10

Process Overview

A ***software development process*** provides a basis for the organized production of software, using a collection of predefined techniques and notations. The process in this book starts with formulation of the problem, then continues through analysis, design, and implementation. The presentation of the stages is linear, but the actual process is seldom linear.

10.1 Development Stages

Software development has a sequence of well-defined stages, each with a distinct purpose, input, and output.

- **System conception**. Conceive an application and formulate tentative requirements.
- **Analysis**. Deeply understand the requirements by constructing models. The goal of analysis is to specify *what* needs to be done, not *how* it is done. You must understand a problem before attempting a solution.
- **System design**. Devise a high-level strategy—the architecture—for solving the application problem. Establish policies to guide the subsequent class design.
- **Class design**. Augment and adjust the real-world models from analysis so that they are amenable to computer implementation. Determine algorithms for realizing the operations.
- **Implementation**. Translate the design into programming code and database structures.
- **Testing**. Ensure that the application is suitable for actual use and that it truly satisfies the requirements.
- **Training**. Help users master the new application.
- **Deployment**. Place the application in the field and gracefully cut over from legacy applications.
- **Maintenance**. Preserve the long-term viability of the application.

The entire process is seamless. You continually elaborate and optimize models as your focus shifts from analysis to design to implementation. Throughout development the same concepts and notation apply; the only difference is the shift in perspective from the initial emphasis on business needs to the later emphasis on computer resources.

An OO approach moves much of the software development effort up to analysis and design. It is sometimes disconcerting to spend more time during analysis and design, but this extra effort is more than compensated by faster and simpler implementation. Because the resulting design is cleaner and more adaptable, future changes are much easier.

Part 2 covers the first four topics and Part 3 covers implementation. In this book we emphasize development and only briefly consider testing, training, deployment, and maintenance. These last four topics are important, but are not the focus of this book.

10.1.1 System Conception

System conception deals with the genesis of an application. Initially somebody thinks of an idea for an application, prepares a business case, and sells the idea to the organization. The innovator must understand both business needs and technological capabilities.

10.1.2 Analysis

Analysis focuses on creation of models. Analysts capture and scrutinize requirements by constructing models. They specify *what* must be done, not *how* it should be done. Analysis is a difficult task in its own right, and developers must fully understand the problem before addressing the additional complexities of design. Sound models are a prerequisite for an extensible, efficient, reliable, and correct application. No amount of implementation patches can repair an incoherent application and compensate for a lack of forethought.

During analysis, developers consider the available sources of information (documents, business interviews, related applications) and resolve ambiguities. Often business experts are not sure of the precise requirements and must refine them in tandem with software development. Modeling quickens the convergence between developers and business experts, because it is much faster to work with multiple iterations of models than with multiple implementations of code. Models highlight omissions and inconsistencies so that they can be resolved. As developers elaborate and refine a model, it gradually becomes coherent.

There are two substages of analysis: domain analysis and application analysis. ***Domain analysis*** focuses on real-world things whose semantics the application captures. For example, an airplane flight is a real-world object that a flight reservation system must represent. Domain objects exist independently of any application and are meaningful to business experts. You find them during domain analysis or by prior knowledge. Domain objects carry information about real-world objects and are generally passive—domain analysis emphasizes concepts and relationships, with much of the functionality being implicit in the class model. The job of constructing a domain model is mainly to decide which information to capture and how to represent it.

Domain analysis is then followed by ***application analysis*** that addresses the computer aspects of the application that are visible to users. For example, a flight reservation screen is

part of a flight reservation system. Application objects do not exist in the problem domain and are meaningful only in the context of an application. Application objects, however, are not merely internal design decisions, because the users see them and must agree with them. The application model does not prescribe the implementation of the application. It describes how the application appears from the outside—the black-box view of it. You cannot find application classes with domain analysis, but you can often reuse them from previous applications. Otherwise, you must devise application objects during analysis as you think about interfaces with other systems and how your application interacts with users.

10.1.3 System Design

During ***system design***, the developer makes strategic decisions with broad consequences. You must formulate an architecture and choose global strategies and policies to guide the subsequent, more detailed portion of design. The ***architecture*** is the high-level plan or strategy for solving the application problem. The choice of architecture is based on the requirements as well as past experience. If possible, the architecture should include an executable skeleton that can be tested. The system designer must understand how a new system interacts with other systems. The architecture must also support future modification of the application.

For straightforward problems, preparation of the architecture follows analysis. However, for large and complex problems their preparation must be interleaved. The architecture helps to establish a model's scope. In turn, modeling reveals important issues of strategy to resolve. For large and complex problems, there is much interplay between the construction of a model and the model's architecture, and they must be built together.

