From

The Object Primer

2 edn.

Scott Ambler, 2001

Model twice, code once.

Chapter 7 Determining How to Build Your System: Object-Oriented Design

What You Will Learn in This Chapter

How to evolve your analysis model into a design model
How to layer the architecture of your system
How to develop a design class model
How to apply design patterns effectively
How to develop state chart diagrams
How to develop collaboration diagrams
How to develop a component-based design
How to develop a deployment model
How to develop a persistence model for your system
How to evolve your user interface prototype
How to take advantage of common object design tips

Why You Need to Read This Chapter

Your analysis model, although effective for identifying what will be built, doesn't contain sufficient information to define how your system will be built. Object-oriented design techniques—such as class modeling, state chart modeling, collaboration modeling, component modeling, deployment modeling, persistence modeling, and user interface prototyping—are used to bridge the gap between analysis and implementation.

The purpose of design is to determine how you are going to build your system and to obtain the information needed to drive the actual implementation of your system. This is different from analysis, which focuses on understanding what will be built.

Object-oriented design is highly iterative.

As you can see in Figure 7-1, your analysis artifacts, depicted as dashed boxes, drive the development of your design artifacts. As with the previous Figures 3-1 and 6-1, the arrows represent "drives" relationships; information in your analysis (conceptual) class model drives information in your design class model and vice versa. Figure 7-1 has three important implications: First, as with requirements and analysis, design is also an iterative process. Second, taken together, analysis and design are highly interrelated and iterative. As you see in Chapter 8, which describes object-oriented programming techniques, design and programming are similarly interrelated and iterative. Third, your analysis class model evolves into your design class model, as you see in this chapter, to reflect features of your implementation environment, design concepts such as layering, and the application of design patterns.

Are you taking a pure object approach or a component-based approach?

You must decide on several high-level issues at the beginning of design. 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 nonobject technology although that is not the topic of this book). Component modeling is the topic of Section 7.6.

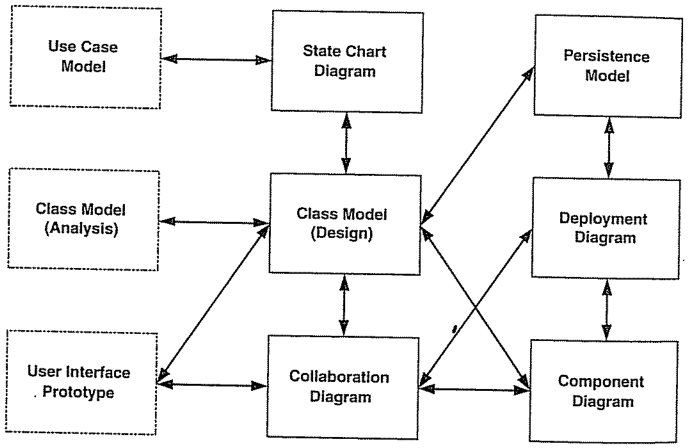

Figure 7-1. Overview of design artifacts and their relationships

DEFINITIONS

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

Class diagram. Show 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 divided into three sections: one indicating the name of the class the card represents, one listing the responsibilities of the class, and the third listing the names of the other classes with which this one collaborates 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.

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

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.

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

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

Use case diagram. A 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 (UI). The user interface of software is the portion the user directly interacts with, including the screens, reports, documentation, and software support (via telephone, electronic mail, and so on).

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 a collection of programmed screens, pages, or reports.

Will you follow a common business architecture?

A second major design decision is whether you plan to follow all or a portion of a common business architecture. This architecture may be defined by your organization-specific business/domain architecture model (Ambler, 1998b), 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 need to reflect this decision, showing how you plan to apply your common business architecture in the implementation of your business classes.

Will you follow a common technical architecture?

