Chapter 1
Introduction

This book focuses on lessons learned and experience gained from transitioning to object-oriented software development. It draws on experience from applying software patterns, developing and adapting object-oriented application frameworks, developing object-oriented distributed systems, and using object-oriented techniques, such as Booch, Coad-Yourdon, Colbert, Fayad, Jacobson, Rumbaugh, Shlaer-Mellor, Unified Modeling Language (UML) and others. The lessons learned are presented from two distinct perspectives: managerial and technical. Based on our experiences, we present a transition framework to object-oriented software development.

1.1 The promise and pitfalls of object-oriented technology

The stakes are getting higher every day as more companies bet on object-oriented technology for military and commercial application development. Instead of being an academic exercise, making a commitment to the use of object-oriented technology is a very serious managerial and technical decision. This is a decision that can impact careers, projects, and entire companies.

On the ***positive side*** is the promise of a higher quality and more competitive applications. Object-oriented technology (OOT) promotes a better understanding of requirements and results in modifiable and more maintainable applications. OOT provides other benefits such as reusability, extensibility, and scalability. OOT promotes better teamwork, good communication among team members, and a way to engineer reliable software systems and applications. OOT helps to build software applications that satisfy human needs and meet the following four objectives:

1. **Operational objectives,** such as reliability and efficiency

2. **Development objectives,** such as understandability, reusability, and maintainability

3. **Managerial objectives**, such as team work, productivity, and engineering of right products

4. **Business or economical objectives**, such as cost efficiency that leads to higher return on investment (ROI).

OOT holds the promise of applications that can be quickly extended to satisfy the changing requirements of customers or increase the lifetime of the application. Software products tend to have a much longer usable lifetime than originally planned. The so-called Year 2000 problems are a result of the continued usefulness of programs their developers and users assumed would be long gone by the millennium. Significant applications are more often modified than completely rewritten and the cost of modification is quite high. OOT promotes applications that are shielded from the continual flux present in operating systems and programming language technologies. Because OOT requires a greater explicit emphasis on design with independent modules and clean interfaces, revisions to object-oriented applications do not demand the same level of systemic analysis and testing that a traditional application would. In addition, by taking a conservative approach to legacy systems, OOT can be successfully used to add new functionality to existing systems without incurring the prohibitively high costs of reverse engineering or re-engineering and redevelopment. OOT should result in applications that sell better. All

of this is the promise of OOT: one change in concept that leads to a multitude of benefits that lead to better software products.

On the ***negative side*** is the significant learning curve involved in bringing teams of system software project organizations, including requirements analysts, system engineers, software engineers, software designers, and software managers, up to a level of competence on OOT. This learning curve implies a longer time to market or a longer development time for initial projects, which is always a bitter pill to swallow. In addition to introducing a new way of developing software, object-oriented technology requires new tools, new programming languages, new metrics, and new software development processes.

Transitioning to object-oriented software engineering (OOSE) is a task with a lot of potential hazards. Transitioning to OOSE complicates the software manager's job. It requires the manager to deal with a different set of problems: staffing, training, scheduling, planning, object-oriented processes, tools, cost estimation, standards, documentation, metrics, and a transition process. While managers may already address some of these areas, OOSE requires a modified approach to this management task. The authors have already encountered many of the surprise aspects of OOT and want to introduce our readers to them and help our readers become familiar with these surprises so they are no longer surprises at all.

The future of object-oriented technology will bring us a mixture of satisfaction and disappointment. We should expect the disappointment because developing software is always a painful process when new ground is being broken by the application of new processes. The lessons learned from the satisfying experiences will help sell the technology inside the organization. Readers should seek to understand the benefits of object-oriented technology and learn how to capitalize on it for their organization's advantage. Reducing the problems will be the result of higher quality software products, such as classes that provide cross-platform portability. Tools will evolve that maximize the productivity of software personnel and minimize the process downtime or loss time from problems with the development process that cause developers to be unsure of how to continue. New methods and techniques will emerge that will help software development teams over the object-oriented technology hurdle and reduce the learning curve. Software products will help people manage complex projects more efficiently.

