Developing software is no easy task, but don't be daunted. This article will get you started by discussing the complexity of object-oriented development and showing you how to prioritize your efforts. In the end, you will have gone through the elemental software development process.

If you've read anything about the object-oriented paradigm, then you quickly discovered that it is a software development strategy based on the idea of building systems from reusable components called objects. The primary concept behind this paradigm is that instead of defining systems as two separate parts (data and functionality), you now define systems as a collection of interacting objects. Objects do things (that is, they have functionality) and they know things (that is, they have data). Sounds simple, but it's not. Developing software is hard, and because you typically apply object technology to solve complex problems, developing systems using object technology often proves to be even harder than with more simple, structured technology. To understand how to use object-oriented artifacts, such as those of the Unified Modeling Language (UML), and additional artifacts, such as essential models and persistence models to fill in the gaps left by the UML, you need an introductory primer. That is exactly what you're going to get now.

In this article I present a minimal process, depicted in Figure 1, for developing software using object-oriented techniques. I say that it is minimal because it focuses on the core activities of software development: requirements, analysis, design, programming, and testing. It does not cover other important topics such as project management, metrics, architecture, and system deployment. Nor does it include topics that would make it a true software process, such as the operation and support of your system once it is put into production. As you will see, the basics are complicated enough; there's no need to get ahead of ourselves yet.
I like to say that software development is serial on the large scale and iterative on the small scale, delivering incremental releases over time. Taking this to heart, I will present the major object-oriented software development activities -- requirements engineering, analysis, design, programming, and testing -- in a serial manner, although you will discover almost immediately that each of these activities is actually quite iterative in practice.

**Object-oriented requirements engineering**

First, there is no such thing as "object-oriented requirements." My experience has been that requirements should be technology independent; therefore you should really be concerned only about requirements at this point in your development life cycle. Regardless of the hype surrounding "iterative development," the first step of any software development effort is to gather requirements -- you may not gather all of your requirements at once, but you should at least start with a few. You cannot successfully build a system if you're not clear on what it should do. The greatest problem during this stage is that many people do not want to invest the time to elicit requirements; instead, they want to jump right into programming. Your subject matter experts (SMEs) have their usual jobs to do and do not have the time to invest. Moreover, your developers want to get into the "real work" of coding, and senior management wants to see some progress on the project, which usually means they want to see some code written. You need to communicate with all project stakeholders that this preliminary work is critical to the success of the project and that their efforts will pay off in the long run.

Figure 2 depicts the relationships between the artifacts that you will potentially develop as part of your requirements engineering efforts. The boxes represent the artifacts and the arrows represent "drives" relationships. For example, you see that information contained in your Class Responsibility Collaborator (CRC) model drives or effects information in your essential use-case model, and vice versa (all artifacts discussed in this article are defined in the Glossary). There are several important lessons to be taken from Figure 2. First, requirements engineering is a very iterative process. Second, there is far more to engineering object-oriented requirements than writing use cases -- I find that the term *use-case driven* $X$ should be replaced with *requirements-driven* $X$ to get right to the point.
Object-oriented analysis

The purpose of analysis is to understand what will be built. This is similar to requirements engineering, the purpose of which is to determine what your users would like to have built. The main difference is that the focus of requirements gathering is on understanding your users and their potential use of the system, whereas the focus of analysis shifts to understanding the system itself. Figure 3 depicts the main artifacts of your analysis efforts and the relationships between them. The solid boxes indicate major analysis artifacts, whereas the dashed boxes represent your major requirements artifacts. As before, the arrows represent "drives" relationships -- for example, you see that information contained in your CRC model drives or effects information in your class model, and vice versa. There are three important implications of Figure 3: first, analysis, too, is an iterative process. Second, taken together, requirements gathering and analysis are highly interrelated and iterative. As you will see, analysis and design are similarly interrelated and iterative. Third, the "essential" models, your essential use case model and your essential user interface prototype (see Software Use in Related topics), evolve into corresponding analysis artifacts; respectively, your use case model and user interface prototype. Similarly, your Class Responsibility Collaborator (CRC) model evolves into your analysis class model.
An important concept that needs to be clarified regarding Figure 3 and similar figures throughout this article is that every possible "drives" relationship is not shown. For example, it is very likely that as you are developing your use case model you will realize that you are missing a feature in your user interface, yet there is not a relationship between these two artifacts. From a purely academic point of view, when you realize that your use case model conflicts with your user interface model, you should first consider what the problem is, update your use case model appropriately, propagate the change to your essential use case model, then to your essential user interface model, then, finally, into your user interface model. Yes, you may in fact take this route. Just as likely, and probably more so, is that you will instead update both your use case model and user interface model together and then propagate the changes to the corresponding requirements artifacts. This is an important aspect of iterative development -- you do not necessarily work in a defined order; instead, your work reflects the relationships between the artifacts that you evolve over time.

