Part I

Introduction to Building Object Applications

Chapters

Been there, done that.

Wait a minute, where'd this new stuff come from?

Chapter 1

Where We've Been Before — Object-Oriented Concepts and Techniques

What We'll Learn in This Chapter

Object-oriented concepts, including a few that we missed in The Object Primer.

Modeling techniques like class responsibility collaborator (CRC) modeling, use-case scenario testing, and class diagramming.

An iterative development life cycle called "The Pinball System Development Life Cycle (SDLC)."

What this book is all about.

Unless you're reading The Object Primer *and* Building Object Applications That Work *back to back you're going to need a quick refresher. Furthermore, there's some new material in this chapter starting in section 1.3 (hey, like we'd want you to skip an entire chapter). Until the release of this book,* The Object Primer *was the greatest thing since time-sliced bread. Now we have* Building Object Applications, *a book that is*

even more spectacular than its predecessor. In The Object Primer *we covered analysis techniques such as class responsibility collaborator (CRC) modeling and use-case scenario testing. We also introduced several object-oriented (OO) concepts and showed you how to model them using class diagrams. We finally put all this into the context of the Pinball System Development Life cycle (SDLC), an iterative and incremental approach to OO systems development.*

1.1 Object-Oriented Concepts — A Really Quick Recap

In *The Object Primer* we introduced you to several concepts that are critical to understanding the OO development process. Let's invest a few minutes reviewing the following object-oriented concepts:

- Objects
- Classes
- Attributes
- Methods
- Abstraction
- Encapsulation
- Information hiding
- Inheritance
- Persistence
- Instance relationships
- Aggregation
- Collaboration
- Coupling
- Cohesion
- Polymorphism

1.1.1 Objects and Classes

An *object* is any person, place, event, thing, screen, report, or concept that is applicable to the system. In a university system, Arthur Dent is a student object, he attends several seminar objects, and he is working on a degree object. In a banking system, Arthur is a customer object, and he has a checking account object from which he bounces rubber check objects. Consider an object-oriented inventory control system: every inventory item is an object, every delivery is an object, and every customer is an object. Things, events, and people are all objects in an OO inventory control system (actually everything is an object, I just like being specific sometimes).

An object is any person, place, thing, event, concept, screen, or report.

Objects are often similar to other kinds of objects. Students share similar characteristics (they do the same sort of things, they are described in the same sort of way), courses share similar characteristics, inventory items share similar characteristics, bank accounts share similar characteristics, and so on. Although we could model (and program) each and every object individually, that's a lot of work. We would prefer to define what it is to be a student once, define courses once, define inventory items once, define bank accounts once, and so on. That's why we need the concept of a *class* of objects.

A class is a blueprint from which objects are created.

When an OO program is running, objects are *instantiated* (created/defined) from classes. In other words a class is effectively a blueprint from which objects are created. We say that an object is an *instance* of a class, and that we *instantiate* objects from classes.

Objects are instances of a class.

DEFINITIONS

Object—Any person, place, thing, event, screen, report, or concept that is applicable to the design of the system. Objects have both data and functionality that define their behavior.

Class—A category of similar objects. A class is effectively a blueprint from which objects are created.

Instance—Another word for object. We say that an object is an *instance* of a class.

Instantiate—To create an instance. When we create an object we say that we *instantiate* it from a class.

1.1.2 Attributes and Methods

The basis of the object-oriented paradigm is the concept that systems should be built out of objects, and that objects have both data and functionality. *Attributes* define the data, whereas *methods* define the functionality.

DEFINITIONS

Attribute—Something that an object or class knows. An attribute is basically a single piece of data or information. Attributes can be simple, like a string or integer, or they can be a complex object, like an address or customer.

Method—Something that an object or class does. A method is similar to a function or procedure in structured programming.

Member function—This is the C++ term for method.

1.1.3 Abstraction, Encapsulation, and Information Hiding

When we determine what a class knows and does, we say that we abstracted the interface of the class. When we hide how the class will accomplish these things, we say that we *encapsulated* them. When we design the class well by restricting access to its attributes, we say that we've hidden their information.

The world is a complicated place. In order to deal with that complexity, we form generalizations, or *abstractions* of the things in it. For example, consider the abstraction of a person. A university needs to know a person's name, address, phone number, social security number, and educational background. A police department needs to know a person's name, address, phone number, weight, height, hair color, eye color, and so on. It's still the same person, just a different abstraction depending on the application at hand.

We abstract, or define, the interface of an object.

Although the act of abstraction tells us that we need to store a student's name and address, as well as be able to enroll students in seminars, it doesn't tell us how we are going to do this. *Encapsulation* deals with the issue of how do we intend to modularize the features of a system. In an object-oriented world we modularize systems into classes, which in turn are modularized into methods and attributes.

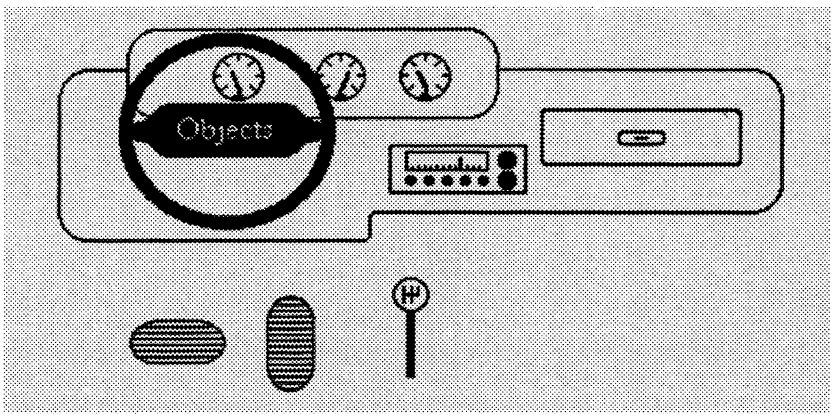

Figure 1.1. The driver's interface for a car.