1.2 The reality of object-oriented technology

Although any new software project presents serious issues, a project that includes moving to object-oriented technology makes those issues more acute, partly because of the hype surrounding OO and partly because OO significantly changes the development environment:

- *It is hard to develop a good software architecture -- object-oriented or not.* In many non-object-oriented techniques the success of the project may depend on selecting the right architecture during the last part of the analysis or early in the preliminary design. With object-oriented techniques, project success may depend on identifying the right set of objects to represent the problem domain during analysis.

- *It is hard to find/buy/make good reusable software components.* Reusability is not a free by product of the development cycle. Careful consideration of future applications must be undertaken. This extra effort to achieve reusability is often sacrificed because of schedule pressures.

- *The interconnection technology between software developed with traditional methods and that developed with the various object-oriented techniques is poor.* Three situations can be identified:

    - The first case is reuse interconnection between software components that were developed using different techniques within object-oriented technology, such as Shlaer-Mellor, Colbert, or Rumbaugh. This interconnection is either difficult or impossible because the inter-object communications models for the various object-oriented techniques are inconsistent.

    - The second case is reuse interconnection between software components developed using non-object-oriented techniques and new object-oriented components. This interconnection is for all practical purposes impossible because non-object-oriented models do not provide the logical communications mechanisms which the new object-oriented application needs.

    - The third case is the reuse of object-oriented software components on new non-object-oriented application developments. This practice seems to work very well because non-object-oriented development uses the communications interconnection required by the object-oriented model.

- *Key object-oriented standards are missing.* This lack of standards is causing much confusion. It is ironic that across the various object-oriented techniques there is not a standard, consistent definition of what an object is. While proposed standard terminology has been published [Snyder93], there are numerous techniques that are based on unique meanings of commonly used object-oriented terminology.

- *There are many snakes in the grass, and snake oil sales is a major occupation. So, watch out!* New techniques are being published nearly every month. Many have not had adequate testing by applying them in multiple problem domains. Developers are getting hungry for object-oriented information and jump at anything with "object-oriented" in the title. Take care that the band wagon you jump on has a firm set of wheels (foundation) and has been fully proven.

- *Object-oriented environments and tools are not completely developed*. There has been a strong focus on tools for object-oriented analysis and design for several years. Object-oriented programming tools are starting to mature and show progress. However, object-oriented testing tools are still in their infancy. Tools that apply a consistent technique to object-oriented need to be in place for all phases of software development.

- *Object-oriented professionals (we mean real professionals) are needed*. As noted earlier, these people need to have real experience applying object-oriented techniques on real projects. Organizations need to decide if they will develop their own object-oriented professionals or contract for the services they need.

- *OOSE increases managerial problems*. -- What might have been the development of a single large process-oriented application becomes, instead, the development of a collection of objects with a potential for reuse. So OOSE tends to lead to more pieces to manage in getting to the same application, and more interface design attention needed to address future reuse possibilities.

- *Transition to OOSE is a mission with problems -- you need help.* All of the above confusion makes it very easy to get off to a false start on early attempts at applying this technology. Find someone that has the kind of experience that has led to the development of a successful object-oriented application to guide your initial efforts.

Despite all of the above challenges, object-oriented technology is showing up in every major software application area, including operating systems, languages, databases, embedded systems, project management, and more from other domains, such as aerospace, avionics, banking, and insurance. Organizations moving to object-oriented technology do not generally move back. After mastering the development discipline, developers do feel that object-oriented technology is superior.

OO is more than a fad. It is a significantly different way of developing software. Of course it uses many of the same constructs that traditional methods use, and the technology itself can be traced to Simula 67 from 30 years ago. But the detailed programming techniques of OO technology that get the most publicity–inheritance, encapsulation, and polymorphism–are far less important than the analysis and design methods that they make possible. The ability to model real-world objects and their interactions at multiple levels of becomes the basis for OO technology real value. If the full value of OO were only a few programming constructs, it would be difficult to justify the costs of the transition.

