object-oriented programming language.

5.6.1. 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, it is preferable to start with data modeling [Kabeli and Shoval 2003].

5.6.2. 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 among 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 nonleaf process connected to a user entity, a corresponding submenu is defined and initialized as empty, and for every process (leaf or nonleaf) 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 Behavior—this stage of the design phase produces the main behavioral 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.

6. AGILE METHODOLOGIES

Agile methodologies have many commonalities, listed by Beck et al. [2001] as a set of agile principles:

—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 behavior accordingly.

Although many agile methodologies claim that their methods are not process-centered, close examination usually shows that some sort of iterative-incremental process (sometimes quite elaborate) can be found in all of them. Whereas at the start of the agile movement words like "process" (even "methodology") were considered dirty, agile methodologists are now showing increasing interest in advertising their agile processes and methodologies [Schuh 2005].

6.1. DSDM (1995, 2003)

Dynamic systems development method (DSDM) was first introduced in 1995 by a consortium of UK companies. Motivated by an ever-increasing need for a standard, generally-accepted rapid application development (RAD) 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 7 phases [DSDM Consortium 2003]; the first and last, though, are not considered main phases, since they are not considered part of the project itself:

(1) *preproject*: the focus is on providing the necessary resources for starting the project, along with a plan for the next immediate phase: the feasibility study;

(2) *project-proper*: 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 18):

  (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*: the focus is 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:

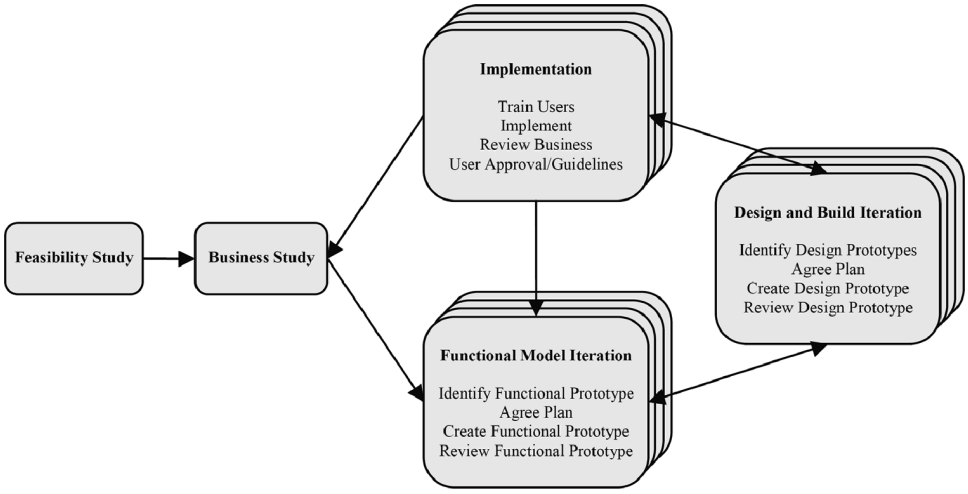

Fig. 18. The DSDM process: the five main phases of the framework—adapted from DSDM Consortium [2003].

(2.2.1) *functional model iteration*: the focus is 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*: the focus is on evolving the prototypes into final deliverable increments of the system;
(2.2.3) *implementation*: the focus is on deploying the deliverable increments into the operational environment, and reviewing and validating the system built so far;

(3) *post-project*: the focus is 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 18, 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 subteams 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 at 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 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 (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 artifacts, 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 the DSDM.

6.1.1. 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 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 nonexhaustive 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.

6.1.2. 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 nonfunctional requirements of the system; the requirements are prioritized according to what DSDM calls the **M**o**SC**o**W** 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.

6.1.3. 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/behavioral modeling notations).

—Nonfunctional 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.

These 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).

6.1.4. 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 nonfunctional 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.

6.1.5. Implementation (DSDM). During this phase of the project (which could well have 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 nonfunctional 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.

