Chapter 4 Encapsulation: Classes and Methods

4.1 Designing a Robot

Interface and Implementation: The interface of a well-designed object describes the services that the client wants accomplished. The good class designer begins by thinking about a class from the point of view of the client. The implementation of an object is the manner in which it performs its job.

Encapsulation hides the nonessential. Rather than `forward()` and `backward()` methods, a truly useful robot would have methods with names like `vacuumHouse()`, `doLaundry()`, etc. A really useful robot would let you tell it what you wanted done and then have only a single method, `run()`!

4.2 Abstraction and Encapsulation: The First Iteration

The purpose of a well-designed class is to model an abstraction from the client's point of view. This means you should always start by designing a minimal public interface, because the public interface is the only thing that the client will see.

Designing the public interface: Some questions can be answered to better design an interface.

- **Who:** What objects are going to use the services of your class? Such objects, called *client objects*, which send messages to your objects and receive information in return.
- **Where (Environment):** What is the environment in which your design will be implemented? The environment imposes limits on how your interface is actually designed.
- **What:** What should the object do? This functionality is described from the user's point of view. Many—sometimes most—of the methods you define will be necessary, yet will not appear in the public interface.

4.3 Client Objects, Environment and Functionality

For the majority of programs that are written, discovering the right requirements is often more difficult than writing the code, and is more error-prone.

A good first step in class design is to describe, in a single paragraph, exactly what the class you are building should do.

[...] Once identified the cliend objects, the task is to create an interface that conforms to their needs.

4.4 Implementing the Interface

Template: Begin with a template to remember necessary but logically non-essential details so you can concentrate on the essential. A template helps you to logically organize your class so the important parts stand out.

STUBS: Write *stubs* before you write method bodies. Stubs allow you to test your interface!

A stub method may contain is an outline of the steps necessary to perform that particular operation.

```java
public int totalArea(List squareList) {
    // STUB

    // 1. Calculate the area of each square
    // 2. Sum all areas
    // 3. Return area

    return 0;
}
```

Refining the Interface: Designing a class is really an iterative process: you design a little, code a little, test a little, and then go back and design a little more.

One useful way to test the interface (once a minimal is build- with stubs), is to write programs that use the object and see if it does everything needed.

Paradoxically, the real advantage of writing code that uses your class is that it gets you out of the "implementation" mind-set. The danger is that you'll fall victim to "featuritis". Your design goal for the public interface is that it should be minimal and complete.

4.5 Implementing State

Attributes: The attributes of your objects store its state or characteristics.

Hide Your Data: All object and class attributes should be declared as **private**, without exception

One of the major advantages, is that in decouples the interface from the implementation: The class implementation can change, and if interface is left the same, the class is perceived as the same.

To access such data, we can write *getters* and *setters*, but you shouldn't generally write them. They uncover implementation.

Three Class Design Errors:

- **Data Warehouse Trap**: An object is not a repository of data. Objects should contain both data and the methods that work on that data.
- **Spectral Object Trap**: An object is not a collection of medthods. (Otherwise ask "why is the data I'm workin on stored in a different place than the operations that work on such data?".
- **Multiple Personality Trap**: An object should model one abstraction. Called principle of *cohesion*. Usually, a multiple personality error is obvious from the name of the class.

Class Constants: Store a single value that is shared amongs all objects and users outside of the class.

Tip: One way to use them is to have propper names for options: People are much better with association than just remembering that ”2 stands for Sunken-Border.” And that is true for developers and users of the class. (Question any number appearing in source code other than 0, 1 or -1.)

4.6 Writing Methods

Cohesion is also important when evaluating methods: Does each method represent, or model, a single operation? That is, are the methods themselves internally cohesive?

Tip: If programs that use your objects rely primarily on `get()` and `set()` methods, this may be a sign that your interface is not complete, that client objects are having to process your object's data.

Tip: Methods should tell an object to perform some meaningful behavior, not act as a way to read and write `private` data.

Constructors: Constructors must initialize an object in a *valid state*, that is with every field initialized and all class invariants satisfied.

Even default constructors, initialize the object with default valid field values. If no reasonable valid value exists for all members, one option is to prohibit the default constructor.

If the values passed to a constructor don't permit the construction of a valid object, the constructor should throw an exception rather than construct an invalid object.

The *working constructor*, is the constructor that requires you to specify all the user-selectable inputs. It should be written the first, and write all overloaded constructors in terms of it (If you write completely separate constructors, you run the risk that, due to a programming error, an object constructed via constructor A will exhibit subtly different behavior from one constructed by constructor B.)

```java
public Fraction(int num, int den) {
    this.numerator = num;
    this.denominator = den;
}

public Fraction(int num) {
    this(num, 1);  // Overloaded valid state
}
```

Tip: You have to be aware of how your constructors interact with superclass constructors (i.e. using `super` constructor in Java).