Among the difficulties with OO technology is that it doesn't eliminate the need for programming skills, and in many cases it requires software developers to have an even stronger grasp of programming details than before. Why then, do we recommend OO technology if we guarantee that the transition to it will be demanding in terms of cost, training, management, and initially efficiency? Isn't this a strong recommendation against the adoption of the technology? We think it is not for the following reasons:

- Existing software development has produced essentially monolithic applications that are hard to evolve and harder still to evaluate in terms of cost and value. System integration and testing has been such a large part of the development cost that most other cost items have been obscured. Likewise, the value-added functionality of the system is too hard to extract, making it unavailable for derivative and next-generation products. Such applications may be viewed as liabilities as much as assets. One specific value of OO technology is that functionality can be found, extracted, and reused.

- OO reduces the portion of development devoted to system integration and integration testing and places greater emphasis on the analysis and design of the system and its components. This shifted emphasis gives both the customer and the development team a better understanding of the product being built.

- The use of third-party object-oriented components has been growing dramatically to the point where there is a viable source for real pieces of an application. Often these components represent needed functionality but do not provide unique value. Examples are GUIs, printing, and networking capabilities. In house development of these capabilities do not necessarily contribute to the core competency of the organization [Prahalad90].

- Frameworks, software architectures designed to create applications from families of components, appear to offer the highest levels of reuse with all the advantages such reuse implies: lower cost, faster development, and greater reliability. In order to use framework architecture with third party components, the core application software must be object-oriented.

- OO technology lends itself to iterative and parallel development. Iterative development makes it possible to incorporate feedback from stakeholders more quickly than traditional waterfall methods would, and it allows decomposition into smaller, more understandable units. The earlier fixing of object functionality and interfaces means that object development can be more effectively parallelized and hence more quickly completed. Again, the reduced number of unintended object interactions also reduces the system integration effort.

- Finally, we feel that object-oriented technology makes itself more accessible to non-programmers. Using the traditional functional decomposition, only the high level analysis is accessible to customers and management. They must take on faith the connection between the analysis in terms of application objectives and the more detailed analysis and design in terms of programming constructs. Object-oriented analysis and design allows a deeper view. For business applications, we talk about business objects that model well understood entities such as credit checking and customers, and we describe the interactions and the data movement between objects as workflows. Similarly, other applications comprise objects that represent meaningful entities in their domain. An automobile control system might have engine and fuel pump objects, and the relationship between them would be described in a way that reflected the physical association between the real-world entities. At deeper levels of the analysis and design, more details and interactions appear, and ever greater skill in software development are required to understand the details, but there is not the abrupt and disconnected barrier between the top and lower levels that functional decomposition presents.

These reasons, when taken together, imply that a substantial change is occurring. Object-oriented development is becoming standard. Practitioners find that the adoption of OO programming is beneficial. Systems built with OO technology are inherently more flexible than traditional systems. Components and frameworks allow more reuse; more development choices, both economically and technically; and more concentration of the organization on its core competencies. Iterative development makes a closer temporal connection between development decisions and the understanding of their consequences. This technological combination can be seen as a "disruptive technology," that is, a technology that advances from a disadvantageous position with respect to the current technology to one that surpasses and replaces it [Bower95]. Although the pieces are not radically new, their combination is. Object technology is not just a set of new techniques or programming constructs; it is a different way of development that is finally developing a significant advantage over the development methods it is replacing. Object-oriented technology is thus the future of application development over a broad spectrum of problem domains.

1.3 What is the difference between standard, methodology, technique, and process?

One source of confusion within the object-oriented technology community is the variety of meanings associated with the various terms we use to discuss object-oriented techniques. This section will present the definitions of terms to be used throughout the remainder of this book to discuss and compare techniques. When discussing each technique, the published terms used to describe the

technique by its developers will be mapped into the terms defined in this section. Hopefully, this will reduce the confusion attendant with the technique's inconsistent usage of terminology.