6.2. 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

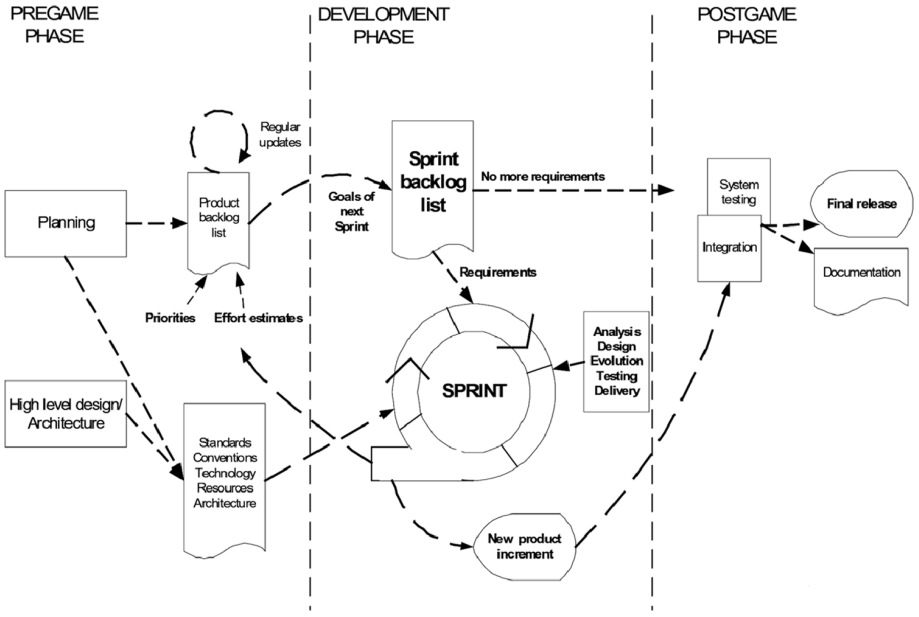

Fig. 19. The Scrum process [Abrahamsson et al. 2002].

conference [Schwaber 1995]. It was influenced by the Patterns movement of the 1990s [Coplien 2006], and has in turn inspired process patterns of its own [Beedle et al. 2000]. 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, 2007].

The Scrum process consists of three phases [Schwaber and Beedle 2001], as shown in Figure 19:

(1) *pregame*: concerned with setting the stage for the iterative-incremental development effort; this phase consists of the following subphases:
   (1.1) *planning*: focuses 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*: focuses 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)*: focuses 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*: focuses 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.

6.2.1. 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 nonfunctional 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 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.

6.2.2. 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. The sprint backlog is therefore 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 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: 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. Another objective of the sprint review meeting is to discuss and resolve issues impeding the progress of the development team. The meeting is also concerned with updating the system architecture according to the insights gained during the sprint.

6.2.3. 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.

6.3. 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 [Wells 2003]. The XP lifecycle consists of six phases (Figure 20):

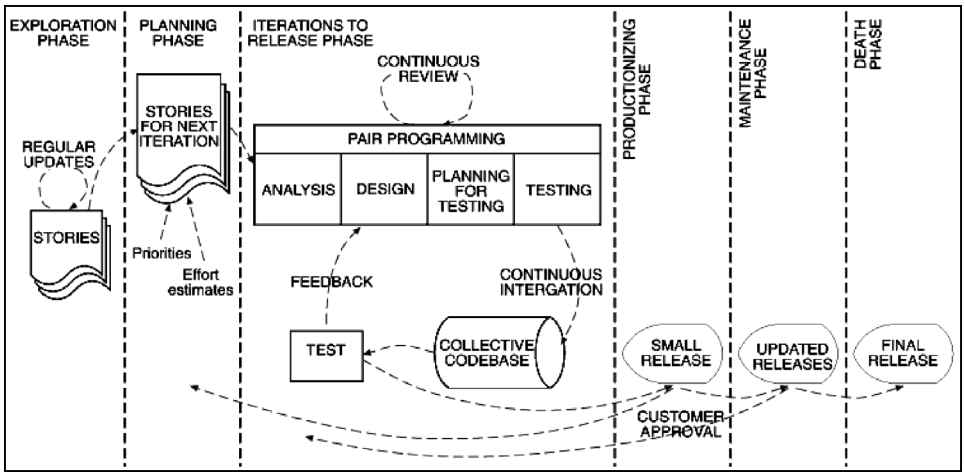

Fig. 20. A general overview of the typical XP process [Abrahamsson et al. 2002].

(1) *exploration*: focuses 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): focuses 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*: focuses 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*: focuses on system-wide verification and validation of the first release, and its deployment into the user production environment;
(5) *maintenance*: focuses 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*: focuses on closing the project and conducting post-mortem review and documentation.

The activities performed in the second, third and fourth phases of the process constitute the development engine of the XP methodology, 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; it is then incrementally improved and complemented during the maintenance phase through further iterations (runs) of the development engine.

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

6.3.1. 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) who 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 (about three sentences) of a certain chunk of functionality needed to be delivered by the system. User stories are only detailed enough to allow a 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.

6.3.2. 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.

6.3.3. 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:

(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 regarding 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.

(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 from one to three 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 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.

(2.2) Analysis, design, coding, testing, and integration in a Collective-Code-Ownership environment. 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 to the code by automated tools; the defects detected are relegated to the next iterations if time constraints do not allow their rectification in the present cycle.

Other rigorous development practices prescribed by XP 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 4.3.1) in order to come up with the simplest design possible for the programming task at 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.

6.3.4. 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.