Third, you must decide whether you plan to 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, such as those suggested in Table 7-1? Enterprise JavaBeans (EJB), CORBA, and the San Francisco

Table 7-1. Common infrastructure services

Service	Description
Data Sharing	Encapsulates the management of common data formats, such as XML and EDI files.
File Management	Encapsulates and manages access to files.
Inter-Process Communication (IPC)	Implements middleware functionality, including support for messaging between nodes, queuing of services, and other applicable system communication services.
Persistence	Encapsulates and manages access to permanent storage devices. such as relational databases, object databases, and object-relational databases.
Printing	Implements the physical output of your system onto paper.
Security	Implements security access control functionality, such as determining who is entitled to work with certain objects or portions thereof, as well as encryption/decryption and authentication.
System Management	Implements system management features, such as audit logging, real-time monitoring (perhaps via SNMP), error management, and event management.
Transaction Management	Manages transactions, single units of work that either completely succeed or completely fail, across potentially disparate nodes within your system.

Component Framework (*www.ibm.com*) are examples of technical infrastructures on which you may decide to base your system. Perhaps one of the goals of your project is to produce reusable artifacts for future projects. If so, then you want to seriously consider technical architectural modeling. Although beyond the scope of this book, technical architectural modeling is a topic covered in *Process Patterns* (Ambler, 1998b), the third book of this series.

To what extent will you be able to support the nonfunctional requirements and constraints defined for your system?

Fourth, you need to decide which nonfunctional requirements and constraints, and, to what extent, your system will support. You refined these requirements during analysis (Chapter 6) and, hopefully, resolved any contradictions, but it is during design that you truly begin to take them into account in your models. These requirements typically pertain to many of the services described in Table 7-1. For example, it is common to have nonfunctional requirements describing security access rights, as well as data-sharing approaches. As you try to fulfill these requirements, you may find you are unable to implement them completely; perhaps it

DEFINITIONS

Common Object Request Broker Architecture (CORBA). An industry-standard, proven approach to distributed object computing, although in practice CORBA has also proven to be a significant force in the middleware arena. CORBA is defined and maintained by the Object Management Group (OMG).

Electronic Data Interchange (EDI). An industry-standard approach to sharing data between two or more systems.

Enterprise Java Beans (EJB). A component architecture, defined by Sun Microsystems, for the development and deployment of component-based distributed business applications.

Extensible Markup Language (XML). An industry-standard approach to data-sharing, an important enabling technology for EAI and e-commerce.

Framework. A reusable set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions.

Middleware. Technology that enables software deployed on disparate computer hardware systems to communicate with one another.

Node. A computer, switch, printer, or other hardware device.

Simple Network Management Protocol (SNMP). A standard protocol that specifies how to communicate status simply, often in near-real time, of system services. SNMP is used to monitor the status of the various software and hardware components of a system.

Transaction. A single unit of work that either completely succeeds or completely fails. A transaction may be one or more updates to an object, one or more reads, one or more deletes, or any combination thereof.

will be too expensive to build your system to support response times of less than a second, whereas a response time of several seconds proves to be affordable. The moral of the story is every system has design trade-offs.

7.1 Layering Your Models—Class Type Architecture

Layering your software increases its robustness.

Layering is the concept of organizing your software design into layers/collections of classes or components that fulfill a common purpose, such as implementing your user interface or the business logic of your system. A class-type architecture provides a strategy for layering the classes of your software to distribute the functionality of your software among classes. Furthermore, class-type architectures provide guidance as to what other types of classes a given type of class will interact with, and how that interaction will occur. This increases the extensibility, maintainability, and portability of the systems you create.

Good class-type architecture leads to systems that are extensible and portable.

What are the qualities that make up good layers? First, it seems reasonable that you should be able to make modifications to any given layer without affecting any other layers. This will help to make the system easy to extend and to maintain. Second, layers should be modularized. You should be able either to rewrite a layer or simply replace it and, as long as the interface remains the same, the rest of the system should not be affected. This will help to increase the portability of your software.

