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*.

Renter
– theLender : Lender – INITIAL_QTY : int = 5
+ void rentItem(Renter) + void returnItem(Renter) + void tellResult(String)