A primary source of semantic confusion seems to center around the hierarchical set of terms used to classify the levels of detail of the ways software development is accomplished. Among these terms, we find the following: methodology, method, technique, paradigm, process, and standard.

In this book we hope to avoid contributing to further confusion by defining a small set of common English language terms and using them to discuss all of the techniques. The terms we will be using are: standard, methodology, technique, and process.

Standard:

A *standard* is "established for use as a rule or basis of comparison in measuring or judging capacity, quantity, content, extent, value, quality, etc. [*standards* of weight and measure]" [Webster70]. *Standards* imply regulations, guidelines, rules, laws, etc. Standards can dictate named methodologies, such as IEEE standards or DOD standards, such as, DOD-STD-2167A. Anyone who is working with government standards quickly learns that a *standard* alone is not sufficient for getting a task completed. The reason for this is that *standards* focus on the attributes of the results instead of how the results will be achieved.

The concept of *standards* also includes a type, model, or example commonly or generally accepted or adhered to, such as, a criterion set for usage or practices [moral *standards*] [Webster70]. *Standard* applies to some measure, principle, model, etc., with which things of the same class are compared in order to determine their quantity, value, quality, etc. [*standards* of purity in drugs] [Webster70].

Methodology:

Methodology is used to refer to the very highest levels of the way we do things. It implies a systematic process for handling the ideas that are involved in doing something. In fact, dictionaries define methodology as "the science of method, or orderly arrangement" [Webster70]. Thus this word refers to the very top-level science of defining order. *Methodology* also refers to "a system of methods, as in any particular science" [Webster70]. The particular science we are addressing is computer science. This science has a growing set of systems of methods, and the term *methodology* in computer science can refer to one of these systems of methods.

So, what are the methodologies or systems of methods, associated with computer science or software engineering? There are methodologies associated with almost every aspect of computing from booting the computer software system to operating the system and finally to shutting the computer down. The booting methodology includes methods for loading operating systems on various hardware platforms. The methodologies for operating a system include methods for user interfaces, among other things. Methodologies for shutting down systems, again, include methods for safely getting various combinations of software and hardware to point where they can be powered off without the risk of losing information or causing damage.

However, our concern is in a much more specific aspect of computing -- software engineering. In this area the methodologies group methods for getting from the concept of a software application, through the development of the application and delivery to the customer, to the retirement of the application when it is no longer needed. The methodologies we will consider here cover only the development of the software application.

A methodology might also be called a process model or macro development process. A methodology serves as the controlling framework for the micro process [Booch94]. The macro development process represents the activities of the entire development team on a scale of weeks or months at a time.

Methodology applies to object-oriented as well as non-object-oriented systems. Throughout this book we shall use the terms *methodology* or *macro process development* instead of process model or XYZ model, as shown in Figures 1.1 and 1.2.

Technique (Method):

A *method* implies a regular, orderly, logical procedure for doing something, such as, a *method* of finding software requirements [Webster70]. Other software engineering methods include: waterfall model, spiral model [Boehm84], and fountain model [Henderson-Sellers90]. A search of various references indicates that a *method* may be thought of in a number of ways. A *method* may be a disciplined process for producing software products or a particular way of applying the concepts of the methodology. In more general terms a *method* may be a way of doing anything; mode; procedure; process; especially, a regular orderly, definite, procedure or way of teaching, investigating, etc. [Webster70]. A method implies regularity and orderliness in action, thought, or expression; a system of doing things or handling ideas [Webster70]. From this we see that methods define an approach to accomplishing a task in general terms. Jacobson defines a method as a planned procedure by which a specific goal is approached step by step [Jacobson92]. Examples of a software design method are a set of work procedures, a set of notations, or a set of heuristics.

