Unit 2. Encapsulation

In this Chapter:

Unit 2. Encapsulation {.unit-1-background}

2.0 Encapsulation Design

Sam is working on a new programming project: to play the card-game War. As he works through the design of the project, he is having difficultly settling on the data type for a single playing card. Should it be an integer with the value $0 - 51$? Should it be a structure with two member variables? While C++ offers him many built-in data types to choose from, there is nothing which is exactly a playing card. This project would be so much easier if the designers of the C++ language added a special data type for him.

Objectives

By the end of this chapter, you will be able to:

- List and define the components of a UML class diagram
- Enumerate the four encapsulation design rules
- Design a new data type to fit the needs of a given project
- Explain why custom data types simplify the design of large projects

Prerequisites

Before reading this chapter, please make sure you are able to:

- Use UML class diagrams to describe a structure (Chapter 1.3)
- Define functional cohesion and explain how cohesion helps programmers design better functions (Procedural Programming in C++ Chapter 2.0)

What is encapsulation and why you should care

Encapsulation is the process of separating the use of an item from its implementation. Perhaps this is best explained by analogy. The owner of a car can become well-versed in driving a car (use) without understanding the engineering behind how it is built (implementation). If the interface between the internal workings of the car and the driver is sufficiently clear, and if the engineering of the car is sufficiently reliable, a car-owner can use a car for decades without knowing the compression ratio or spark plug firing sequence of the engine. The designers of the car went to great lengths to shield the driver from this knowledge so the driver can focus on the one think he cares about: getting the car safely from point A to point B.

This principle of encapsulation is also at work in the programming world. However, instead of just shielding the user from the internal workings of our code, we strive to shield the client (other programmers we are collaborating with) from the internal workings of our code. Programmers should design their functions and custom data types in such a way that the other programmers who use our code need to only concern themselves with what the code does, not how it does it.

Two types of encapsulation

There are two main forms of encapsulation: procedural encapsulation commonly known as modularization, and data encapsulation (Schach, 2007, p. 196).

Procedural encapsulation

Procedural encapsulation is the process of modularizing a collection of related operations into a single function. A well modularized design will free the client of a function from having to know about how the function was implemented. Similarly, a well modularized design will allow the programmer to alter how a given function solves a problem without concern as to how the clients will behave.

The main tools we have to measure procedural encapsulation are cohesion and coupling (Robertson, 2004). As you recall, cohesion is a measure of the internal strength of a module. The stronger the cohesion, the more focused the module is on performing a single task. Coupling is a measure of the information interchange between modules. The simpler the interface, the easier it is to use the function in more contexts.

Functions exhibiting sound modularization techniques have strong procedural abstraction. This facilitates both repurposing the function and changing the function implementation without side effects in the rest of the program.

Data encapsulation

Object oriented design adds another form of abstraction: data encapsulation. Data encapsulation is the process of creating new data types with interfaces useful to the clients of the type yet whose implementation is not necessarily known. This technique allows the programmer to focus on using the data rather than maintaining the data. Data encapsulation is also called data abstraction and information hiding.

Data encapsulation is the process of creating a new type, called a class. A class is the collection of data-items and the operations on those items. These are called member variables and member functions.

- Member variables. A member variable is one of a collection of items in a class necessary to represent a given abstract data type. While these could consist of built-in types, they could also consist of other custom data types.
- Member functions. A member function, typically called a method, is a part of the class enabling clients of the class to access member variables, and manipulate and/or compute with them.

When a variable is declared of a class type, it is called an object. The user of the class, or the client, is able to treat this class like any other variable to help him satisfy the needs of his program. The author of the class tries to design the class in such a way as to maximize the utility of the new data type to the client.

UML class diagrams

Recall from Chapter 1.3 the UML class diagram we used to describe structures.

Complex
real
imaginary

This consisted of two parts: the first row is the name of the structure (Complex in this case) and the second is a list of the member variables (real and imaginary). The class diagram to describe a class is similar except a third row is added to describe the functions (called member functions or methods) associated with the class.

