Part I
Program Design Fundamentals

This first part reinforces your understanding of some basic program design concepts. The author expects these concepts to be familiar to many of you. Chapters 2 through 10 are covered in the first two weeks of a 15-week semester for a 3 credit-hour course on software design. While reviewing program design fundamentals, you may choose to focus only on object-oriented program (OOP) design or structured program (SP) design by using only those chapters whose title includes the prefix OOP or SP.

2 Program Design Criteria and Simple Design Models

The objective of this chapter is to review the criteria used to evaluate a program design and to introduce a few simple design models.

2.1 Preconditions

The following should be true prior to starting this chapter.

- You have experience developing software code.
- You understand that a structured program contains functions that are grouped into modules. An entire solution may consist of many modules where each module contains many functions.
- You understand that an object-oriented program contains methods that are grouped into classes. An entire solution may consist of many classes, where each class contains many methods.
- You understand that software design is a:
  - Process that, when followed, will produce artifacts representing an abstraction of the implementation code. These abstractions describe the structure and/or behavior of the solution.
  - Description of the structure and behavior of software at a higher level of abstraction than the code.
  - Collection of artifacts that describe the architecture, data, interfaces, and components of the software.
  - High-level description of the knowledge represented by the code.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_2

2.2 Concepts and Context

The bottom-up learning approach used in this book is briefly described. A description of program design criteria to evaluate program code is then described. This is followed by a description of a few simple design models to describe the structure or behavior of a solution.

2.2.1 Case Study and Bottom-Up Approach

The same case study is used throughout the remainder of this book to help illustrate software design thinking and alternatives. This case study is a simple Address Book Application (ABA) that will store contact information of people that you know. In part I of this book, the focus is on an ABA that does not store contact information persistently. The contact data is stored only when the application is running; each time the application is started the address book contains no data.

Storing data non-persistently was chosen as the first bottom-up design to avoid having to introduce persistent data storage technologies (e.g., XML, relational database). This should allow you to use the material in part I as a review of prior programming knowledge, which hopefully includes program design concepts.

2.2.2 Evaluating Program Designs

We'll assess the quality of the program designs using the following three criteria. Note that both structured and object-oriented program designs are discussed for each criterion. The acronyms SP (structured programming) and OOP (object-oriented programming) are used throughout this chapter to denote whether the concept is related to structured and/or object-oriented programming.

2.2.2.1 Separation of Concerns

Each function (in SP) or method (in OOP) should do one, and only one, thing. When you see a function or method that is designed to do X, but also does Y, split the function/method into smaller functions/methods so that the Y processing is in a separate function/method (which may be called from the function/method doing X). As you put distinct processing in separate functions/methods, look for opportunities to create functions/methods that are reusable.

As you group functions into modules (when doing SP) or methods into classes (when doing OOP), each module or class should have one, and only one, responsibility. For example, a Student module/class would contain functions/methods that perform computations on student data. Another module/class might be needed to represent processing performed on data for a Teacher. The Student module/class should not have any functions/methods that process Teacher data, and the Teacher module/class should not have any functions/methods that process Student data.

2.2.2.2 Design for Reuse

Redundant code should be eliminated, when possible. When you see code duplicated in two or more places, create a function or method definition and replace the duplicated code with a function or method call. A more challenging application of design for reuse is when two segments of code are similar but not identical. In this case, these similar segments of code should be put into a function/method. This new function/method will likely include parameter variables that are used to distinguish the processing described by the two (or more) original code segments.

2.2.2.3 Design Only What is Needed

Only write code for the processing that is required. Do not add extra functionality. Why? This extra functionality needs to be tested and debugged, even though it is not required. Since testing and debugging code is hard (and time consuming), this places an extra burden on the developer and tester. As Benjamin Franklin wrote many years ago—"time is money" [1]. Since developing software is a labor-intensive effort, no need to do any more than what is necessary!

2.2.3 Design Models that Describe Structure

A hierarchy chart shows the structure of an SP solution while a class diagram shows the structure of an OOP solution.