6.3.5. Maintenance (XP). This post-deployment phase of the XP process encompasses the same activities as those in the previous three phases (the development engine): 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 no longer lends itself to necessary evolution.

6.3.6. 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), which provides a brief tour of the system, and writing a review report summarizing the lessons learned from the project.

6.4. ASD (1997, 2000)

Adaptive software development (ASD) was introduced by James Highsmith [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.

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. 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 becomes the basic ASD framework for developing software systems.

ASD goes further than just specifying 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*: focuses 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*: focuses 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*: focuses on concurrent design and implementation of the components assigned to individual cycles.
  (2.3) *quality review*: focuses on conducting group reviews of the components produced and rectifying the problems confronted.
(3) *final Q/A and release*: focuses on validating the produced system and deploying it into the working environment.

Figure 21 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.

6.4.1. 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 artifacts, consisting of the following (the necessary information for producing the artefacts is usually obtained through JAD sessions):
  (a) Project Vision (Charter), which sets boundaries on the following:
  —scope, size, and context of the project;
  —resources allocated to the project;

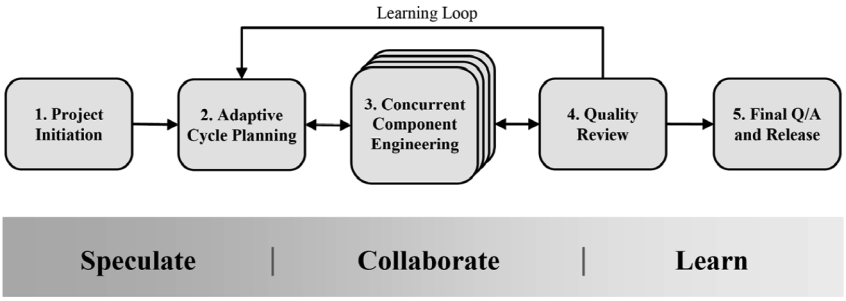

Fig. 21. The ASD process—adapted from Highsmith [2000a].

—project staff; defining the skills, knowledge, and authority required to successfully execute the project;
—communication among the people involved in or affected by the project, called 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 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.

(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.

6.4.2. 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, which enact the business

logic of the system; and technology components and support components, which are domain-independent design components acting 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. 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.

6.4.3. 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 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.

6.4.4. 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. Feedback and change requests are carefully documented in order to be considered in later iterations.

(2) Determine next step: a 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. Problems are then rectified so as not to adversely affect the next iterations.

6.4.5. 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.

6.5. 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*: focuses 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*: focuses on iterative and incremental design and coding of higher-priority (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*: focuses 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*: focuses 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.

6.5.1. 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 artifacts that are 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.

6.5.2. 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.

6.5.3. 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.

6.5.4. 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 release 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

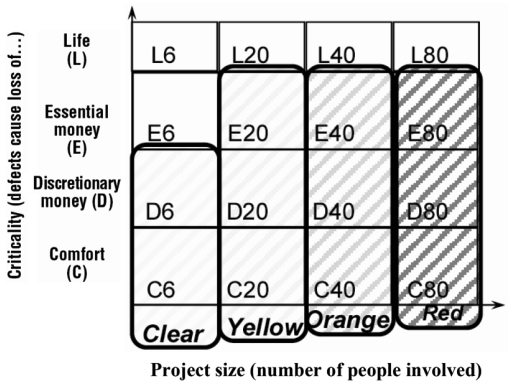

Fig. 22. Project types in Crystal and the corresponding Crystal methodologies (partial grid)—adapted from Cockburn [2001].

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.

6.6. 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, 2006]. 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 color showing its relative complexity: the heavier the methodology, the darker the colour assigned to it. Figure 22 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: clear, yellow, orange, and red, in ascending order of complexity. Other more heavyweight members of the family—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.

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 at hand: the development team(s) select 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 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*: the main development engine of the process, this phase 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 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 23 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.

6.6.1. 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;