Mutators: Methods that change fields of an object are called *mutator* methods. These mutator methods must preserve invariants of the class.

Accessors: Methods that acces to information of the class and return some meaningful data.

Tip: When your accessor method returns a reference to an object, the caller may be able to modify your object. We can avoid this by returning *immutable* objects or a *copy* of the object. Or better, don't return access to individual fields at all.

Private Methods: Classes are not function libraries. If you find yourself creating unneeded objects just to use their methods, you've fallen into the ”spectral object trap” and you need to make those methods private.

Chapter 5 Designing Classes and Objects

5.1 The Renter Applet

Class Design: A good design hides much of the complexity of a class in the implementation of the class. Ideally, the interface of a class should be small and simple.

There is an exception to the rule: when dealing with very simple classes often there will be more interface than implementation.

5.2 A System Design Process

One approach can be applied to a variety of problems when designing a new program (or system).

1. Determine the requirements of the program.
2. Identify te classes and objects.
3. Describe the object collaborations and the classes.
4. Sketch the user interface.

The design process is exploratory and iterative. Each step may take the designer forward to the following step or back to a preceding step. If it seems to the designer as though a tentative design is going nowhere, the designer may abandon it and restart the entire process.

5.3 Determining the Requirements

The first step of object-oriented design is to determine the general requirements the program must satisfy.

Tip: Don't try to perfeclty design the system in your head. The trick is to get something down on the paper and then study it and improve it (The simple act of writing will help form ideas).

Summary Paragraph: A good way to begin to determine the program requirements is to write a *summary paragraph*. The summary paragraph should view the system from a functional perspective, focusing on the inputs and outputs of the system and on its external. In effect, it should describe the interface of the program rather than the implementation of the program.

This paragraph must be detailed: specify all but necessary actions.

5.4 Identifying Classes and Objects

The next step in the object-oriented design is to identify al the classes and objects that will comprise the program.

CRC Cards: An effective way to identify classes and objects is by preparing what are known as CRC (*class-responsiblity-collaboration*) cards. Each CRC card describes a single class in terms of the data attributes, responsibilities, and collaborations of the class. The responsibilities of a class are simply the messages it responds to (that is, the public methods it contains). The collaborations (or, collaborators, if you prefer) of a class are simply other classes that interact with the class by sending it messages or receiving messages from it.

Tip: CRC cards are prepared from a problem summary statement rather than from a completed design.

Some suggested ways of finding classes, responsibilities and collaborations from the summary paragraph:

- **Classes:** Read through the problem summary statement and identify nouns and noun phrases, placing them on a list. Often it's necessary that the design include classes that are merely implicit in the problem summary statement.
- **Responsibilities:** Responsibilities relate to actions. A good starting place for finding responsibilities is in the verbs of the problem summary statement
- **Collaborations:** After you've identified the classes and their responsibilities, finding the collaborations is easy. Simply scan the list of responsibilities of each class and identify any other classes it needs in order to fulfill its responsibilities. List these as its collaborations.

[...] Design involves a myriad of such decisions, wherein each alternative gains you something, but only at a price. Knowing which to choose is no simple matter. That's why good designs are usually the result of an iterative process. You make a decision that you later see was more costly than you'd originally thought. A conscientious designer will go back and try the other alternative.

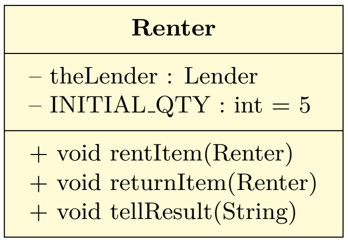
5.5 Descibint the Object Collaboration and the Classes

Tip: You may also discover that a class for which you prepared a CRC card is not as important as it initially seemed. For any such class, you allocate its responsibilities and attributes to other classes and discard its CRC card.

Use-Case Scenarios: Just as CRC cards helped you discover classes, *use-case scenarios* help you discover and describe collaborations. A use-case is simply a transaction or a sequence of related operations that the system performs in response to a user request or event.

Walk through each use-case (transaction). During the walk-through, you try to identify the objects involved in the use-case and the messages they exchange.

Class Diagrams: Though CRC cards are helpful during design, they're not very convenient for later reference. After you've completed the collaboration diagrams, you transfer information from each CRC card onto a *class diagram*.



Chapter 6 Round-Trip Design: A Case Study

6.1 Day 1: The System Concept

The best way to begin the system design is to develop a problem summary statement (summary paragraph). Offer any suggestions at all of what features and functions the system should have.