Object-oriented design

The purpose of design is to determine how you are going to build your system -- information needed to drive the actual implementation of your system. This is different from analysis, which focuses on understanding what will be built. As you can see in Figure 4, your analysis artifacts, depicted as dashed boxes, drive the development of your design artifacts. As before, the arrows represent "drives" relationships -- information in your analysis (conceptual) class model drives information in your design class model, and vice versa. There are three important implications of Figure 4: first, like requirements and analysis, design too is an iterative process. Second, taken together, analysis and design are highly interrelated and iterative. As you will see in the next section, design and programming are similarly interrelated and iterative. Third, your analysis class model evolves into your design class model to reflect features of your implementation environment, design concepts such as layering, and the application of design patterns.
There are several high-level issues that you must decide on at the beginning of the design process. First, do you intend to take a pure, object-oriented approach to design or a component-based approach? With a pure OO approach your software is built from a collection of classes, whereas with a component-based approach your software is built from a collection of components. Components, in turn, are built using other components or classes (it is possible to build components from non-object technologies).

A second major design decision is whether you will follow all or just a portion of a common business architecture. This architecture may be defined by your organization-specific business or domain architecture model (See Process Patterns in Related topics), sometimes called an enterprise business model, or by a common business architecture promoted within your business community. For example, standard business models exist within the manufacturing, insurance, and banking industries. If you choose to follow a common business architecture, your design models will need to reflect this decision, showing how you will apply your common business architecture in the implementation of your business classes.

Third, you must decide whether you will take advantage of all or a portion of a common technical infrastructure. Will your system be built using your enterprise's technical infrastructure, perhaps comprised of a collection of components or frameworks? Enterprise JavaBeans (EJB) technology, CORBA, and the San Francisco Component Framework (see Related topics) are examples of technical infrastructures that you may decide to base your system on. Perhaps one of the goals of your project is to produce reusable artifacts for future projects. If this is the case, then you want to seriously consider technical architectural modeling. Although beyond the scope of this article, technical architectural modeling is a topic that I cover in Process Patterns (see Related topics).

Fourth, you must decide which non-functional requirements and constraints your system will support. You refined these requirements during analysis and, hopefully, resolved any contradictions, but it's during design that you truly begin to take them into account in your models. These requirements will typically pertain to technical services; for example, it is common to have non-functional requirements describing security access rights as well as data sharing approaches. As you try to fulfill these requirements you may find that you are unable to implement them completely. Perhaps it will be too expensive to build your system to support sub-second response time, whereas a response time of several seconds proves to be affordable. Every system has design trade-offs.
Object-oriented programming

The purpose of object-oriented programming is to build your actual system -- to develop the code that fulfills your system's design. As you can see in Figure 5, your design artifacts, depicted as dashed boxes, drive the development of your source code. The most important implication is that design and programming are highly interrelated and iterative. Your programming efforts will immediately reveal weaknesses in your design that will have to be addressed. Perhaps the designers were unaware of specific features in the programming environment and, therefore, did not take advantage of them.

Figure 5. Design artifacts drive development of source code

What isn't so obvious is that you will focus on two types of source code: object-oriented code such as Java code or C++, and persistence mechanism code such as data definition language (DDL), data manipulation language (DML), stored procedures, and triggers. Your class models, state chart diagrams, user interface prototypes, business rules, and collaboration diagrams drive the development of your object-oriented code, whereas your persistence model drives the development of your persistence code. (See "Toward a UML Profile for a Relational Persistence Model" in Related topics for more information about persistence models.)

Object-oriented testing

A fundamental rule of software engineering is that you should test as early as possible. Most mistakes are made early in the life of a project and the cost of fixing defects increases exponentially the later they are found.

Technical people are very good at things like design and coding -- that's what makes them technical people. On the other hand, technical people are often not as good at non-technical tasks such as gathering requirements and performing analysis -- perhaps another trait that makes them technical people. The end result is that developers have a tendency to make more errors during requirements definition and analysis than during design and coding.

The other motivating factor for testing early is that fixing these defects gets costlier the later they are found. This happens because of the nature of software development -- work is performed based on work performed previously. For example, modeling is performed based on the
information gathered during the definition of requirements. Programming is done based on the models that were developed, and testing is performed on the written source code. If a requirement was misunderstood, all modeling decisions based on that requirement are potentially invalid, all code written based on the models is then in question, and the testing efforts are based on verifying the application against the wrong conditions. As a result, errors detected near the end of the development life cycle or after the application has been released are likely to be very expensive to fix. On the other hand, errors that are detected early in the life cycle, where they are likely to have been made, will be much less expensive to fix because only a few documents have to be updated.

The Full-Life cycle Object-Oriented Testing (FLOOT) methodology is a collection of testing techniques to verify and validate object-oriented software. The FLOOT life cycle is depicted in Figure 6, which illustrates that there is a wide variety of techniques available to you throughout all aspects of software development. (See The Object Primer 2nd Edition, Building Object Applications That Work, and Testing Object-Oriented Systems in Related topics.) More testing, not less, is often the reality for object-oriented systems because of the greater complexity of the problem domains at which object technology is targeted.