2.2.3.1 Hierarchy Chart (SP)

A hierarchy chart depicts the structure of a program by showing the call hierarchy of the functions. It shows a top-down view of the functions where the function at the top of the chart is the function that begins to execute when the program is run.

For example, Fig. 2.1 shows that Func_A is the first function to execute when the program executes since it appears at the top of the hierarchy chart. The left-to-right ordering of Func_B and Func_C in the chart indicates that Func_B is called before Func_C is called. Since both functions are immediately below Func_A, Func_A calls each of these functions. Finally, the chart shows that Func_C calls Func_D.

A hierarchy chart does not show how many times a function is called. In the example, Func_A could have a call to Func_C inside a loop statement. During program execution this may result in Func_C being called many times. A hierarchy chart does not convey this iterative program design element.

2.2.3.2 Class Diagram (OOP)

A class diagram shows the structure of a class by listing its attributes and methods (aka operations). In Fig. 2.2, Class_A is a class that contains three attributes and four methods.

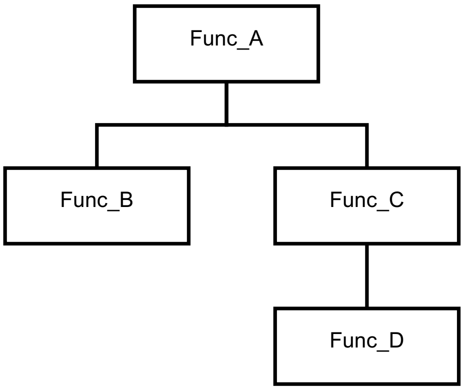

Fig. 2.1 Hierarchy chart example

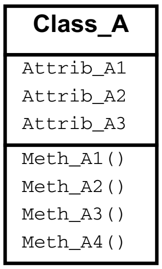

Fig. 2.2 Class diagram example

When a class diagram includes two or more classes, the diagram may use a class relationship type to describe how the two classes are related to each other. These relationship types include association, aggregation, composition, inheritance, and realizes. These relationship types will be discussed in later chapters. In addition, a class diagram will show an access modifier for each attribute and method. An access modifier indicates the code that can reference/use the attribute or method. Access modifiers will also be discussed in later chapters.

2.2.4 Design Models that Describe Behavior

The two design models described below express the behavior of a solution and may be used to describe a structured (SP) or object-oriented (OOP) solution.

2.2.4.1 Nassi–Shneiderman Diagram (SP or OOP)

This modeling technique is used to describe an algorithm performed within a solution. The entire diagram is a rectangle that contains only three types of shapes.

1. A rectangle that represents statements being executed in sequence.
2. A carpenter's square (or a sideways capital "L") that represents a loop condition and a block of statements executed each time the loop condition is true.
3. A shape that looks like the back of an envelope that represents conditional logic.

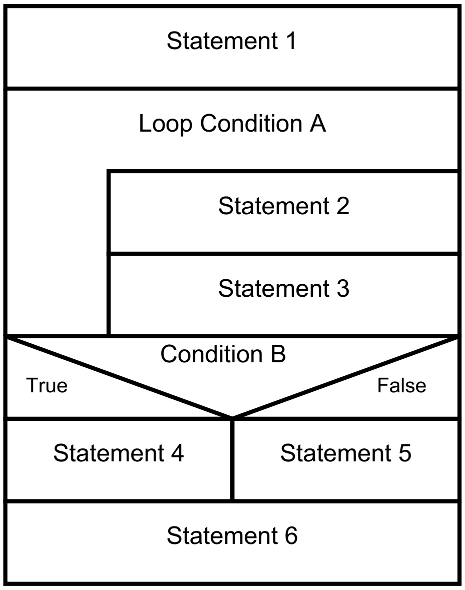

Fig. 2.3 Nassi–Shneiderman diagram example

Execution of an algorithm starts at the top of the diagram and ends when execution falls out of the bottom of the diagram.