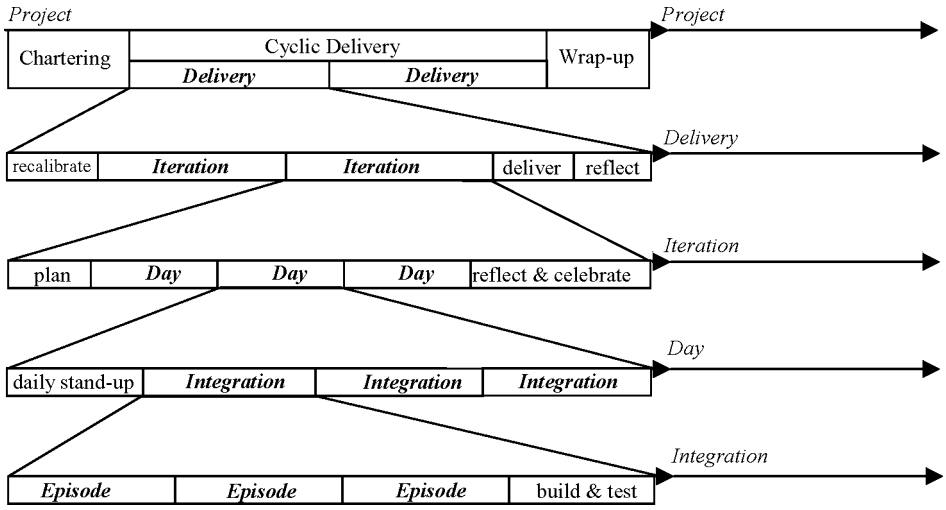

Fig. 23. Example of phases, cycles, and activities in Crystal Clear—adapted from Cockburn [2004].

(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 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 at 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.

6.6.2. 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 23). 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 reflects 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.

6.6.3. 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.

6.7. FDD (1999, 2002)

De Luca and Coad introduced Feature-Driven Development (FDD) 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

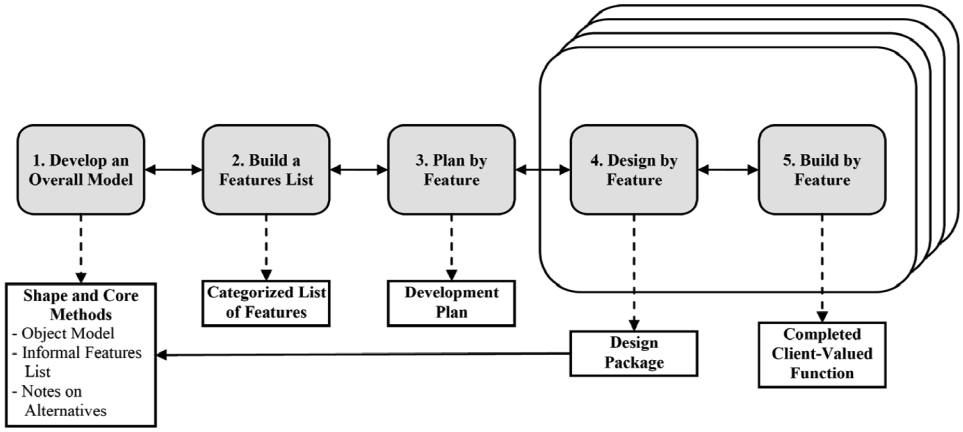

Fig. 24. The FDD process and its deliverables—adapted from Palmer and Felsing [2002].

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 (major feature sets). This three-layered structure allows the developers to adequately manage the complexity of the requirements. Furthermore, features can 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 24). 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 24, 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 25). The subprocesses, in the order they are performed, are as follows:

   (1.1) *develop an overall model*: focuses 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 behavioral 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*: focuses 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.

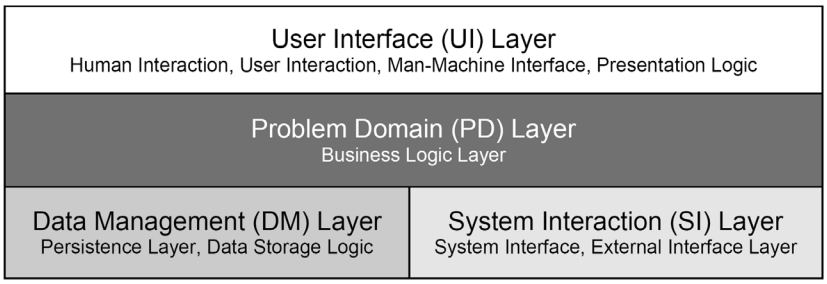

Fig. 25. The general layered architecture of software systems as proposed by FDD—adapted from Palmer and Felsing [2002].

(1.3) *plan by feature*: focuses 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*: focuses 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*: focuses 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.

The FDD methodology cannot be considered an all-inclusive software development methodology, since 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.

6.7.1. 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 analyzed and modeled through tasks 2.1 to 2.4 (explained later); the resulting submodel 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 interrelationships 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 a 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.

6.7.2. 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.

6.7.3. 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, directs the development effort and 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 interdependencies 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.

6.7.4. 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 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 they exist, in order to obtain a better understanding of the features. As with the previous task, this 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 artifacts 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 artifacts 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, the design notes (on the alternatives explored, constraints, and assumptions).

6.7.5. 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.

7. 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.

7.1. Introduction

The first recorded reference to the term "process pattern" was made by Coplien in his landmark paper in 1994 [Coplien 1994]. He defined process patterns as "the patterns of activity within an organization (and hence within its projects)"; 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 [Coplien 1994; Coplien and Harrison 2005]. 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].