Complex
real
imaginary
display
set

There is one additional level of complexity to the UML class diagram: access modifiers. An access modifier allows the designer of the class to specify whether a given member variable or method is accessible to clients of the class or just to methods of the class itself. If a member variable or method can be freely accessed by anyone, including the client or methods within the class, then it is called public. A public method or member variable is signified with the + sign on the UML class diagram. However, if access is restricted to only a method, the member variable or member function is called private and the - sign is used in the UML class diagram:

ClassName
+ publicVariable
- privateVariable
+ publicMethod
- privateMethod

For example, consider the string class built into the STL. There are externally-facing methods (`size()` and `c_str()`, for example), and methods not visible to the client (`reallocate()`). There are externally facing variables (well, not really, but we will say so now for the sake of argument that the array is externally facing) and private variables (such as length). In this case, the class diagram would look like:

String
+ `array`
- `length`
+ `size`
+ `c_str`
- `reallocate`

Designing classes

When designing a class, the foremost concern that should be in the programmers mind is how convenient the class will be for the client to use. In other words, can the class be designed in such a way that it is as useful as possible and requires the smallest amount of work to integrate? To accomplish this goal, one simple thing is needed:

Design the interface of the class before worrying about implementing the class

Typically it is best to design the publically visible methods before the implementation details are considered. A primary goal of the class author is to make the class as useful to the clients as possible. This can be achieved through careful consideration of all the possible needs of the clients, making a method custom designed for each of these needs. In other words, focus on the interface of the class rather than the data.

A common mistake for new class designers is to start the other way around: with the data representation. It is easy to lose sight of the needs of the client when we adopt a data-centric perspective.

As a general rule, start with the class diagram and design all the publicly visible methods (with the + sign) first. Only after this and all other classes are designed do we begin to explore how those methods will work. At this point in time, the private methods and the data representation strategies are addressed.

Because encapsulation is such a core part of using classes effectively, four design rules have been developed. (Wick, Stevenson, & Phillips, 2004). Each of these rules are intended to help the programmer focus his efforts on making the class as easy for the client to use as possible. These rules are:

1. Define properties independent of data. In other words, consider first how to receive information from the client and send information to the client in a way that is as convenient for the client as possible.
2. Use only interval data if it is truly interval. Do not force the client to have to know that you start the week on Sunday rather than Monday. He should not have to be burdened with implementation details such as that.
3. Use getters and setters if we have interval data. When data needs to be validated, make sure that the only public interfaces go through validation checks.
4. Don’t use getters and setters if you don’t validate. Why make access to data more inconvenient that necessary of getters and setters do not validate?

Rule #1: Define properties independent of data

Define all properties of an object in the language of the domain and independent of the member data used to implement the object

When defining an object, first design all the publically visible interfaces in a way that makes them as easy to use as possible. All data returned to the client must be immediately useful without further processing. Similarly, all data requested by the object should be readily producible by the client.

A fundamental principle in understanding data encapsulation is the difference between object properties and member data (Wick, Stevenson, & Phillips, 2004).

- Properties: An object property is a conceptual characteristic of the object. It is the client conceptualizes or thinks about the object. Another way to think about it is the properties are how the client will want to use or manipulate the object.
- Data: The object data is how the object is actually implemented. The author of an object cares deeply about the data representation of the object data because he will have to interface with it when he writes the methods. Note that if any of the implementation details of the data representation are visible outside the object, then clients of the object could likely depend on these details. This will effectively prohibit the author of the object from making any meaningful changes to the data representation and thus defeats the goal of data encapsulation.

One way to see the difference between an object’s properties and data is the built-in data type int. Programmers are encouraged to think of an integer as a number without a decimal. This understanding is sufficient in most cases; you can perform integral math and perform integral comparisons in a way that is highly predictable. The data representation is quite different. Internally, an integer is a collection of 1’s and 0 's corresponding to powers of 2 . Because all the operations working with integers are built into C++, programmer does not have to be aware of the internal representation of integers. This results in robust data encapsulation of the integer type.