Tip: Begin a brainstorming session, listing functionalities as separated number items.

Knowing from experience that too much critical analysis can stifle creative problem solving, add as many suggestions to the list, even though you're convinced they're irrelevant or worse.

If no additional ideas have surfaced, review and discuss the functions identified so far. This, may lead to discover additional functions or help understand why one or more of the functions already on the list should be removed.

6.2 Day 2: Preparing CRC Cards

Follow up by making the classes CRC cards, which can be produced from the summary paragraph to identify different classes of the system, ther responsibilities and collaborations.

Tip: To extract the classes of the system from the summary underline the noun phrases in the problem statement. For each noun phrase, prepare a CRC card. These cards represent potential classes. The list of potential classes is then either expanded or contracted through group discussion. After a set of classes is agreed on, use the back of each card to record the attributes of each class. Don't be too quick to eliminate classes and attributes, because removing them later is simple; after all, they're not cast in code yet. Nevertheless, you want to avoid adding classes that are tied to the implementation or technical issues.

6.4 Day 4: Drawing Collaboration Diagrams

Exploring Use Cases: When we first wrote up the system definition, you described what you thought the system should do. Now, I'd like you to expand on that idea. We want to divide the system functions into specific operations and then elaborate on each of them. We don't care about the internal operation of the system just now. What we're interested in is <u>how a user would make use of the system</u>. We call these situations use cases, or scenarios. You might like to think of them as little dramas.

Tip: The easiest way to do that is by working through use-case scenarios. A scenario is developed by asking, "What happens when the user does this?" Remember, you're not interested in determining what actually happens at the technical level—that is, which records are deleted or which database tables affected. Instead, you're interested in describing system interactions from the user's point of view.

Chapter 7 Object Relationships: Implementing Associations

Relationships play a central role in software design. Describing, defining, and working on the relationships between classes and objects is one of the chief occupations of the object-oriented designer. Here we study *association relationships*: relationships between objects in which the objects exchange messages.

7.1 The Basics of Relationships

[Object Oriented] programs are communities of objects that collaborate to accomplish a goal. For that collaboration to be effective, these communities of classes must be organized. Like human communities as well, not just any organization will do.

One way to decide whether a particular organizational strategy is likely to be successful is to rely on simple rules, usually formed from past experience. Such rules are called *heuristics*. These help you design classes and objects that are robust, easy to use, and resistant to the unintended side effects of inevitable change. Two kind of rules exist for designing relationships:

i) Rules that help you decide which kind of relationship will be most effective.

ii) Rules that help you implement a particular relationship most effectively.

Let's first see what's available.

Types of Relationships: There are two basic types of realtionships: *class relationships* and *object relationships*. Class realtionships always involve inheritance. Each individual object always has this fixed relationship with the other related class (see next chapter). Object relationships, called *associations* are the relationships objects use to work together, and not all objects of the class are required to obey the relationship.

7.2 Kinds of Associations

Object realtionships (or associatinos) are more flexible than class realtionships and come in a wider variedy. Objects can work together in three different ways:

- An object ca se another object to do some work. Also called an *association*, or *acquaintance*, or *uses* relationship.

- A complex object may be composed of several simpler parts. Called a *composition* relationship.

- When two object depend on each other but don't directly interact, a *simple* or *weak association* is formed.

Tip: In reality, there aren't just three forms of association. These somewhat simplistic categories make it easier for you to analyze the collaborations between objects, but don't be surprised if you have difficulty "fitting" a particular collaboration into one of these.

Coupling: The underlying principle behind these categories is found in the idea of *coupling*, which measures the mutual dependence of two objects on each other. In general, you want to create classes that are as loosely coupled as possible while still being able to efficiently carry out their responsibilities. This goal is laudable; however, most of the time the nature of the problem itself will dictate the type of association you must use.

7.3 How Many? A Cardinal Question

When you design associations between objects, regardless of the type of association, you should ask yourself these questions:

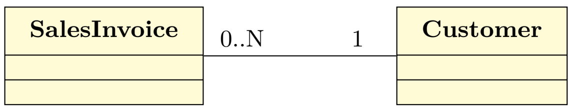
- Which objects collaborate with which other objects?

- How many objects will participate in the association?

- Is the association *mandatory* or *optional*?

Tip: Draw some simple collaboration diagrams to see the collaborating classes and all the different collaborating objects.

Cardinality: *Cardinality* refers to the number of objects that participate in an association and whether their participation is optional or mandatory. Every relationship has two sides. It i normally represented in a collaboration diagram.