Process patterns have also been introduced in a methodology-specific context: the process components of OPF [Firesmith and Henderson-Sellers 2001] and the process patterns based on Scrum [Beedle et al. 2000] are prominent examples.

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

7.2. 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;

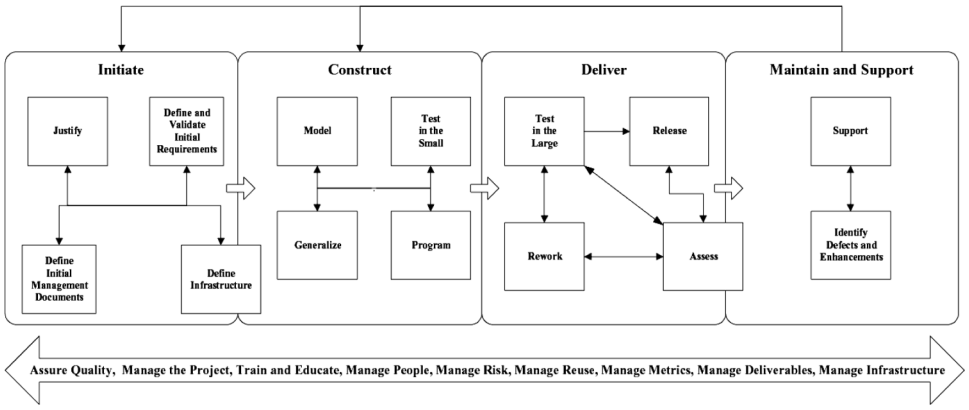

Fig. 26. Ambler's Object Oriented Software Process (OOSP) [Ambler 1998a].

(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, 1999].

7.3. 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 26, 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.

8. 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 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 in our description of the OPEN methodology. A brief overview of SPEM is given in the following sections.

8.1. 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.

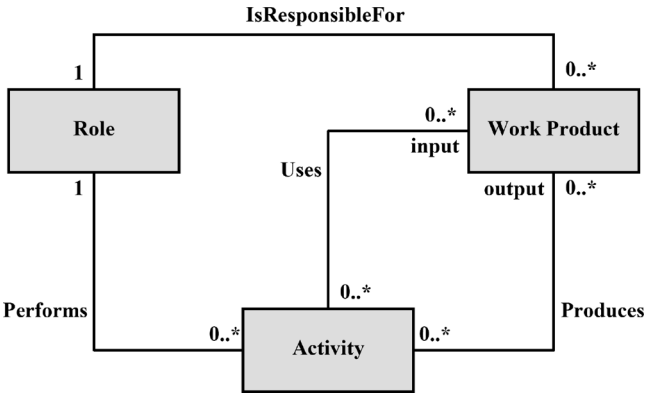

Fig. 27. Core structure of a software development process, as defined by SPEM [OMG 2002].

8.2. 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 artifacts, called *work products*, until the artifacts 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 27).

The complete structure of a process in SPEM is actually much more complex than the core structure mentioned here. A work product may be composed of other work products, and can be associated with a state machine showing the possible states for the work product, 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 subactivities 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 behavioral commonalities of the various types of process elements. It also includes well-formedness rules, to be observed when instantiating processes.

9. OBSERVATIONS AND CONCLUSIONS

This review of a selection of object-oriented methodologies, process patterns, and process metamodels (as listed in Table I) has not only formed a foundation for further analyses, but has also resulted in several observations and conclusions as to the status quo, problems afflicting the domain, and the road ahead, a brief summary of which is presented in the following sections.

9.1. Status Quo

The integration and standardization efforts of the mid 1990s, which resulted in the development of the UML and third generation methodologies, promised an end to the methodology war, but the present situation is far from what was initially expected.

Table 1. Object-Oriented Methodologies, Process Patterns and Process Metamodels Reviewed in this Article

Methodologies	Seminal	Shlaer-Mellor (1988, 1992) Coad-Yourdon (1989, 1991) RDD (1990) Booch (1991, 1994) OMT (1991) OSA (1992) OOSE (1992) BON (1992, 1995) Hodge-Mock (1992) Syntropy (1994) Fusion (1994)
	Integrated	OPM (1995, 2002) Catalysis (1995, 1998) OPEN (1996) RUP/USDP (1998, 1999, 2000, 2003) EUP (2000, 2005) FOOM (2001, 2007)
	Agile	DSDM (1995, 2003) Scrum (1995, 2001) XP (1996, 2004) ASD (1997, 2000) dX (1998) Crystal (1998, 2004) FDD (1999, 2002)
Process Patterns		Ambler (1998)
Process Metamodels		OPF—as part of the OPEN methodology (2001) SPEM (2002)

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. The integrated, heavyweight methodologies are very complex, and some of their competitors are little more than controlled code-and-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 process as possible.