Figure 7-2 depicts a five-layer class-type architecture for the design of object-oriented software. As the name suggests, a user interface (UI) class implements a major UI element of your system. The business behavior of your system is implemented by two layers: business/domain classes and controller/process classes. Business/domain classes implement the concepts pertinent to your business domain such as "student" or "seminar," focusing on the data aspects of the business objects, plus behaviors specific to individual objects. Controller/process classes, on the other hand, implement business logic that involves collaborating with several business/domain classes or even other controller/process classes. Persistence classes encapsulate the capability to store, retrieve, and delete objects permanently without revealing details of the underlying storage technology. Finally, system classes provide operating-system-specific functionality for your applications, isolating your software from the operating system (OS) by wrapping OS-specific features, increasing the portability of your application.

Collaboration between classes is allowed within a layer. For example, UI classes can send messages to other UI classes and business/domain classes can send messages to other business/domain classes. Collaboration can also occur between classes in layers connected by arrows. As you see in Figure 7-2, user interface classes may send messages to business/

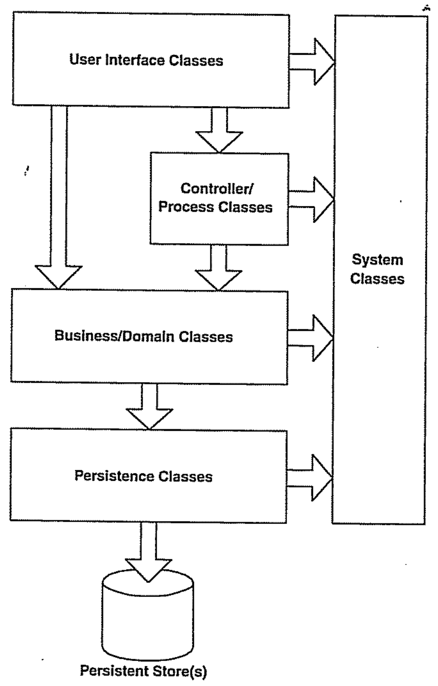

Figure 7-2. Layering your system based on class types

domain classes, but not to persistence classes. Business/domain classes may send messages to persistence classes, but not to user interface classes. By restricting the flow of messages to only one direction, you dramatically increase the portability of your system by reducing the coupling between classes. For example, the business/domain classes don't rely on the user interface of the system, implying that you can change the interface without affecting the underlying business logic.

Restricting message flow between layers increases portability by decreasing the coupling between classes.

All types of classes may interact with system classes. This is because your system layer implements fundamental software features such as inter-process communication (IPC), a service classes use to collaborate with classes on other computers, and audit logging, which classes use to record critical actions taken by the software. For example, if your user-interface classes are running on a personal computer (PC) and your business/domain classes are running on an Enterprise JavaBean (EJB) application server on another machine, then your UI classes will send messages to the business/domain classes via the IPC service in the system layer. This service is often implemented via the use of middleware.

7.1.1 The User-Interface Layer

Your system may need to support several user interfaces.

A user interface class contains the code for the user interface part of an application. For example, a graphical user interface (GUI) will be implemented as a collection of menu, editing screen, and report classes. Don't lose sight of the fact that not all applications have GUIs, however. For example, integrated voice response (IVR) systems using telephone technology are common, as are Internet-based approaches. Furthermore, by separating the user interface classes from the business/domain classes, you are now in a position to change the user interface in any way you choose. Consider the university where users currently interact with the system through an existing GUI application. It seems reasonable that people should also be able to interact with the system, perhaps find out information about seminars, or even enroll in seminars, over the phone or the Internet. To support these new access methods, you should only have to add the appropriate user interface classes. Although this is a dramatic change in the way the university interacts with its customers (students), the fundamental business has not changed, therefore, you shouldn't have to change your business/domain classes. The point to be made here is that the user interface for any given system can take on many possible forms, even though the underlying business is still the same. The only change is the way you interact with that business functionality.

User interface classes are often identified as part of your UI prototyping efforts, as well as sequence modeling. Referring back to Figure 6-8, you see the sequence diagram for the basic course of action for the "Enroll in Seminar" use case. Classes that belong to the user interface layer include the three boxes with the stereotype of "<<UI>>": the security logon class, the seminar selector class, and the fee display class.