Software engineering methods can be divided into three major categories:

1. **Process-oriented methods:** Top-down functional or structured methods that concentrate on representations of software algorithms [DeMarco78, Yourdon89, Yourdon-Constantine79]

2. **Data-oriented methods:** Information engineering or data structure design techniques that concentrate on data flow and data representation [Martin90]

3. **Object-oriented methods:** Methods that concentrate on objects and the relationships between objects and their operations [Coad-Yourdon91a, 91b, Colbert89, Fayad93, 94a, 94b, Fayad98, Rumbaugh91, Selic94, Shlaer-Mellor88, 92, Rational96, Wirfs-Brock90]

The terms *technique*/*method* are interchangeable throughout this book.

The process-oriented and data-oriented techniques are traditional techniques that imply constant paradigm shifts, since they manipulate different concepts at each phase of the software development life cycle. The object-oriented technique offers a seamless process that helps in viewing the software architecture in terms of problem space elements [Nerson92].

Of these available philosophies, the object-oriented approach is currently receiving the most attention from academia as well as the commercial and DOD software sectors. Once the desire to introduce an object-oriented technique has been established, a particular OO software development technique must be selected. The available techniques provide a multitude of techniques to develop object-oriented software. Some techniques are based on structured analysis -- they begin with a functional decomposition of the system into data flow diagrams (DFDs) that are then used to derive low-level objects [Bailin 89]. These low-level objects are then combined somewhat arbitrarily into higher level objects that define the system. Other object-oriented techniques, such as Colbert's object-oriented

software development technique (OOSD), allow developers to begin with high-level abstract objects that are methodically decomposed during requirements analysis and design phases.

Figure 1.1: Engineering Process Hierarchy

The choice of a particular technique should be made based on the development team's knowledge of the system requirements and the system's operating environment. If the system is data intensive and the individual data elements are mostly understood prior to requirements analysis (such as a database system), then a data object driven, bottoms-up technique should be chosen, such as Shlaer-Mellor's Object-Oriented Analysis [Shlaer-Mellor88]. However, if the system details and data requirements are not fully understood at the outset, then a top-down requirements analysis technique should be selected that develops lower level objects from abstract objects, such as Colbert's OOSD [Colbert89]. Most of the object-oriented techniques concentrate on object-oriented analysis and do little or nothing with object-oriented design, testing, and maintenance.

Process:

A *process*, as shown in Figure 1.3, defines specifically who does what, when, and how [Fayad97]. The dictionary says it is "a particular method of doing something, generally involving a number of steps or operations" [Webster70]. Now we begin to see why there is confusion. Earlier, we saw that a method is a disciplined process, and now we see that a process is a particular method. We want to emphasize that a process implements one part of a method in sufficient detail such that the results are repeatable by any number of similarly trained individuals following the steps of the process. However, processes are generally locally documented implementations of methods. Processes tell what tools will be used to implement a method.

Figure 1.2: The Big-Picture

Processes generally define "what" needs to be done, but they are only one part of what a method defines. They may define a set of "high-level" or "low-level" activities that need to be performed during the software development effort. They are usually partially ordered by time (e.g., activity A must proceed activities B & C and activities B & C must be done concurrently). Software processes may define a set of reviews or they may define how a review is to be conducted. Any complete set of processes will list the deliverables that result from each process. Processes put object-oriented techniques to work.

Where a method or a technique defines the theory behind an approach, a process addresses the practicalities of using the method in a given development environment. A technique explains the ideas that are to be applied while a process lays out the concrete actions that have to take place. A technique can only predict results while a process might define the metrics to be used to verify result. Figure 1.4 illustrates the differences between a method and a process.

Figure 1.3: The differences Between a Method and a Process

We divide the software development processes1 into three-level processes [Fayad97], as shown in Figures 1.1 and 1.2:

Level 1: The ***macro development process*** represents a span of monthly or yearly activities and is equivalent, for example, to the OMT methodology which consists of analysis, system design, object design, and implementation. The ***macro development process*** is considered to be a highest level process and is shown in Figures 1.1 and 1.2 as equivalent to software development cycle, methodology, or paradigm.