The evolution process seems to have gone astray, and as a result, we are witnessing the comeback of some of the old 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 2007], 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—PIM) into physical implementation models (called Platform Specific Models—PSM) [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 vigor in the few such methodologies so far introduced [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 OO 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.

The situation definitely calls for endeavors aimed at further evolution of the methodologies, but this time with moderation in mind, and with a more systematic approach. The famous phrase "software processes are software too" [Osterweil 1987, 1997], quoted so many times over the years, has seemingly had little success in convincing methodologists that an engineering approach to development of methodologies is the natural choice.

9.2. Problem Areas

Despite all the advances, a closer look at the present state of affairs in the field of object-oriented software development methodologies shows the following problems:

(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, 2005; Coram and Bohner 2005; Nerur et al. 2005; Turk et al. 2005; and Boehm 2006]; the following are some of the more commonly cited problems:
   —unrealistic assumptions;
   —lack of scalability;
   —lack of a specific, unambiguous process.

(5) Seamless development, pioneered by seminal methodologies, is not adequately appreciated and supported in modern-day methodologies [Paige and Ostroff 2002].

9.3. The Road Ahead

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; Booch et al. 2007]. The status of the field clearly shows potential for improvement through addressing these issues. There is motivation for developing methodologies that use the lessons learned 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 can be pointed out as general characteristics of such methodologies:

(1) *compactness*: an extensible core is preferable to a customizable 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 artifacts should be traceable to the requirements;
(4) *consistency*: artifacts produced should not be allowed to contradict each other; alternatively, there should be mechanisms for detecting inconsistencies;
(5) *testability of the artifacts from the start*: this will allow tools to be developed to verify and validate the artifacts;
(6) *tangibility of the artifacts*: producing executable material as early as possible in the lifecycle;
(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 artifact produced; the developers should be able to see this logic, truly sensing that any digression will put their objectives at risk.

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 method for developing the target 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, and 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. The advent of a revolutionary methodology in this sense does not necessarily indicate the occurrence of a Kuhnian revolution: preexisting methodologies might coexist 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 methodologies. New ideas are always present, yet their dependence on ideas borrowed from existing methodologies is such that they cannot be classified 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 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 usually 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 preplanned, objective analysis of the features in existing methodologies; rather, features are scavenged from favorite methodologies in order to fill the needs of the methodologist. Fusion [Coleman et al. 1994] is a good example.

—*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. 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 they have been rather too particular in choosing their methodologies for analysis. Furthermore, there is little trace of disciplined and clear-cut design, implementation, and test activities performed in developing the methodology [Hodge and Mock 1992; 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. Therefore 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. Furthermore the instability caused by the eccentric leanings of integrated methodologies and agile methods as the main contenders 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 should not 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 monstrosity is not appealing.

Contaminated with favoritism 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 to those favored 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. Although it might seem that the direction of this discussion has been such as to justify the engineering approach via elimination of alternatives, 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, 1997],

so 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 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.

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* (*ICSE*). 244–254.

ALABISO, B. 1988. Transformation of dataflow analysis models to object oriented design. In *Proceedings of the Conference on Object-Oriented Programming Systems, Language, and Applications* (*OOPSLA'88*). 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. Available online at http://www.ambysoft.com/processPatterns.pdf.

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 online at `http://www.enterpriseunifiedprocess.info/downloads/eupIntroduction.pdf`.

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.

AVISON, D. E. AND FITZGERALD, G. 2003a. *Information Systems Development: Methodologies, Techniques and Tools*, 3rd ed. McGraw-Hill, New York, NY.

AVISON, D. E. AND FITZGERALD, G. 2003b. Where now for development methodologies? *Commun. ACM 46*, 1 (January), 79–82.

BECK, K. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA.

BECK, K. AND ANDRES, C. 2004. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison Wesley, Reading, MA.

BECK, K., BEEDLE, M., VAN BENNEKUM, A., COCKBURN, A., EUNNINONAM, W., FOWLER, M., HIGHSMITH, J., HONI, A., JEFFRIES, R., KERN, J., MARICK, B., MARTIN, R. C., SCHWABED, K., SUTHERLAND, J., AND THOMAS, D. 2001. Manifesto for agile software development. Available online at `http://agilemanifesto.org`.

BEEDLE, M., DEVOS, M., SHARON, Y., SCHWABER, K., AND SUTHERLAND, J. 2000. SCRUM: A pattern language for hyperproductive software development. In *Pattern Languages of Program Design 4*. N. HARRISON, B. FOOTE, AND H. ROHNERT, Eds. Addison-Wesley, Reading, MA. 637–651.