User interface classes are often referred to as interface classes (Jacobson et al., 1992) or boundary classes (Jacobson, Booch, and Rumbaugh, 1999), so don't be surprised if you see stereotypes of "<<interface class>>" or "<<boundary class>>." As usual, pick one style and stick with it.

7.1.2 The Controller/Process Layer

The purpose of a controller/process class is to implement business logic that pertains to several objects, particularly objects that are instances of different classes. On Figure 7-3, controller classes are given the stereotype of "<<controller>>" although you may also see "<<controller class>>" applied as well (Jacobson, Booch, and Rumbaugh, 1999).

I reworked the analysis version of the sequence diagram, depicted in Figure 6-8, to reflect more closely how the system would actually be built,

DEFINITIONS

Audit logging. The recording of information to identify an action of interest to the system, when the action took place, and who/what took the action.

Business/domain class. Implements the concepts pertinent to your business domain, such as "customer" or "product." Business/domain classes are usually found during the analysis process. Although business/domain classes often focus on the data aspects of your business objects, they will also implement methods specific to the individual business concept.

Class-type architecture. A defined approach to layering the classes that comprise the software of a system. The interaction between classes is often restricted based on the layer to which they belong.

Controller/process class. Implements business logic that involves collaborating with several business/domain classes or even other controller/process classes.

Extensibility. A measure of how easy it is to add new features to, to extend, existing software. If item *A* is easier to change that item *B*, then we say that item *A* is more extensible than item *B*.

Inter-process communication (IPC). The act of having software running on two separate pieces of hardware interact with one another.

Layering. The organization of software collections (layers) of classes or components that fulfill a common purpose.

Maintainability. A measure of how easy it is to add, remove, or modify existing features of a system. The easier a system is to change, the more maintainable we say that system is.

Persistence class. Provides the capability to store objects permanently. By encapsulating the storage and retrieval of objects via persistence classes, you are able to use various storage technologies interchangeably without affecting your applications.

Portability. A measure of how easy it is to move an application to another environment (which may vary by the configuration of either their software and hardware). The easier it is to move an application to another environment, the more portable we say that application is.

System class. Provides operating-system-specific functionality for your applications or wraps functionality provided by other tool/application vendors. System classes isolate your software from the operating system (OS), making your application portable between environments, by wrapping OS-specific features.

User interface class. A class that provides the capability for users to interact with the system. User interface classes typically define a graphical user interface for an application, although other interface styles, such as voice command or HTML, are also implemented via user interface classes.

DEFINITIONS

Graphical user interface (GUI). A style of user interface composed of graphical components, such as windows and buttons.

Major user interface element. A large-grained item, such as a screen, HTML page, or report.

Sequence diagram. A UML diagram that models the sequential logic, in effect, the time ordering of messages between objects.

Stereotype. Denotes a common use of a modeling element. Stereotypes are used to extend UML in a consistent manner.

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.

Controller classes collaborate with other controller classes and business classes.

resulting in Figure 7-3. Part of the rework effort was to layer the application appropriately, including the refactoring (Fowler, 1999) of the controller class to interact only with business classes. This refactoring included the introduction of a new user interface class representing the main menu (or front page) of the application, the purpose of which is to manage the user's main interactions with the rest of the system's major user interface items. The second aspect of the refactoring is that the controller class, "EnrollInSeminar," now only manages interactions between business classes. Notice how this refactoring supports the message flow rules, indicated by Figure 7-2. For example, the only types of classes that interact with user interface classes are other user interface classes, whereas in the analysis version of the diagram, the controller class also interacted with UI classes. The problem with the approach in the analysis version is the controller classes, which should just implement business logic, has knowledge of the user interface, reducing its portability and reusability (you couldn't use this controller with a browser-based UI, for example). Now, taking a layered approach, the class is more robust. Interesting to note is that the controller class destroys itself at the end because it is no longer needed—it completes its job, and then removes itself from memory when it is finished.