Every relationship needs some rules, even if they're informal and unspoken. E.g. Every `SalesInvoice` object collaborates with only one `Customer` object. You may wish you could bill the same invoice to more than one `Customer`, but that would be illegal as well as unethical. Also, every `SalesInvoice` object must have a `Customer`—you can't send out a bill without one.

The first rule in an association relationship is the membership criteria (who gets to join?). The cardinality constraints lay out the answers to this fundamental question.

To implement an association, you not only have to know about the cardinality of the association, you also have to know how messages will be passed.

7.4 Objects That Use Objects

Whenever one object sends a message to another object, a uses relationship is established between the object that sends the message and the object that receives it. Does an object send messages to or receive messages from another object? If so, a uses relationship exists between those two objects.

One defining characteristic of the uses relationship is that it doesn't require the same level of commitment as other relationships. The uses relationship generally implies a lower level of coupling between objects than inheritance or containment relationships do.

Implementation: When trying to implement a *uses* relationship between two objects, then a queston arises: how does the first object (the *sender*) know the name of the second object (the *receiver*)? [Read *Object-Oriented Design Heuristics*, Arthur J. Riel for more on the below treatment.]

Consider the relationship between a car and a gas station. It doesn't make sense to say that a car contains a gas station; however, cars do ask gas stations to give them gasoline. How does the car know the name of the gas station? There are five implementations of the uses relationship aside from using a containment relationship (aside from uses via *containment*).

1. The car is given the name of the gas station as a formal parameter of the message.

```java
// In Car class ...
public void getGasoline(GasStation theStation,
                        int quantity) {
    theStation.giveMeGasoline(quantity);
    // Do more stuff ...
}
```

2. The car asks a third-party class (for example a map or a gps system for the name of an appropriate gas station.

```java
// In Car class ...
public void getGasoline(int quantity) {
    GasStation theStation;
    theStation = this.GPS.findGasStation();
    theStation.giveMeGasoline(quantity);
    // Do more stuff ...
}
```

   The problem here is delayed, where do we get the name from the third-party class? (In the example above, it was from a composition realtionship.)

3. A third possibility is that all cars in the world go to one global gas station and we all know its name by convention. This is actually a special case of the first method, since global data are considered implicit parameters to a method. (In the example it is done through a class constant.)

```java
// In Car class ...
private static final GasStation theStation;

public void getGasoline(int quantity) {
    theStation.giveMeGasoline(quantity);
    // Do more stuff ...
}
```

4. This method is for the wealthy. Whenever our car needs gasoline, we pull over onto the side of the road, buy the land that's there, build a gas station, use the gas station, and destroy the gas station when we leave. In short, the `getGasoline()` method for the car class builds a gas station as a local object, uses it, and the destroys it on exiting the method.

```java
// In Car class ...
public void getGasoline(int quantity) {
    GasStation theStation = new GasStation();
    theStation.giveMeGasoline(quantity);
    // Do more stuff ...
}
```

   While this is not appropriate for the car/gas station domain, there are many domains where building a local object to perform some functionality is useful.

5. last method for implementing the uses relationship is that "God" tells a car, when it is built, who its gas station is. The car stores this information in a special type of attribute called a *referential attribute* for later use in the `getGasoline()` method.

```java
// In Car class ...
public Car(GasStation theStation, ...) {
    this.myStation = theStation;
    // Other constructions ...
}

public void getGasoline(int quantity) {
    this.myStation.giveMeGasoline(quantity);
    // Do more stuff ...
}
```

   A referential attribute is an object field that refers to another object, even though that object isn't logically contained in the class that holds the field. Although this solution solves both the mandatory and validity constraints, it lacks a little something in practicality. Users of the class having the reference mustget hold of a fully constructed object before they can start.

7.5 Rules for Using Objects

Each Technique for implementing uses relationships has advantages and drawbacks. You should use a referential attribute when:

- The object needs to be directed from several different methods, or the object stores persistent state information between method calls.
- The object you are going to use is used repeatedly.
- The object you are going to use is very expensive or time consuming to construct, and you will use the object more than once.

You should pass the object you'll use as an argument when:

- The object you want to use will be used only in a single method.
- It's easier to construct the object you want to use outside your class. This is the case when the object you're going to use brings in some information supplied by the caller.

You should construct the object you want to use on-the-fly—that is, inside the method where it's used—when:

- The object will be used in only that method.
- The invoking object has information needed to construct the object that will be used, information that would be more difficult or impossible for an outside caller to supply.

7.6 Optional Associaions

The major difference between the `SalesInvoice-Customer` association and the `SalesInvoice-Recipient` association is that the `SalesInvoice-Recipient` association is *optional* rather than mandatory. You could, if you wanted, make this relationship mandatory, but making it optional allows you to look at how you can implement such optional associations.