10.1.4 Class Design

During ***class design***, the developer expands and optimizes analysis models; there is a shift in emphasis from application concepts toward computer concepts. Developers choose algorithms to implement major system functions, but they should continue to defer the idiosyncrasies of particular programming languages.

10.1.5 Implementation

Implementation is the stage for writing the actual code. Developers map design elements to programming language and database code. Often, tools can generate some of the code from the design model.

10.1.6 Testing

After implementation, the system is complete, but it must be carefully tested before being commissioned for actual use. The ideas that inspired the original project should have been nurtured through the previous stages by the use of models. Testers once again revisit the original business requirements and verify that the system delivers the proper functionality. Testing can also uncover accidental errors (bugs) that have been introduced. If an application runs on multiple hardware and operating system platforms, it should be tested on all of them.

Developers should check a program at several levels. Unit tests exercise small portions of code, such as methods or possibly entire classes. Unit tests discover local problems and often require that extra instrumentation be built into the code. System tests exercise a major subsystem or the entire application. In contrast to unit tests, system tests can discover broad failures to meet specifications. Both unit and system tests are necessary. Testing should not wait until the entire application is coded. It must be planned from the beginning, and many tests can be performed during implementation.

10.1.7 Training

An organization must train users so that they can fully benefit from an application. Training accelerates users on the software learning curve. A separate team should prepare user documentation in parallel to the development effort. Quality control can then check the software against the user documentation to ensure that the software meets its original goals.

10.1.8 Deployment

The eventual system must work in the field, on various platforms and in various configurations. Unexpected interactions can occur when a system is deployed in a customer environment. Developers must tune the system under various loads and write scripts and install procedures. Some customers will require software customizations. Staff must also localize the product to different spoken languages and locales. The result is a usable product release.

10.1.9 Maintenance

Once development is complete and a system has been deployed, it must be maintained for continued success. There are several kinds of maintenance. Bugs that remain in the original system will gradually appear during use and must be fixed. A successful application will also lead to enhancement requests and a long-lived application will occasionally have to be restructured.

Models ease maintenance and transitions across staff changes. A model expresses the business intent for an application that has been driven into the programming code, user interface, and database structure.

10.2 Development Life Cycle

An OO approach to software development supports multiple life-cycle styles. You can use a waterfall approach performing the phases of analysis, design, and implementation in strict sequence for the entire system. However, we typically recommend an iterative development strategy. We summarize the distinction here and elaborate in Chapter 21.

10.2.1 Waterfall Development

The waterfall approach dictates that developers perform the software development stages in a rigid linear sequence with no backtracking. Developers first capture requirements, then construct an analysis model, then perform a system design, then prepare a class design, fol-

lowed by implementation, testing, and deployment. Each stage is completed in its entirety before the next stage is begun.

The waterfall approach is suitable for well-understood applications with predictable outputs from analysis and design, but such applications seldom occur. Too many organizations attempt to follow a waterfall when requirements are fluid. This leads to the familiar situation where developers complain about changing requirements, and the business complains about inflexible software development. A waterfall approach also does not deliver a useful system until completion. This makes it difficult to assess progress and correct a project that has gone awry.

10.2.2 Iterative Development

Iterative development is more flexible. First you develop the nucleus of a system—analyzing, designing, implementing, and delivering working code. Then you grow the scope of the system, adding properties and behavior to existing objects, as well as adding new kinds of objects. There are multiple iterations as the system evolves to the final deliverable.

Each iteration includes a full complement of stages: analysis, design, implementation, and testing. Unlike the strict sequence of the waterfall method, iterative development can interleave the different stages and need not construct the entire system in lock step. Some parts may be completed early, while other, less crucial parts are completed later. Each iteration ultimately yields an executable system that can be integrated and tested. You can accurately gauge progress and make adjustments to your plans based on feedback from the early iterations. If there is a problem, you can move backward to an earlier stage for rework.

Iterative development is the best choice for most applications because it gracefully responds to change and minimizes risk of failure. Management and business users get early feedback about progress.

10.3 Chapter Summary

A software engineering process provides a basis for the organized production of software. There is a sequence of well-defined stages that you can apply to each of the pieces of a system. For example, parallel development teams might develop a database design, key algorithms, and a user interface. An iterative development of software is flexible and responsive to evolving requirements. First you prepare a nucleus of a system, and then you successively grow its scope until you realize the final desired software.

analysis	domain analysis	system conception
application analysis	implementation	system design
architecture	iterative development	testing
class design	life cycle	training
deployment	maintenance	waterfall development

Figure 10.1 Key concepts for Chapter 10