BOEHM, B. 2006. Some future trends and implications for systems and software engineering processes. *Syst. Eng. 9*, 1 (Spring), 1–19.

BOEHM, B. AND TURNER, R. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley, Reading, MA.

BOEHM, B. AND TURNER, R. 2005. Management challenges to implementing agile processes in traditional development organizations. *IEEE Softw. 22*, 5 (September/October), 30–39.

BOOCH, G. 1986. Object-oriented development. *IEEE Trans. Softw. Eng. 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, MA. Chapter on RUP and dX is available online 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, MA.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 2005. *Unified Modeling Language—User's Guide*, 2nd ed. Addison-Wesley, Reading, MA.

BOOCH, G., MAKSIMCHUK, R. A., ENGEL, M. W., YOUNG, B. J., CONALLEN, J., AND HOUSTON, K. A. 2007. *Object-Oriented Analysis and Design with Applications*, 3rd ed. Addison Wesley, Reading, MA.

BRINKKEMPER, S. 1996. Method engineering: engineering of information systems development methods and tools. *Inform. Softw. Tech. 38*, 4, 275–280.

CAPRETZ, L. F. 2003. A brief history of the object-oriented approach. *ACM SIGSOFT Softw. Eng. Notes 28*, 2 (March).

CHEESMAN, J. AND DANIELS, J. 2001. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, Reading, MA.

COAD, P. AND NICOLA, J. 1993. *Object-Oriented Programming*. Yourdon Press/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.

COAD, P., LEFEBVRE, E., AND DE LUCA, J. 1999. *Java Modeling in Color with UML: Enterprise Components and Process*. Prentice Hall, Englewood Cliffs, NJ.

COCKBURN, A. 1998. *Surviving Object-Oriented Projects: A Manager's Guide*. Addison-Wesley, Reading, MA.

COCKBURN, A. 2001. *Agile Software Development: Software through People*. Addison-Wesley, Reading, MA.

COCKBURN, A. 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, Reading, MA.

COCKBURN, A. 2006. *Agile Software Development: The Cooperative Game*, 2nd ed. Addison-Wesley, Reading, MA.

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)*.

COPLIEN, J. O. 2006. For those who were Agile before Agile was cool. Presented at *ØREDEV'06 Conference*. Available online at `http://www.oredev.org/download/18.5bd7fa0510edb4a8ce4800028067/James_Coplien_-_Keynote.pdf`.

COPLIEN, J. O. AND HARRISON, N. B. 2005. *Organizational Patterns of Agile Software Development*. Prentice Hall, Englewood Cliffs, NJ.

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.

DEMARCO, T. 1978. *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, NJ.

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 behavior. *J. Logic 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. *Commun. ACM 45*, 11 (November), 82–85.

DSDM CONSORTIUM. 2003. *DSDM: Business Focused Development*, 2nd ed. J. Stapleton, Ed. Addison-Wesley, Reading, MA.

D'SOUZA, D. F. AND WILLS, A. C. 1995. Catalysis—practical rigor and refinement: Extending OMT, Fusion, and Objectory. Available online at `http://www.catalysis.org/publications/papers/1995-catalysis-fusion.pdf`.

D'Souza, D. F. and Wills, A. C. 1998. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, MA.

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.

Firesmith, D. and Henderson-Sellers, B. 2001. *The OPEN Process Framework: An Introduction*. Addison-Wesley, Reading, MA.

Fowler, M. 2004. Model Driven Architecture. Available online at `http://martinfowler.com/bliki/ModelDrivenArchitecture.html`.

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, MA.

Graham, I., Henderson-Sellers, B., and Younessi, H. 1997. *The OPEN Process Specification*. ACM Press/Addison-Wesley, New York, NY.

Henderson-Sellers, B. and Graham, I. 1996. OPEN: Toward method convergence? *IEEE Comput. 29*, 4 (April), 86–89.

Henderson-Sellers, B., Simons, A., and Younessi, H. 1998. *The OPEN Toolbox of Techniques*. ACM Press/Addison-Wesley, New York, NY.

Henderson-Sellers, B. and Unhelkar, B. 2000. *OPEN Modeling with UML*. Addison-Wesley, Reading, MA.

Highsmith, J. 1997. Messy, exciting, and anxiety-ridden: Adaptive software development. *Amer. 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. *Softw. Test. Qual. Eng. 2*, 4 (July/August), 22–28.

Highsmith, J. 2002. *Agile Software Development Ecosystems*. Addison-Wesley, Reading, MA.

Hodge, L. R. and Mock, M. T. 1992. A proposed object-oriented development methodology. *Softw. Eng. J. 7*, 2 (March), 119–129.