An optional `Recipient` association means that the `shipTo` field contained in the `SalesInvoice` class can refer to either a valid `Customer` object or a `null` reference.

This means that we would have to check for special `null` cases everywhere we have an optional association. This is the major difficulty you'll encounter when you want to implement optional relationships: they make the code more complex. *Tip*: However, don't fake mandatory associations just for simplicity's sake.

7.7 Weak Associations

A weak association exists whenever you have an association where no messages are exchanged, but the association is necessary to the success of the overall abstraction you're trying to model.

The real danger you face when confronted with a potential weak association is determining whether it's really necessary for the abstraction you are working with.

Chapter 8 Object Relationships: Compositions and Collections

Your PC is composed of self-contained parts that interact by using well-defined interfaces; these parts are easily tested and replaced without affecting the rest of your system. Software too can be composed of parts that are tightly "wired" together, or it can be built in layers (subsystems), with each subsystem acting independently from the others, and with more complex systems made from differing arrangements of less complex and more fundamental parts.

8.1 Composition: The *part-of* Relationship

Sometimes, the best way to describe the relationship between two objects is to say that one object contains another object. It seems intuitive and natural to say that a car contains an engine, a university contains colleges, or a house contains a kitchen.

Compound Objects: Whenever a particular object is composed of other objects, and those objects are included as object fields, the new object is known as a *compound* object. Individual objects can be combined to create a more complex and useful object. And just as a uses association implies greater coupling between its members than a weak association, so too the part-of relationship implies greater mutual dependence than the simple uses association.

Complex Systems: Complex systems can be seen to include these four observations:

- Stable complex systems usually take the form of a *hierarchy*, where each system is built of simpler subsystems, and each subsystem is built from simpler subsystems still.
- Stable complex systems are *nearly decomposable*. That means you can identify the parts that make up the system and can tell the difference between interactions between the parts and interactions inside the parts. Stable systems have fewer links between their parts than they do inside the parts.
- Stable complex systems are almost always composed of only a few different kinds of subsystems, arranged in different combinations.
- Stable complex systems that work have almost always evolved from simple systems that worked. Rather than build a new system from scratch—reinventing the wheel—the new system builds on the proven designs that went before it.

Using composition is one of the key techniques you can use in your battle against complexity.

Composition—building relatively independent subsystems using simple parts—is the technique you'll most often use to apply the above insights to your own software designs.

Implementation of Composition Relationships: You may wonder at this point what's the difference between this new *part-of* relationship and a simple *uses* relationship. After all, as you saw in Chapter 7, the main way in which both weak associations and uses relationships are implemented is by embedding a reference to the associate as a field inside the class definition. Does this mean that those relationships are also composition relationships? Not at all.

The distinction between the three forms of association is a *semantic* distinction, not an *implementation* distinction. In Java, it's likely that all three forms—weak association, uses relationship, and part-of relationship—will be represented in exactly the same way when the code is written.

Composition vs. Association: Despite what the heading of this section might seem to imply, composition relationships are also a form of association. Specifically, composition is a strong form of the uses relationship. For composition to exist, either the parts or the whole must send messages to each other. If an object contains other objects and doesn't send messages to those objects, such a relationship is a weak association, not a containment relationship. Thus, it's something of a misnomer to speak of composition as entirely distinct from association.

Unlike a regular uses relationship, however, a composition relationship implies that one class (the whole) is made up of other objects (the parts). Composition is thus a whole-part relationship, where:

- One object is physically built from other objects.
- An object represents a logical whole, composed of smaller parts.
- Each part belongs to only a single whole.

  Tip: This is a useful test to use to differentiate between composition and *uses* relationships. (For example, in the `SalesInvoice` application, a single `Customer` may have placed several orders, and thus be referenced on several `SalesInvoices`. Therefore, the `SalesInvoice` doesn't contain a `Customer` object.)
- Each part lives and dies with the whole.

  For example, In the SalesInvoice application, you certainly wouldn't want your Customers disappearing every time you deleted a SalesInvoice object. Therefore, this relationship is a uses relationship, not a composition relationship. You would, however, want the LineItems to disappear with the rest of the SalesInvoice when you deleted it. This second relationship is a composition relationship.