Classes provide collections of their own instances.

Another interesting aspect of Figure 7-3 is the introduction of the "Seminar" class to support getting a list of available seminar objects via the static method "getAvailableSeminars()." You know it is a static method because the message is being sent to a class. Had it been sent to an object, then it would have been an instance method. Typically, the responsibility of a class is to implement basic searching responsibilities, such as providing a collection of all seminars or, in this case, all seminars that are available to be enrolled in (some seminars may not be offered this term).

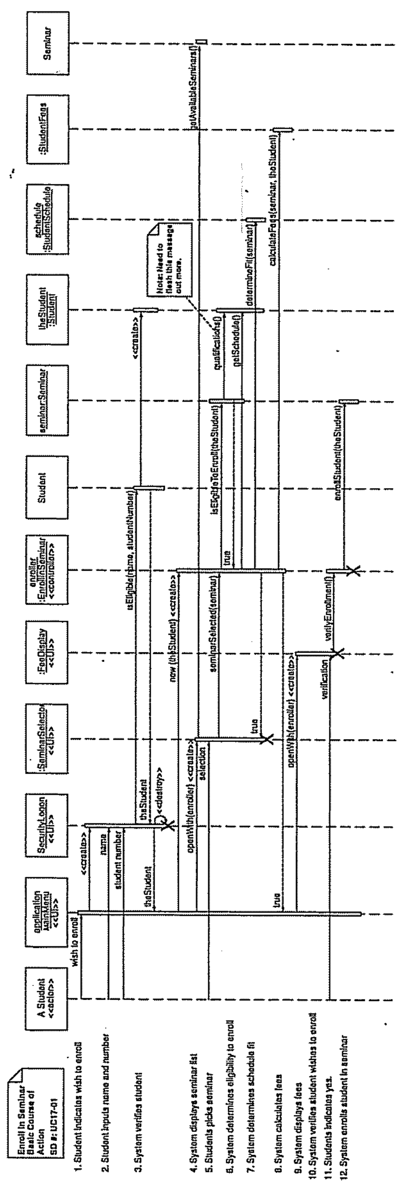

Figure 7-3.
A reworked sequence diagram respecting layering

See next page for larger version

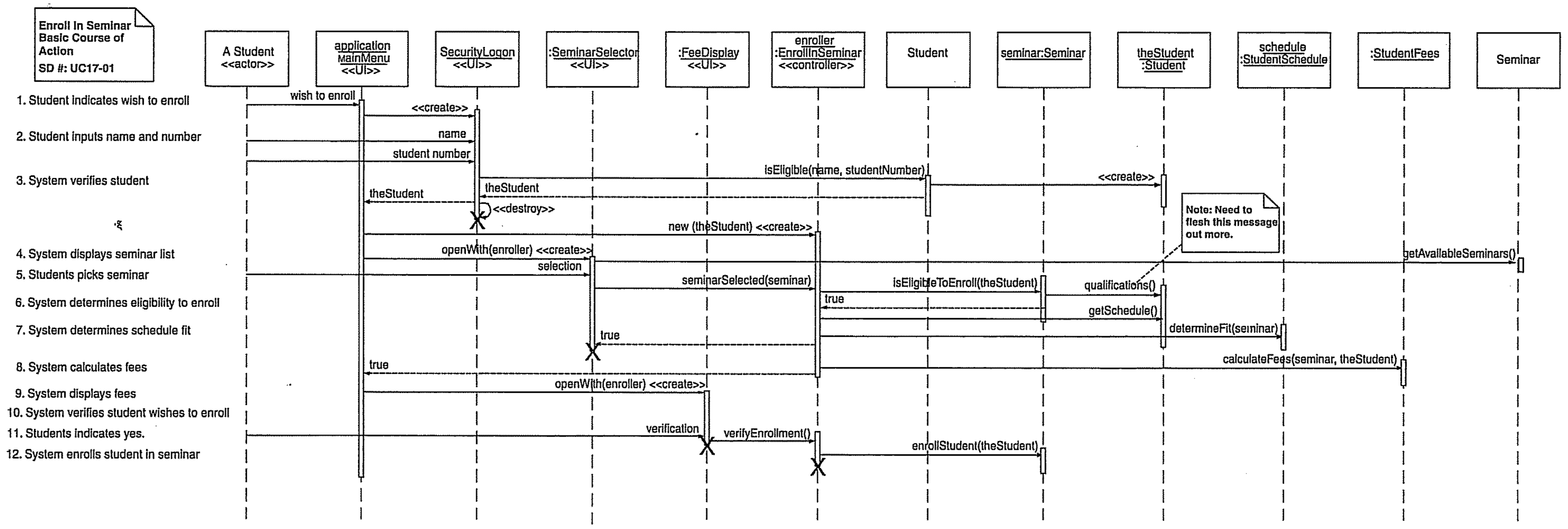
Enroll In Seminar Basic Course of Action
SD #: UC17-01
A Student <<actor>>
application MainMenu <<UI>>
SecurityLogon <<UI>>
:SeminarSelector <<UI>>
:FeeDisplay <<UI>>
enroller :EnrollInSeminar <<controller>>
Student
seminar:Seminar
theStudent :Student
schedule :StudentSchedule
:StudentFees
Seminar
1. Student indicates wish to enroll
2. Student inputs name and number
3. System verifies student
4. System displays seminar list
5. Students picks seminar
6. System determines eligibility to enroll
7. System determines schedule fit
8. System calculates fees
9. System displays fees
10. System verifies student wishes to enroll
11. Students indicates yes.
12. System enrolls student in seminar
wish to enroll
<<create>>
name
student number
isEligible(name, studentNumber)
<<create>>
theStudent
theStudent
<<destroy>>
new (theStudent) <<create>>
Note: Need to flesh this message out more.
openWith(enroller) <<create>>
getAvailableSeminars()
selection
seminarSelected(seminar)
isEligibleToEnroll(theStudent)
qualifications()
true
getSchedule()
determineFit(seminar)
true
true
calculateFees(seminar, theStudent)