We say that we *encapsulate* behavior into a class, or that we *encapsulate* functionality into a method.

We encapsulate, or build, behavior into a class.

In order to make our applications maintainable we want to restrict access to data attributes and some methods. The basic idea is this—if one class wants information or services from another class, it should have to ask for it instead of take it. When you think about it, this is exactly the way that the real world works. If you want to find out somebody's name what would you do? Would you ask them their name or would you steal their wallet and look at their ID? By restricting access to attributes, which we call *information hiding,* we prevent programmers from writing highly coupled code. When code is highly coupled, a change in one part of the code forces us to make a change in another, and then another, and so on. We'll discuss coupling in detail later in this chapter.

Information hiding is the restriction of access to attributes.

In Figure 1.1, the abstraction is how we work with the wheel, pedals, and gear shift to drive a car. Encapsulation allows the different carmakers to provide a consistent interface, even though each make of car is built differently. Information hiding is represented by the fact that although there is a certain amount of oil in the engine, the driver doesn't know exactly what it is (well, at least I don't). In other words, information about the oil is hidden from the user.

DEFINITIONS

Interface—The set of messages an object or class will respond to.

continued

DEFINITIONS

continued

Abstraction—The definition of the interface of a class (what it knows and does).

Encapsulation—The hiding of the implementation of what a class/object knows or does, without telling anyone how it's done.

Information hiding—The restriction of access to attributes.

1.1.4 Inheritance

There are often similarities between different classes. Very often two or more classes will share the same attributes and/or the same methods. Because we don't want to have to write the same code over and over and over and over and over we need a mechanism that takes advantages of these similarities. Inheritance is that mechanism. *Inheritance* models "is-a" and "is-like" relationships, allowing us to reuse existing data and code.

Inheritance models "is-a" and "is-like" relationships.

There are often similarities between classes. For example, students have names, addresses, and phone numbers, and they drive vehicles. At the same time, professors also have names, addresses, and phone numbers, and they drive vehicles. Without a doubt, we could develop the classes for student and professor, and get them both running. In fact, we could even develop the class **Student** first, and once it is running make a copy of it, call it **Professor**, and make the necessary modifications. Although this is fairly straightforward to do, it isn't perfect. What if there was an error in the original code for **Student**? Now we'd have to fix it in two places which is twice the work. What would happen if we needed to change the way we handled names (say we go from a length of 30 to a length of 40)? Now we'd have to make the same change in two places again. That's a lot of dull, boring, tedious work.

Wouldn't it be nice if somehow we could only have one copy of the code to develop and maintain? That's what inheritance is all about. Inheritance allows us to take advantage of similarities between classes and develop the code for them only once. In fact, you could say that inheritance goes hand in hand with abstraction: you abstract out the similarities and use inheritance to take advantage of them.

DEFINITIONS

Inheritance—This allows us to take advantage of similarities between classes by representing "is-a" and "is-like" relationships.

Superclass—If class "B" inherits from class "A," then we say that "A" is a superclass of "B."

Subclass—If class "B" inherits from class "A," then we say that "B" is a subclass of "A."

Single inheritance—When a class directly inherits from only one class, we say that we have single inheritance.

Multiple inheritance—When a class directly inherits from more than one class, we say that we have multiple inheritance. Note that not all OO languages support multiple inheritance.

Concrete class—A class that has objects instantiated (created) from it.

Abstract class—A class that does not have objects instantiated from it, but will provide functionality inherited by its subclasses.

Override—A term used to indicate that we redefine attributes and/or methods in subclasses to provide slightly or completely different behavior.

Overload—See override.

Class hierarchy—A set of classes that are related through inheritance.

Root—The topmost class in a class hierarchy.

1.1.5 Persistence

Persistence describes the issue of how to save objects to permanent storage. To make an object persistent you must save the values of its attributes to storage (onto disk), as well as any information needed to maintain the relationships (both aggregation and instance relationships) that it is involved with. Persistence allows objects to exist between executions of the system. A *persistent object* should be able to be recreated exactly at a future time.

Persistent objects are saved to permanent storage and transitory objects aren't.

From a development point of view, there are two types of objects: persistent objects that stick around and transitory objects that don't. For example, a customer is a persistent object. You want to save customer objects into permanent storage so that you can work with them again in the future. A customer editing screen, however, is a *transitory* object. Your application creates the customer editing screen object, displays it, then gets rid of it once the user is done editing the data for the customer with whom they are currently working. A common use for transitory objects is to manipulate data for a short period of time and then be discarded. In chapter 10 we will cover persistence in detail.

DEFINITIONS

Persistence—The issue of how objects are permanently stored.

Persistent object—An object that is saved to permanent storage, making it retrievable for future use.

Transitory object—An object that is not persistent.

Transient object—See transitory object.

Persistent memory—Main memory plus all available storage space on the network.

1.1.6 Instance Relationships

Objects are associated with, or are related to, one another. For example, students TAKE courses, professors TEACH courses, criminals ROB banks, politicians KISS babies, and captains COMMAND starships. Take, teach, rob, kiss, and command are all verbs that define relationships between objects. We want to identify and document these relationships so that we can gain a better understanding as to how objects interact with one another. An *instance relationship* is the same concept as a relationship in an entity-relationship (ER) diagram or a data model.

Not only must we identify what the relationship(s) are between objects, we must also describe the relationship. For example, it isn't