Figure 6. The techniques of the Full Life cycle Object-Oriented Testing (FLOOT) methodology

The complexities of object-oriented development

One myth of object-oriented development is that it is much easier than structured development. However, as you see in Figure 7, an amalgamation of Figures 2 through 6, object-oriented development is actually quite complex — far more complex than most "object gurus" will care to admit. Furthermore, as I pointed out at the beginning of this article, I have only focused on the basics, leaving management, production, and cross-project issues out of the discussion.
It is possible to develop mission-critical software using object-oriented and component-based techniques and technologies; thousands of firms do it every day. You just have to accept that it is a complex effort that requires a significant range of skills. This article merely presented a primer for object-oriented development. Its purpose has been to reveal the complexity of object-oriented development to you and to underscore the fact that you need to understand both the techniques of object-oriented development as well as how they fit together. To be successful your organization will need to invest in training, education, and mentoring in addition to a new set of development tools, techniques, and processes. Do not underestimate the task before you: successful object-oriented development is rather difficult.

Glossary: The deliverables of object-oriented development

**Activity diagram.** A UML diagram that is used to model high-level business processes or the transitions between states of a class (in this respect, activity diagrams are effectively specializations of statechart diagrams).

**Business rule.** An operating principle or policy of your organization.

**Change case.** A potential requirement that your system may need to support in the future.

**Change-case model.** The collection of change cases applicable to your system. See *Designing Hard Software* (in Related topics) for details.

**Class diagram.** A UML diagram that shows the classes of a system and the associations between them.

**Class model.** A class diagram and its associated documentation.

**Class Responsibility Collaborator (CRC) card.** A standard index card that has been divided into three sections: one indicating the name of the class that the card represents, one listing the responsibilities of the class, and one listing the names of the other classes that this one collaborates with to fulfill its responsibilities.
Class Responsibility Collaborator (CRC) model. A collection of CRC cards that model all or part of a system.

Collaboration diagram. A UML diagram that shows instances of classes, their interrelationships, and the message flow between them. Collaboration diagrams typically focus on the structural organization of objects that send and receive messages.

Component diagram. A UML diagram that depicts the software components that compose an application, system, or enterprise. The components, their interrelationships, interactions, and their public interfaces are depicted.

Constraint. A restriction on the degree of freedom you have in providing a solution.

Deployment diagram. A UML diagram showing the hardware, software, and middleware configuration for a system.

Essential model. A model that is intended to capture the essence of a problem through technology-free, idealized, and abstract descriptions. See Software Use (in Related topics) for more information about essential modeling.

Essential use case. A simplified, abstract, generalized use case that captures the intentions of a user in a technology- and implementation-independent manner.

Essential use-case model. A use-case model comprised of essential use cases.

Essential user-interface prototype. A low fidelity prototype of a system's user interface that models the fundamental, abstract characteristics of a user interface.

Model. An abstraction describing a problem domain or a solution to a problem domain. Traditionally models are thought of as diagrams plus their corresponding documentation although non-diagrams, such as interview results and collections of CRC cards, are also considered to be models.

Non-functional requirements. The standards, regulations, and contracts to which your system must conform; descriptions of interfaces to external systems that your system must interact with; performance requirements; design and implementation constraints; and the quality characteristics to which your system must conform.

Persistence model. A model that describes the persistent data aspects of a software system.

Prototype. A simulation of an item, such as a user interface or a system architecture, the purpose of which is to communicate your approach to others before significant resources are invested in the approach.

Requirements model. The collection of artifacts, including your use-case model, user-interface model, domain model, change-case model, and supplementary specification that describes the requirements for your system.
Sequence diagram. A UML diagram that models the sequential logic -- in effect the time ordering of messages.

State chart diagram. A UML diagram that describes the states that an object may be in, as well as the transitions between states. Formerly referred to as a state diagram or state-transition diagram.

System use case. A detailed use case that describes how your system will fulfill the requirements of a corresponding essential use case, often referring to implementation-specific features such as aspects of your user interface design.

System use-case model. A use-case model comprised of system use cases.

Use case. A sequence of actions that provide a measurable value to an actor.

Use-case diagram. A UML diagram that shows use cases, actors, and their interrelationships.

Use-case model. A model comprised of a use-case diagram, use-case definitions, and actor definitions. Use-case models are used to document the behavioral requirements of a system.

User interface-flow diagram. A diagram that models the interface objects of your system and the relationships between them. Also known as an interface-flow diagram, a windows navigation diagram, or an interface navigation diagram.

User-interface model. A model comprising your user interface prototype, user-interface flow diagram, and any corresponding documentation regarding your user interface.

User-interface prototype. A prototype of the user interface (UI) of a system. User-interface prototypes could be as simple as a hand-drawn picture or as complex as a collection of programmed screens, pages, or reports.
Related topics

- The Process Patterns Resource Page by Scott Ambler
- "Enhancing the Unified Process" by Scott W. Ambler
- "Object-Oriented Training, Education, and Mentoring" by Scott W. Ambler
- Software Requirements by K. Wiegers (Redmond, WA: Microsoft Press)

© Copyright IBM Corporation 2000
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)