openWith(enroller) <<create>>
verification
verifyEnrollment()
enrollStudent(theStudent)

Figure 7-3.
A reworked sequence diagram respecting layering

DEFINITIONS

Instance method. A method that operates on a single instance (object) of a class.

Static method. A method that operates at the class level, potentially on all instances of that class.

7.1.3 The Business/Domain Layer

A business/domain class, also called an analysis or entity class (Jacobson, Booch, and Rumbaugh, 1999), is a class that is usually identified during analysis. Your subject matter experts (SMEs) are often the people who identify these classes or, at least, the concrete business/domain classes. Referring back to Figure 6-16 we see a conceptual class model that contains business/domain classes pertinent to a university information system. The business layer enables you to encapsulate the basic business functionality without having to concern yourself with user interface, data management, or system management issues.

7.1.4 The Persistence Layer

Your business objects should not be affected by changes to your persistence strategy.

The persistence layer provides the infrastructure for the storage and retrieval of objects. This helps to isolate your application from changes to your permanent storage approach. You might decide to install the latest version of your database, change your existing database schema, migrate to a new database vendor, or even change your data storage approach completely (perhaps migrating from a relational database to an object database). Regardless of how your persistence strategy changes, your applications should not be affected. The persistence layer, by encapsulating data management functionality, increases the maintainability, extensibility, and portability of your applications.

The persistence layer encapsulates access to permanent storage, but it is not the storage mechanism itself.