Example: A Labeled TextField : Consider the example using the AWT Java library. The library provides a `Label` and `TextField` classes. However, not a Labeled Text Field class built-in class that combines both. By using composition, it's easy to build an aggregate object that meets the needs.

```
public class LTextField extends Panel {
    // Constructor
    public LTextField(String label,
                      int textSize);

    // Public Interface
    public String getText();
    public void setLabel();

    // Object Fields
    private FixedWidthLabel theLabel:
    private TextField       theText;
}
```

The most important characteristic of the `LTextField` class is that its parts—the `FixedWidthLabel` and `TextField`—are encapsulated as part of its state. Users of the `LTextField` class have no more access to these internal parts of the `LTextField`. Instead, users are presented with a simplified interface. *Tip*: So we should never return the aggregate objects themselves and let the user use the objects. That would break all the encapsulation! By hiding non-essential details inside a class, and then replacing those details with a simpler interface, composition allows you to concentrate on the task at hand without getting bogged down in minutia, much the way procedures allowed an earlier generation to see their programs at a higher level of abstraction.

8.2 Rules of Thumb for Composition

Iteration—trying out a design, noting its deficiencies, and then building a better model—is an essential component of object-oriented modeling. However, unless you have some way of telling whether you're actually making progress, you may end up in an endless loop. You need some way of telling whether you're making net progress, and some idea of which direction to go. To build robust composite objects, you first have to decide which parts go with what whole.

This is the main problem in composition: deciding what to put together and what to separate.

Localize Message Traffic. The first place to start looking for composite objects is to find out what objects already "hang out" together. Objects with high-frequency communication between them generally belong in the same subsystem. Objects with less communication belong in separate subsystems or modules.

The closer you can get to a collection of independent modules working in concert, the more stable and maintainable your system will be. But not completely independent, otherwise you wouldn't have a system, but merely a collection of autonomous agents.

Tip: Begin looking for communication links by identifying the frequency of message traffic between your objects. If you've already identified the associations in your system with the related messages, you've already done this.

Arthur Riel suggests that if a class contains objects of another class, the container should be sending messages to the contained objects. Furthermore, most of the methods defined in the class should be using most of the data members most of the time. These rules will help you determine which objects should be eliminated as candidates for composite classes.

Avoid Mixing Domains. In software development, a subsystem's domain is the area of functionality for which the subsystem is responsible.

Layering (breaking apart an application based on common functionality) is a time-tested method of grouping similar code together. As in the OSI model for computer networks, each layer provides a general service for networking applications.

The clear distinction between each domain makees it possible to write applications that work together, because each layer or part refrains from interfering with another,

Tip: One natural form of layering, which you should definitely consider whenever you build software, is to separate the user-interface, data storage, and business logic portions of your programs into different layers. Most computer applications require you to process some information, store the information, and present the information to users. By keeping the data-storage logic out of your user-interface classes, you make it much less likely that changes to your input screens will affect the integrity of your data, or that switching database vendors will require you to rewrite your user interface.

For systems of any size, you'll also want to consider partitioning the business-logic portion of your program along functional lines. By breaking each of these functional parts into its own subsystem, and then defining a set of clearly defined interfaces to facilitate communication between them.

Isolate The Changeable. A third criterion for composition is to group objects by their *stability*, their tendency to change. If you can, identify those portions of your system likely to change, and put those parts together. Then, combine the relatively unchangeable parts into separate subsystems. That way, when the inevitable change comes, rather than make changes scattered throughout 10 subsystems, you can concentrate on only one or two.

Create Simple Interfaces. Whereas composite objects should have high-frequency internal communication (that is, most of the methods should use most of the objects most of the time), communication between the "outside" world and the object should have lower frequency. This corresponds to the procedural idea of loose coupling between functions. One way to facilitate this is to create simple interfaces.

One consequence of using composition in this way is that users of `LTextField` objects can't do all the things that they might do with the component parts. You may instinctively recoil at these restrictions and set about making sure that `LTextFields` have at least as rich an interface as the `TextField` and `Label` objects. You should, however, resist this impulse. The strength of composition lies in its capability to bring structure to complexity. Just as the simpler interface of the steering wheel, gas pedal, and brake harnesses the complexity of the automobile, so too you should use composition to build ease-of-use into your objects. [For those times when all the gears and levers need to be visible, such as the cockpit of a jetliner, inheritance rather than composition is the better choice.]

Generalize When Possible, Specialize When Necessary. Another problem you'll encounter when creating composite objects is deciding how general or specific each part should be. By making very specific parts—parts specialized to do only a single task—you reduce the effort that a user needs to expend in using your part. On the other hand, if you create only specialized parts, you'll soon be drowning in classes. A better solution is to specialize only when necessary.

Tip: As mentioned earlier in the chapter, stable complex systems tend to be composed of a few simple parts, arranged in different ways. If you want your system to be maximally stable, create generalized parts whenever you can.

Prefer A Tree To A Forest. Should your composite objects be shallow or deep?