For example, consider an object holding the time of day. The client of the object will want to set the hour and minute, compare if two times are equal, and display the time on the screen. All these are properties of the time. The author of the object may choose to implement the member data as a single integer specifying the number of seconds since midnight. While the data representation should be very elegant and efficient, it should be completely opaque to the client.

Time
- secondsSinceMidnight
+ setHour
+ setMinute
+ compare
+ display
- validate

Rule #2: Only use interval data if the data is truly interval

If a property has a small range of possible values and that range is not likely to change in the future, define “test()” and “changeTo()” methods for that purpose that test for or change to each possible value. Do not define getter and setter methods.

In order to understand this rule, it is first necessary to discuss some types of numbers. Continuous data is defined across a continuum, such as GPA $(0.0-4.0)$ and Celsius temperature $(-273.15^{\circ}-\infty)$. Note that there are an infinite number of possible GPAs even though the range is limited. Discrete data is where every value is distinct, such as ACT score ($1-36$). There is no 24.5 ACT score. Interval data can be either continuous or discrete but there must be a well-defined zero point. In other words, everyone must agree what a GPA of $3.7, 27^{\circ}\mathrm{C}$, and a 24 ACT score means. Numbers that are not interval include the day of the week and the “ace of spades.” Note that we can easily assign either of these values to an integer, but we will need to make an arbitrary decision of what the zero point means. For example, is Sunday the zero value or possibly Monday? The choice is arbitrary. Is the “ace of spades” represented as zero or possibly is the " 2 of clubs" represented as zero? If we are going to expose to the client some data that is not truly interval, then we are exposing our arbitrary zero decision.

A better form of encapsulation would be to hide any underlying encoding and present the interface to the client on his terms. For example, consider an object holding the day of the week. While the implementation of this object will probably use an int (or char) to represent the day, a few other implementation details need to be solved (such as which day is zero?). All these design decisions should be opaque to the client of the class. The client would want to perform the following tasks:

- test(): Determine if two days are the same.
- set(TUESDAY): Set a day to a given value.
- adjust(4): Advance by a fixed number of days. Note that we will handle week wrapping (Saturday → Sunday, for example).

- display(): Put “Tuesday” on the screen.

DayOfWeek
- day
+ test
+ set
+ adjust
+ display
- validate

Rule #3: Use setters and getters if we have interval data

If a property has a large range of possible values, or if the set of values is likely to change in the future, define public access methods for that property. When defining these access methods, follow the traditional getx() and setx() naming conventions.

When working with interval data (a range of values with a large number of intermediate values, such as a GPA or the temperature outside), allow the client of the class to get and set values directly. There is one caveat to this rule however. Good encapsulation design dictates that we should not burden the client with the work to keep the data of a class valid. It should be impossible for the client to set data into an invalid state. Fortunately, the class author has a pair of tools to accomplish this: getters and setters.

A getter (traditionally called get() or getProperty()) is a method whose sole purpose is to give the client access to member data. Note that it may be that the member data is stored in a different format than the client expects. In this case, it is the job of the getter to translate member data into client properties. Consider, for example, a class representing the time of day. The data for the class is a single member variable storing the number of seconds since midnight. The properties of the class are the second, minute, and hour of the day. Thus the pseudocode for the three getters would be:

```
Time::getSecond()
    RETURN time % 60
Time::getMinute()
    RETURN (time / 60) % 60
Time::getHours()
    RETURN time / 3600
```

A setter (traditionally called `set()` or setProperty()) is a method serving two purposes: to translate client properties into member data, and to validate that the member data is within the valid range (this will be mentioned in more detail in Chapter 2.3). Thus one of the setters for our Time class would be:

```
Time::setSeconds(seconds)
    IF (seconds < 0 OR seconds \geq 60)
        THROW
    time \leftarrow seconds + getMinute() * 60 + getHours() + 3600
```

Another example is a class storing a student's GPA. Since the data is interval $(0.0 \rightarrow 4.0)$ and the property is the same as the data (both floats), getters and setters are great candidates. The getter will simply return the value in the member data variable (no translation from data to properties required). The setter, however, will first validate the parameter to ensure it is in the correct range $(4.0 \geq \text{gpa} \geq 0.0)$ before modifying the member variable.