In the layered class-type architecture of Figure 7-2, messages flow from the business/domain class layer to the persistence class layer. These messages take the form of "create a new object," "retrieve this object from the database," "update this object," or "delete this object." These types of messages are referred to as *object-oriented create, retrieve, update, and delete* (OOCRUD). Another essential concept here is that the persistence layer

DEFINITION

Subject matter expert (SME). A person who is responsible for providing pertinent information about the problem and/or technical domain either from personal knowledge or from research.

only provides access to permanent storage; it is not the permanent storage mechanism itself. For example, the persistence layer may encapsulate access to a relational database, but it is not the database itself. The goal of the persistence layer is to reduce the maintenance effort that is required whenever changes are made to your database.

The persistence layer isolates you from the impact of changes to your storage strategy.

Why do you need a persistence layer? You know your database will be upgraded. You know tables will be moved from one database to another, or from one server to another. You know your data schema will be changed. You know field names will be changed. The implication is clear: because the database is guaranteed to change, we need to encapsulate it to protect ourselves from the change. A persistence layer is the best way to do this because it minimizes the effort required to handle changes to permanent storage.

7.1.5 The System Layer

The system layer provides access to the operating system and non-OO resources.

Every operating system offers functionality that we want to be able to access in our applications—file handling, multitasking, multithreading, and network access to name a few. Most operating systems offer these features, albeit in slightly different manners. Although many people find this little fact to be worthy of great debate and, perhaps, it actually is, the real issue is that the differences between operating systems can make it tough if you are writing an application that needs to work on many different platforms. You want to wrap the features of an operating system in such a way that when you port an application, you only need to modify a minimum number of classes. In other words, you need to create classes that wrap specific features of the operating system. Even if you don't intend to port your applications to other operating systems, you still need to consider wrapping system functionality. The reason for this is

DEFINITIONS

Object database (ODB). A permanent storage mechanism, also known as an objectbase or an object-oriented database management system (OODBMS), that natively supports the persistence of objects.

OOCRUD. Object-oriented create, retrieve, update, and delete.

Permanent storage. Any physical medium to which data can be saved, retrieved, and deleted. Potential permanent storage mechanisms for objects include relational databases, files, and object databases.

Relational database (RDB). A permanent storage mechanism in which data is stored as rows in tables. RDBs don't natively support the persistence of objects, requiring the additional work on the part of developers and/or the use of a persistence layer.

simple: Operating systems constantly get upgraded. Every time an upgrade occurs, there are always changes to the way that functionality is currently being offered, including issues such as bug fixes and completely new ways to do things.

System classes encapsulate non-OO functionality by wrapping it with OO code.

The key concept here is wrapping (Ambler, 1998a). System classes for the most part encapsulate non-OO functionality that we need to make accessible to objects within an application. It is quite common to wrap a series of related operating system calls to provide a related set of functionality. A perfect example would be the file stream classes commonly found in Java and C++. When you look into the inner workings of these classes, you find their methods make specific file-handling calls to the operating system. These classes, particularly the Java ones, provide a common way to work with files, regardless of the platform.

Message flow for system classes is greatly restricted. System classes are only allowed to send messages to other system classes, even though each type of class is allowed to send messages to system classes. This is because system classes are the lowest common denominator in software development. This means they don't need to know anything about the business logic or user interface logic to do their job. Actually, it is not completely true that system classes don't interact with nonsystem classes. The use of callbacks, when one object passes itself as a message parameter to another so the receiver can later call it back, is permitted (it is allowed between all layers, but it is most common with system classes). For example, instead of waiting, a business class may request that a printing system class inform it when/if the print request was successful. When the printing is complete, a message would be sent from the system object to the business object informing it of success.

7.2 Class Modeling

Your design class model will reflect the wide variety of technology decisions you make.

The purpose of design is to model how the software will be built. As you would expect, the purpose of design-class modeling is to model the static structure of how your software will be built. The techniques of Chapter 7

DEFINITIONS

Callback. An approach where one object indicates it wants to be sent a message once its request has finished processing; in effect, it wants to be "called back."

Wrapper. A collection of one or more classes that encapsulates access to non-OO technology to make it appear as if it is OO.

Wrapping. The act of encapsulating non-OO functionality within a class, making it look and feel like any other object within the system.