Jacobson, I. 1987. Object-oriented development in an industrial environment. In *Proceedings of the Conference on Object-Oriented Programing Systems, Languages, and Applications* (*OOPSLA'87*). 183–191.

Jacobson, I., Booch, G., and Rumbaugh, G. 1999. *Unified Software Development Process*. Addison-Wesley, Reading, MA.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA.

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.

Kroll, P. and Kruchten, P. 2003. *The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process*. Addison-Wesley, Reading, MA.

Kruchten, P. 2001. A process engineering metamodel. Available online at `http://www.forsoft.de/zen/sdpp02/papers/Kruc01.pdf`.

Kruchten, P. 2003. *Rational Unified Process: An Introduction*, 3rd ed. Addison-Wesley, Reading, MA.

Lang, N. 1993. Shlaer-Mellor object-oriented analysis rules. *Softw. Eng. Notes 18*, 1 (January), 54–58.

Lano, K., France, R., and Bruel, J. 2000. A semantic comparison of Fusion and Syntropy. *Comput. J. 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. *Softw. Eng. J. 7*, 2 (March), 114–118.

Monarchi, D. E. and Puhr, G. I. 1992. A research typology for object-oriented analysis and design. *Commun. ACM 35*, 9 (September), 35–47.

Nerson, J. 1992. Applying object-oriented analysis and design. *Commun. ACM 35*, 9 (September), 63–74.

Nerur, S., Mahapatra, R., and Mangalaraj, G. 2005. Challenges of migrating to agile methodologies. *Commun. 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 (ICSE 2000)*. 35–46.

OMG 2001. *Model Driven Architecture (MDA)*. Object Management Group (OMG). Available online at `http://www.omg.org/docs/ormsc/01-07-01.pdf`.

OMG 2002. *Software Process Engineering Metamodel Specification (v1.0)*. Object Management Group (OMG). Available online at http://www.omg.org/technology/documents/formal/spem.htm.

OMG 2003. *Unified Modeling Language Specification (v1.5)*. Object Management Group (OMG). Available online at http://www.omg.org/docs/formal/03-03-01.pdf.

OMG 2004. *Unified Modeling Language Specifications (v2.0)*. Object Management Group (OMG). Available online at http://www.omg.org/technology/documents/formal/uml.htm.

OPEN Consortium. 2000. What is OPEN? Available online at http://www.open.org.au/Introduction/main.html.

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. *J. Obj. 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.

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. *J. Obj. Oriented Prog. 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, MA.

Schwaber, K. 1995. SCRUM development process. In *Proceedings of the Conference on Object-Oriented Programing Systems, Languages, and Applications (OOPSLA'95) Workshop on Business Object Design and Implementation*, available online at http://jeffsutherland.com/oopsla/schwapub.pdf.

Schwaber, K. 2004. *Agile Project Management with Scrum.* Microsoft Press, Redmond, WA.

Schwaber, K. 2007. *Enterprise 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. Available online at http://www.projtech.com/info/smmethod.pdf.

Shoval, P. 1988. ADISSA: Architectural design of information systems based on structured analysis. *Infor. Syst. 13*, 2, 193–210.

Shoval, P. 2007. *Functional and Object Oriented Analysis and Design: An Integrated Methodology*. Idea Group Publishing, Hershey, PA.

Shoval, P. and Kabeli, J. 2001. FOOM: Functional- and object-oriented analysis and design of information systems: An integrated methodology. *J. Database Manage. 12*, 1 (January–March), 15–25.

Siegel, J., and OMG. 2001. *Developing in OMG's Model Driven Architecture (MDA)*. Object Management Group (OMG). Available online at http://www.omg.org/docs/omg/01-12-01.pdf.

Thomas, D. 2004. MDA: Revenge of the modelers or UML utopia. *IEEE Softw. 21*, 3 (May/June), 22–24.

Turk, D., France, R., and Rumpe, B. 2005. Assumptions underlying agile software-development processes. *J. Database Manage. 16*, 4 (October-December), 62–87.

Walden, K. and Nerson, J. 1995. *Seamless Object-Oriented Software Architecture*. Prentice-Hall, Englewood Cliffs, NJ.

Webster, S. 1996. *On the Evolution of OO Methods*. Bournemouth University.

Wells, D. 2003. Extreme programming: A gentle introduction. Available online at http://www.extremeprogramming.org.

Wieringa, R. 1998. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv. 30*, 4 (December), 459–527.

WIRFS-BROCK, R. AND MCKEAN, A. 2002. *Object Design: Roles, Responsibilities and Collaborations*. Addison-Wesley, Reading, MA.

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, MA.

YOURDON, E., AND CONSTANTINE, L. L. 1979. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ.

Received December 2004; revised June 2006, April 2007; accepted May 2007