Level 2: The ***mini development process*** represents the daily to weekly activities of a small team of developers. The ***mini development process*** represents activities dealing with a single phase, such as OMT's analysis or OMT's object design. The ***mini development process*** is shown in Figures 1.1 and 1.2 as equivalent to object-oriented technique.

Level 3: The ***micro development process*** represents the daily activities of an individual developer, such as identifying object and classes, or inspecting or reviewing a part of a document. The ***micro development process*** is considered to be a low-level process and is shown in Figures 1.1 and 1.2.

Figure 1.4: Three-Level Development Processes

Modeling

Building a model is a well established human process. We build models for most every project we undertake. The best way to convey the beauty and functionality of a new building is to first build a model for the users of the building to examine. Designers of electronic products use a schematic model to represent the circuits to be built. Physicists use models to conceptualize and visualize their theories of the physical processes in the universe.

All models are descriptions of something. They allow us to answer questions about a real thing before we build it. Models capture only those features deemed "essential" by model builders for their goals. A single thing might be represented by a large number of models.

Models can be validated (i.e., checked against the original thing) by experimenting with physical things and/or quizzing experts in a field of endeavor about a conceptual entity.

Our focus here is on a model designing approach to system development allowing an early **explicit** representation of the system to be built (functionality, data, interfaces) and to reason about the implicit properties of the system (response time, completeness). In particular, our focus will be on the graphical modeling notations, "A picture is worth a thousand words".

Modeling can be tangible or intangible.

Tangible modeling is a modeling technique in which special symbols denote and distinguish those things that **must be physically presented**. For example, a context diagram of a new system is a way to use tangible modeling. Intangible modeling abstracts away the physical content of things and focuses on their properties and behavior. It is a technique that frees the observer from the consideration of limitations inherent in real-world mechanisms. An intangible model of a system represents:

1 Booch divides the software development process into two-level processes which are macro and micro that map to methodologies and techniques. [Booch 94]

- What type of system inputs are converted into what type of outputs
- When, in a sequence of processing a conversion take place
- What needs to be stored in the system between processing
- Which things are dependent upon each other for information flows

The major properties of an essential model

The model is an abstraction of reality and lets you see the relationship between the parts and the whole. Modeling is a well-established human activity. All models are descriptions of something (i.e., a representation that is not the real thing), that allow us to answer questions about the real thing, that capture only those features deemed essential by the modeler, and that can be validated by experimenting with physical things or by quizzing experts. A single thing can be represented by a large number of models. There are two types of modeling: intangible modeling (e.g., logical models, behavior models, object models) and tangible modeling (e.g., physical models).

The logical model represents the key abstractions and mechanisms that define the system's architecture. The logical model also describes the system behavior and defines the roles and responsibilities of the objects that carry out the system behavior. Logical modeling frees the analysts from the consideration of limitations inherent in real-world mechanisms. Colbert uses object-interaction diagrams and object-class diagrams to illustrate the key abstractions in systems and STDs (State Transition Diagrams) and state charts to model the behavior of the system [Colbert89]. OMT and Shlaer-Mellor both use object model and STDs for the same purposes [Rambaugh91, Shlaer-Mellor92].

The physical model of the system uses symbols to represent things that must be physically present. In the software sense. The physical model is used to describe either the system's context or implementation. Colbert and O-ET use system context diagrams as a physical model of the system [Rambaugh91, Fayad94]. There are six properties essential to any good model.

1. Simple -- This property covers those attributes of the object-oriented model that present modeling aspects of the problem domain in the most understandable manner. This property measures the technique complexity in terms of number of process steps, notational aspects, constraints and design rules.

2. Complete (most likely to be correct) – This property ensures that model artifacts are free of conflicting information and all the required information is present. For example, component names within the model should be uniform and no incomplete sections of the model should exist. This property determines if the object-oriented model provides internal consistency and completeness of the model’s artifacts. The model must be able to convey the essential concepts of the its properties.