In Fig. 2.3, execution would begin with Statement 1 and then continue to Loop Condition A. As long as this condition remains true, iteration occurs on Statement 2 and Statement 3. When Loop Condition A becomes false, execution continues with Condition B. When Condition B is true Statement 4 then Statement 6 is executed. When Condition B is false Statement 5 then Statement 6 is executed. In either case, the algorithm ends after Statement 6.

2.2.4.2 Statechart Diagram (SP or OOP)

This modeling technique is used to describe states and transitions between these states. A state is a discrete condition or situation during the life of a system or object. A transition is an action or event that is applied to a state. A transition usually results in a change to a different state. However, a transition may result in no change in state (i.e., the transition points back to the same state that initiated the action or event). Each state is given a name and most transitions are labeled to indicate the action or event that causes the transition. A statechart diagram is a powerful modeling technique that can describe the behavior of many different types of systems. Examples in Chaps. 3 and 4 will show how this type of model can be used to describe the behavior of the case study software.

In Fig. 2.4, the start state immediately transitions to state-1. When one of three events occurs (either event 1, event 2, or event 3) a transition would occur to a new state (either state-2, state-3, or state-4, respectively). Once in state-2, an occurrence of event 4 results in no change in state. The end state is reached by being in state-2 and event 6 occurring or by being in state-4 and event 7 occurring.

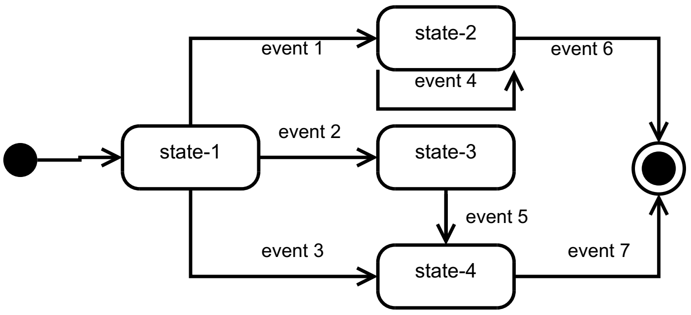

Fig. 2.4 Statechart diagram example

2.3 Post-conditions

The following should have been learned after completing this chapter.

- Program design should be evaluated using at least three criteria:
  - Separation of concerns
    Each named element of code (e.g., function or method) should do one, and only one, thing.
    When functions are grouped into a module, or when methods are grouped into a class, each module or class should have one, and only one, responsibility.
  - Design for reuse
    Redundant code should be eliminated. Parameters should be used to generalize a function or method so that it may be called in situations that are similar (but not exactly the same).
  - Design only what is needed
    Adding code for processing that is not required will increase the time needed to test the software. This extra cost should be avoided whenever possible.

- Program design models include the following.
  - A hierarchy chart is used to show the structure of a solution by illustrating the call hierarchy of functions in a structured programming solution.
  - A class diagram is used to show the structure of a solution by identifying the attributes and methods for each class in an object-oriented programming solution.
  - A Nassi–Shneiderman diagram is used to show the behavior of a solution by explaining the algorithmic processing.
  - A statechart diagram is used to show the behavior of a solution by describing the states the software is in and how the solution transitions to different states during execution.

2.4 Next Chapter?

If you are interested in seeing how the program design criteria and program design models may be applied to a small object-oriented programming solution, continue with Chap. 3. This chapter will introduce you to the Address Book Application case study using Python and Java.

If you are interested in seeing how the program design criteria and program design models may be applied to a small structured programming solution, continue with Chap. 4. This chapter will introduce you to the Address Book Application case study using Python and C++.

Exercises

Discussion Questions

1. What are the benefits to applying *separation of concerns* to a program design?
2. Can you think of a situation where you would want to ignore separation of concerns when developing a program design?
3. Besides the "time is money" argument for the *design only what is needed* program design criteria, are there other reasons why a developer should only implement what is needed?
4. What are the benefits to applying *design for reuse* to a program design?
5. Can you think of a situation where you would want to ignore design for reuse when developing a program design?

Reference

1. Franklin B (1748) Advice to a young tradesman. Retrieved 30 May 2017. https://founders.archives.gov/documents/Franklin/01-03-02-0130