In a shallow containment hierarchy (*forest*), most composite objects have many fields, and the fields are composed of relatively basic types. (These basic types aren't necessarily the language's primitive types, but could be simple concrete components, strings, or other "built-in" objects.) As the previous section mentioned, building a system in this way—from only a few parts—tends to promote stability.

In a deep containment hierarchy (*tree*), your system is composed of more vertical layers.

The main argument in favor of deep hierarchies is that, as humans, our short-term memory can handle only a limited amount of information. To handle greater amounts of information, we combine related pieces together into chunks. When you create a deep hierarchy, you limit the amount of information you need to absorb at any one level.

Chapter 9 Implementing Class Relationships: Inheritance and Interfaces

9.1 From Encapsulation to Inheritance

Encapsulation—the specification of attributes and behavior as a single entity—allows us to build on this understanding of the natural world as we create software. By creating abstract data types that model categories in the "real world," we have confidence that our software solutions closely track the problems we are trying to solve, rather than think in terms of computer files and variables. *Inheritance* adds to encapsulation the ability to express relationships between classes.

- Superclasses and subclasses can be arranged in a hierarchy, with one superclass divided into numerous subclasses, and each subclass divided into more specialized kinds of subclasses.

- A classification hierarchy represents an organization based on generalization and specialization. The superclasses in such a hierarchy are very general, and their attributes few.

Reuse and Organization: Inheritance gives you a way of taking existing classes and using them to make new and different classes. Rather than build a `BeveledPanel` class from scratch, for instance, you can use the Panel class you already have and just write code for the aspects that are different. Your new `BeveledPanel` class will silently and invisibly inherit all the fields and methods from its superclass, `Panel`; you don't have to do any extra work at all.

Kinds of Inheritance Relationships: When working with inheritance, the new class *is a kind of* the existing class (often called *isA* relationship. However, there are three forms of this *isA* relationship:

- **Extension relationship.** A subclass may *extend* an existing class, so that all the data members and methods of the class are left intact, and only new methods or fields are added. That is we inherit the public interface and implementation of the superclass.

- **Specification relationship.** A superclass may specify a set of responsibilities thhat a subclass must fulfill, but not provide any actual implementation. That is, we only inherit the interface of the superclass.

- **Combination of extension and specification.** We call this *polymorphic inheritance* because its principal value lies in its ability to provide specialized behavior in response to the same messages.

9.1.1 Discovering Inheritance Relationships

After you decide on the classes and objects in your system, and after you decide which objects cooperate together to form association relationships, you are ready to begin looking at the *class relationships* in your system.

Your inheritance hierarchy will evolve from both iterative rounds of top-down specialization and bottom-up generalization.

Specialization: Is when one class is semantically *is-a-kind-of* another class. The subclass inherit both data and methods from parent, and won't be redefined. (E.g. the `CoffeMachine` is a specialized form of the more general `VendingMachine` class, and won't redefine but use the `collectMoney()` and `makeChange()` inherited methods.

Substitutablity and Subtyping (Liskov Principle): When you decide that one class is a specialization of another, you need to see whether it's a *subtype*. Having one class as a subtype of another means that you can use the new, more specialized type anywhere an object of the original type would appear.

This test—the *Liskov Substitution Principle*—is especially important when you're designing class hierarchies.

In *strict subtyping*, you have the additional requirement that not only does every subclass have to carry out the meaning of the interface defined in the superclass, but it must also rely on the methods already defined in the superclass to do so.

Check out the SOLID principles of Object-Oriented design.

Generalization: Sometimes when working on an inheritance hierarchy, you'll notice that several classes have data and methods in common. In the object-oriented world, this is a clue to look for generalization.

Tip: When looking at the superclass in a such a relationship, you can call it a generalization. When looking from the other direction, it's a specialization. The only difference is in the way the relationship is originally discovered.

For example, if writting some classes `CoffeeMachine` and `SodaMachine`, you might notice the common factors in these classes, and you could use those factors to create a new generalized superclass (say `VendingMachine`) that encapsulated the shared information.

Specification: Sometimes you kwow *what* a subclass should do, but simply have no idea *how* it should do it. In this case you want to *specify* what the soubclass should do, and let the compiler make sure each subclass complies.

For example, consider a class `Shape`, wich we know it should have a `draw()` method, but how are we going to paint an abstract shape? Specific shapes, like `Circle` and `Square` that inherit `Shape`, must implement their specific `draw()` method.

Specialization and Specification: When you have a class that needs to specify some behavior that a subclass must perform, and yet has some behavior of its own to pass on to its subclasses, you find yourself needing a combination of specialization and specification. (Maybe, the above example fits in this case.)