3. Stable to technological change – Unfortunately, object-oriented models are fuzzy due to the absence of quantitative heuristics, and most OO models are built upon false assumptions.

4. Testable -- To be testable, the model must be specific, unambiguous, and quantitative wherever possible. For our purposes we define simulation as an imitation of the actual model. This definition leads us to validate the characteristics of the model against the user's requirements.

5. Easy to understand -- In addition to the familiarity of the modeling notations, the notational aspects, design constraints, and analysis and design rules of the model should be simple and easy to understand by the customers, users, and domain experts.

6. Visual or graphical -- A picture worth a thousand words. As a user you can visualize and describe the model. The graphical model is essential for visualization and simulation.

1.4 Summary

We believe that software development is undergoing a dramatic change and that object-oriented technology is a key piece of the change. While there are many difficult issues to address in the transition to the new technology, we feel that staying behind is not an option. Object-oriented software is much more than a fad; it is a fundamental reordering of software development processes that will manifest itself in new technological and economic realities. Our aim in this book is to alert you to the larger issues surrounding the move to technology.

The first hurdle in understanding object-oriented technology is to understand some of the commonly used terms. In brief, the following points outline the relationships between concepts of standard, methodology, method, and process that we shall use in this book to discuss our experiences with a collection of object-oriented techniques:

- Defined processes are the baseline for improvements: You can't improve anything that you can't repeat. It is very difficult to repeat anything without a documented process.

- Software process hierarchy:

  - **Standard Level** - Industry/Government --

    IEEE standards

    ISO 9000 series

    Military standards

    DOD standards

    FAA Standards

    FDA Standards

- **The *Macro Development Process*** corresponds to company process that a collection of processes that conform to one or more of the above standards.

- **The *Mini Development Process*** corresponds to object-oriented techniques that are used by different projects.

- **The *Micro Development Process*** corresponds to the software development team process within a project.

  - *Macro/mini/micro* - Processes are especially important for new object-oriented development teams to maximize individual contribution.

  - *Macro/mini/micro* - Processes interpret the applicable standards in the company or project environment.

  - *Macro* - Processes must be tuned for specific projects. This process tailoring is required in order to satisfy unique project requirements within the context of company standards.

  - *Mini*- Processes must be tuned for specific problem domain. This process tailoring is required in order to satisfy unique domain requirements within the context of company standards.

  - *Micro* - Processes are often treated as a methodology or method. This creates confusion.

1.5 Organization of this Book

This book is organized into five major parts based on the transition framework: A Framework (Part 1), Planning & Pre-Project Activities (Part 2), Object-Oriented Insertion Activities (Part 3), and Object-Oriented Project Management Activities (Part 4). Part One includes two (2) chapters that define several poorly understood OO terms and concepts and describe a complete transition framework. Part Two has Chapters 3 and 4, which include activities to condition the development environment for project start-up. Part Three, comprising Chapters 5 through 10, describes actions to identify and plan for the required OO resources. Part Four consisting of Chapters 11 through 18, describes activities to monitor and provide direction on the project and to deal with software quality assurance and software configuration management.

References

Bailin89] Bailin, S. C., "An Object-Oriented Requirements Specification Method," the Communications of the ACM, Vol. 32, No. 5, May 1989, pp. 608-623.

[Boehm84] Boehm, B.W., "Verifying and Validating Software Requirements and Design Specifications", IEEE Software, Vol. 1, No. 1, January 1984, pp. 75-88. Reprinted in R.H. Thayer and M. Dorfman (eds.), Tutorial: System and Software Requirements Engineering, IEEE Computer Society Press, Washington, DC, 1990.

[Booch94] Booch, G., Object-Oriented Analysis and Design, 2st. Ed., Benjamin/Cummings Publishing Company, 1994.

[Bower95] Bower, Joseph L., and Christensen, Clayton M., Disruptive Technologies: Catching the Wave," Harvard Business Review, January-February, 1995.