GPA
- gpa
+ get
+ set
- validate

The `set()` function for our GPA class would be:

```
GPA::set(input)
    IF 0.0 < input < 4.0)
        gpa \leftarrow input
```

Rule #4: Don't use getters and setters if you don't validate

If no validation is necessary, use public visibility to provide access to member data of a pure data-structure. Use traditional getters/setters only if validation is necessary. One principles of data encapsulation is to ensure the data is always valid. If your implementation has no data validation requirements, then using methods to access the data serves little purpose. In these very limited cases, public access to member data is faster than function calls and provides no loss in data hiding.

For example, consider a class consisting of complex numbers. The two member data variables (the real and the imaginary component in this example) have no data validation associated with them because they are simply numbers. In this case, it serves no purpose to restrict client access to getters and setters. It would be more efficient to publicly expose the variables to the clients.

Complex
+ real
+ imaginary
+ display
+ add
+ subtract
+ multiply
+ divide

Final thoughts

When designing programs with procedural tools, structure charts and pseudocode were the best tools to draft out a solution. Using the Object-Oriented design tools and methodologies, we still use structure charts and pseudocode in the design process. However, the main design tools are UML class diagrams. You may find that the data-structure part of the design document becomes the focus of your design efforts.

Example 2.0 - Playing card

This example will demonstrate how to use the UML class diagram as an important design tool for an Object Oriented problem. Here, we will implement the data-structures part of a design document.

Design a program to play the card game of War. The game of War is played by dealing out an entire deck of cards to two players. Each player will turn the top card over at the same time. The winner will take both cards and put them in the bottom of his stack. There is also a special procedure for dealing with the case when both cards are of the same rank and thus a tie.

First we need a class to describe a playing card.

Card
- card
+ display
+ setRank
+ setSuit
+ getRank
+ getSuit
+ isSameRank
+ isLargerRank
+ isSmallerRank
+ isSameSuit
+ isLargerSuit
+ isSmallerSuit
- validate

Note how the design of the class tries to take into account every possible use of the card for the game of War. It might be that some of the methods are not used by the game. We also need a class to describe a collection of cards called a hand:

Hand
- cards
+ getTopCard
+ setToBottom
+ getSize

The client does not need to know how the collection of cards is stored. In fact, the only things he will need to do is to get the top card off the stack, put a new card on the bottom, and find out how many there are in the collection.

As a challenge, modify the Card class to handle the game of Klondike solitaire. What additional methods would be necessary for this?

The complete design document for the card game of War: Example 2.0: Card Game War The working program can be played at: /home/cs165/examples/2-0-war.out

Problem 1

Consider the following UML class diagram:

Point
+ x
+ y
- invalid
+ get
+ set
+ display
- clear

Find the corresponding variable, function, and class name for each of the following:

- Private member function:
- Private member variable:
- Class name:
- Public member variable:
- Public member function:

Please see page 110 for a hint.

Problem 2

Identify the name corresponding to the following descriptions:

- The measure of information interchange between functions

- A collection of tools to facilitate OO design
- The process of creating a protective layer around data
- The data associated with a class
- A subroutine or procedure associated with a class
- The name of a function designed to access data in a class
- The definition for a new data type
- The name of an instantiated class
- The name of a function designed to alter data in a class
- The process of breaking a large problem into smaller ones
- The extent in which a function does one thing only

Problem 3

Design a class to hold the position on a chess board.

Problem 4

Design a class to hold a date (month, day, and year).

Problem 5

Design a class to hold a playing card. Note that there are 4 suits (● ● ●) and 13 ranks (2-10, jack, queen, king, and ace)

Challenge 6

Design a class to hold an integral number (number without a decimal) with a potentially infinite number of digits.