Pure Specification: You may find remaining places where you want to factor out common behavior. The problem is that the objects sharing that common behavior may be otherwise unrelated. Here we use these *pure specifications* (also `interface` in Java).

This inteface mechanism differs in from other forms of inheritance in several ways:

- When a class implements an interface, it doesn't receive any method implementations or nonfinal fields from the interface. Just public interface and constant public data.
- A class may implement several interfaces.
- Classes that implement the same interface don't need to be substitutable in the Liskov sense, except with respect to the interface they implement.

Inheritance of Implementation: Contraction. This is a form of inheritance which must be avoided. In this form, we inherit a superclass for a specific functionality, and delete all other uninteresting interface. This should be avoided for two reasons:

- You're violating the substitutability rule.
- It is more work than the right thing to do.

Tip:You might find yourself tempted to use contraction in one last place—when you have misidentified an *isA* relationship and don't want to go back to square one. In this case, the best thing to do is redesign the system.

9.2 Implementing Inheritance

The first step in designing any object is deciding what you'd like it to do. Start with the fundamental requirements first, then refine design as you iterate through succeeding generations.

Choosig An Implementation Mechanism: What form of inheritance is appropriate? To answer the question, look at the requirements of the class, then ask the following questions:

1. Is the new class a subtype of an existing class? Can you use the new class anywhere you would have used the previous one, without making any changes? If so—if the new class is substitutable for the existing class—you should use inheritance.
2. Do you want the new class to inherit behavior, or only an interface, from the existing class?
3. Will the existing behavior be modified, or are you just adding new behavior? Most of the time, a subclass will make some change to the behavior of its superclass. *Tip*: Even if you don't plan on making such a modification at first, you may choose to make one in the future. Thus, in almost all cases that call for inheritance, you'll want to use polymorphic inheritance rather than simple extension.

Inheritance and Constructors: Whenever you use inheritance, the constructor for your new subclass calls the constructor for its superclass before it begins executing its own statements.

You may explicitly choose to call a specific superclass constructor.

```java
public class Dog extends Animal {
    // Constructor
    public Dog(int weight, ...) {
        // Superclass constructor
        super(weight);

        // Do more stuff
    }
}
```

Inheritance and Overriden Methods: To override an inherited method, you must declare the method with exactly the same signature as in the superclass method.

Sometimes, we simply want to augment the overriden method, rather than defining a completely new behavioror. If you want to replace the behavior of your superclass wholesale, you don't have to do anything special. You can use the `super` object (not to be confused with the `super()` method, which can be used only in a constructor) to invoke a superclass method in such cases.

```java
public class Panel3D extends Panel {
    public void paint(Graphics g) {
        // Call the Panel.paint() method
        super.paint(g);

        // Augment the method ...
    }
}
```

9.3 Writing and Using Interfaces

Extension, or *inheritance*, means that a subclass is a kind of its superclass—that the subclass is a more specialized form of its more general superclass. By contrast, the *interface* mechanism is used to model a relationship between classes that share a specific set of behaviors (an interface) but don't need to be otherwise related. Because of this, implementing an interface is sometimes called *inheritance of specification.*

Consider for example the interface `Nameable`:

```java
public interface Nameable {
    public String getName();
    public void setName(String name);
}
```

The following two classes:
```java
class Dog extends Animal implements Nameable
class Employee implements Nameable
```
implement the `Nameable` interface, but are not related by any means. The interface just sets a contract that a class that implements it can perform a certain action or behavior, and it may share this capability with other classes, but it's not necessarily related to those other classes.

9.5 `ImageButtons`: An Extended Inheritance Example

We want an `ImageButton` class that have buttons displaying images instead of text. In Java AWT we have both `Buttons` and `Images` classes. Which one can we use? It is that an `ImageButton` is a `Button` that contains an `Image`, or a `Clickable Image`? There's no hard-and-fast rule for making these decisions, but you should start by asking yourself, "Is an `ImageButton` a special kind of `Button`?" If your answer is yes, you should probably use inheritance and extend the `Button` class. If you do this, remember the following:

- Everything a `Button` does, an `ImageButton` should do as well. After all, your design asserts that an `ImageButton` *isA* `Button`.
- Anywhere you have a reference to a `Button` object, you should be able to refer to an `ImageButton`. The `ImageButton` should be perfectly substitutable for the built-in `Button`, and clients should be unaware that any substitution has been made.
- The `ImageButton` may extend the data or behavior of the `Button` class, but it shouldn't attempt to "remove" methods from the Button class by overriding them with empty methods. (Otherwise the `ImageButton` is not really a `Button` after all, but just has some behavior that you find attract.)