[Coad-Yourdon91a] Coad, P. and E. Yourdon, Object-Oriented Analysis. 2nd edition, Prentice-Hall, Englewood Cliffs, N.J., 1991.

[Coad-Yourdon91b] Coad, P. and E. Yourdon, Object-Oriented Design, Prentice-Hall, Englewood Cliffs, N.J., 1991.

[Colbert89] Colbert, E., "The Object-Oriented Software Development Method: A Practical Approach to Object-Oriented Development," Ada Technology in Context: Application, Development, and Deployment Proceedings, ACM TRI-Ada '89, October, 1989, pp. 400-415.

[DeMarco78] DeMarco, T., Structured Analysis and System Specifications, Yourdon Inc., New York, 1978.

[Fayad93] Fayad, M.E., et al., "Using the Shlaer-Mellor Object-Oriented Method," IEEE Software, March 1993, pp. 43-52.

[Fayad94a] Fayad, M.E. et al., "Adapting an Object-Oriented Development Method" IEEE Software, May 1994.

[Fayad94b] Fayad, M.E., W.T. Tsai, R. Anthony, and M.L. Fulghum, "Object Modeling Technique (OMT): Experience Report," Accepted and will appear in the Journal of Object-Oriented Programming (JOOP), September 1994.

[Fayad97] Fayad, Mohamed E. Software Development Processes: The Necessary Evil, The ***Communications of the ACM,*** Sept. 1997.

[Fayad98] Fayad, Mohamed E. Object-Oriented Software Development Process: A Comparative Study, a Chapter in "Elements of Software Process Assessment and Improvement," Eds. Khald El-Emam and Nazim H. Madhavji, ***IEEE Computer Society Press,*** 1998

[Henderson-Seller90] Henderson-Seller, B. and Edwards, J.M., The Object-Oriented System Life Cycle, The ***Communications of the ACM,*** Vol. 33, No. 9, Sept. 1990.

[Jacobson92] Jacobson, I., et al., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[Martin90] Martin, J. Information Engineering, Books I, II, and III, Prentice-Hall, Englewood Cliffs, N.J., 1990.

[Nerson92] Nerson, J.M. "Applying Object-Oriented Analysis and Design," Comm. ACM, Vol. 35, No. 9, September 1992, pp. 63-74.

[Prahalad90] Prahalad C. K., Gary Hamel, “The Core Competence of the Corporation,” Harvard Business Review, May-June 1990.

[Rumbaugh91] Rumbaugh J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, 1991.

[Selic94] Selic, B., P. Ward, and G. Gullekson, Real-Time Object-Oriented Modeling, John Wiley & Sons, 1994.

[Shlaer-Mellor88] Shlaer, S. and S.J. Mellor, Object-Oriented System Analysis: Modeling the World in Data, Yourdon Press Computing Series, Englewood Cliffs, N.J., 1988.

[Shaler-Mellor92] Shlaer, S. and S.J. Mellor, Object Lifecycles: Modeling the World in States, Yourdon Press Computing Series, Englewood Cliffs, N.J., 1992.

[Snyder93] Snyder, A., "The Essence of Objects: Concepts and Terms," IEEE Software, January 1993, pp. 31-42.

[Rational96] Booch, G., Rumbaugh, J., and Jacobson, I., The Unified Modeling Language for Object-Oriented Development Documentation Set, Version 0.9, Santa Calra, CA, Rational Software Corporation, 1996.

[Webster70] Webster's New World Dictionary of the American Language, The World Publishing Company, New York, 1970.

[Wirfs-Brock90] Wirfs-Brock, R., B. Wilkerson, and L. Wiener, Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, N.J., 1990.

[Yourdon89] Yourdon, E., Modern Structured Analysis, Yourdon Press, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Yourdon-Constantine79] Yourdon, E. and L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Programming and Design, 2nd edition, Prentice Hall, New York, 1979