Part II
Introduction to Software Design

This second part introduces the characteristics of a good software design and the Model–View–Controller (MVC) architectural pattern. Chapters 11 through 16 represent the core software design topics that are further explored in part III. The author covers chapters 11 through 16 in weeks 3 through 5 of a 15-week semester for a 3 credit-hour course on software design.

Characteristics of Good Software Design

11

The objective of this chapter is to introduce the characteristics that may be used to determine if a software design is good.

11.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You have evaluated program code using these five criteria.

11.2 Concepts and Context

This chapter will focus on six characteristics of good software design and the concept of abstraction.

11.2.1 What are Characteristics of a Good Software Design?

This section describes six characteristics of good software design: simplicity, coupling, cohesion, information hiding, performance, and security. Table 11.1 describes these characteristics from two perspectives—everyday usage [1] and software design usage.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_11

Table 11.1 Definitions

Term	Everyday usage	Software design usage
Simplicity	The quality or state of being unmixed or uncompounded	The amount and type of software elements needed to solve a problem
Coupling	Act of joining together to form a couple A device that couples two things together	The degree to which each program module relies on other modules [2] Identifies the connectedness of a module to other modules [3] Measure of interdependence between two or more modules [4]
Cohesion	State of cohering or of working together	The degree to which a software module is strongly related and focused in its responsibilities [5] A module that has a small, focused set of responsibilities and single-mindedness [3] An attribute of a software unit that refers to the relatedness of its components [6]
Information hiding	Not applicable	A component encapsulates its behaviors and data, hiding the implementation details from other components [7] Modules should be designed so that information (i.e., algorithms and data) contained within one module is inaccessible to other modules that have no need for such information [3]
Performance	Carrying into execution or action The amount of useful work accomplished estimated in terms of work needed	The analysis of an algorithm to determine its performance in terms of time (i.e., speed) and space (i.e., memory usage). This analysis is useful when comparing two algorithmic approaches that solve the same problem
Security	The condition of not being threatened, especially physically, psychologically, emotionally, or financially	A set of technical controls intended to protect and defend information and information systems [8]

11.2.1.1 Simplicity

The key phrase in the software design usage definition is "amount and type of software elements needed." The term *software elements* refers to processing elements and data structures found in the solution.

When we think about software elements, it's natural to think like a programmer. From this perspective, processing elements include sequence, selection, and iteration statements, as well as the structural elements of functions (for SP) or methods and classes (for OOP). Data structures include hash tables, trees, graphs, and arrays. Our challenge is to also think like a software designer. From this perspective, processing elements include components and subcomponents (for SD) or packages and classes (for OOD), while data structures include persistent data stores.

Our intuition suggests that a simpler design solution is less expensive to implement, test, and maintain. The acronym KIS for *Keep It Simple*, or its more derogatory version KISS for *Keep It Simple Stupid*, is a common phrase that even has a Wikipedia entry. To summarize this software design criteria, as long as your design solves the problem (i.e., satisfies the requirements), a simpler design is likely a better design.

11.2.1.2 Coupling

The key to understanding coupling is the word *module* found in the software design usage description. Unfortunately, the word module has been used to describe a wide range of physical objects (e.g., the Apollo Lunar Module, modular home, power supply module) and software constructs (e.g., component, package, class, function). In this textbook, a software module shall be any of the following.

For an object-oriented software design:

- A module is typically a class (i.e., a group of logically related attributes and methods). For example, a class that contains methods that validate data entered by a user. For very large software systems, a module may be a package of classes.
- A less commonly used interpretation is that a module is a single method found in a class definition.

For a structured software design:

- A module is typically a group of logically related functions. For example, a set of functions that perform validation of data entered by a user could be considered a distinct module within a structured design. This type of module is also known as a component (or in a very large software system a subcomponent). For example, a Python source code file is called a module. Thus, each Python module should contain logically related functions.
- A less commonly used interpretation is that a module is a single function found in a source code file.

The coupling definitions indicate that coupling describes the amount of connectedness between two or more modules. Thus, a design that has only one module cannot exhibit coupling. That is, coupling can only be described between distinct modules.

Our intuition suggests having fewer connections between modules mean there is less to test, maintain, and fix. Having fewer connections between modules is called *low coupling*, *loose coupling*, or *weak coupling*. A good design is one that exhibits low coupling between its modules. In contrast, more connections between modules means there is more to test, maintain, and fix. Having more connections between modules is called *high coupling*, *tight coupling*, or *strong coupling*. A bad design is one that exhibits high coupling between its modules [4].

Craig Larman introduces *General Responsibility Assignment Software Patterns*, abbreviated GRASP [9]. GRASP represents a learning approach to developing object-oriented designs, where software patterns are used to explain the rationale for a design. (Software patterns are discussed more fully in Sect. 11.2.2.2.) One of the nine software patterns included in GRASP is named *Low Coupling*. Table 11.2 summarizes the Low Coupling pattern documented in [9].

Table 11.2 Summary of GRASP low coupling

Solution	Assign a responsibility so that coupling remains low
Problem	How to support low dependency, low change impact, and increased reuse? Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems: • Changes in related classes force local changes • Harder to understand in isolation • Harder to reuse because its use requires the additional presence of the classes on which it is dependent
Discussion	Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider
Contraindications	High coupling to stable elements and to pervasive elements is seldom a problem. For example, a Java J2EE application can safely couple itself to the Java libraries (java.util and so on), because they are stable and widespread
Benefits	• Not affected by changes in other components • Simple to understand • Convenient to reuse
Background	Coupling and cohesion (described next) are truly fundamental principles in design, and should be appreciated and applied as such by all software developers

11.2.1.3 Cohesion

The software design usage description for cohesion also uses the word module. We use the same meaning for module as described above for coupling. Note that the cohesion definitions refer to *a module*. Thus, cohesion is evaluated for each module within a design. Note the contrast between coupling and cohesion—coupling exists between two modules while cohesion exists within one module.

Our intuition suggests a module with one basic purpose and is indivisible (i.e., it's difficult to split the module into separate more basic units [6]), results in a module easier to implement, test, and maintain. Having a module with one basic purpose that is hard to split into separate units is called *high cohesion* or *strong cohesion*. A good design is one that exhibits highly cohesive modules. Having a module that contains many basic purposes (i.e., many responsibilities) is called *low cohesion* or *weak cohesion*. A bad design is one that exhibits low cohesive modules [4,6].

Larman also includes a High Cohesion pattern in GRASP [9]. Table 11.3 summarizes the High Cohesion pattern.

Table 11.3 Summary of GRASP high cohesion

Solution	Assign a responsibility so that cohesion remains high
Problem	How to keep complexity manageable? In terms of object design, cohesion (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems: • Hard to comprehend • Hard to reuse • Hard to maintain • Delicate; constantly affected by change
Discussion	Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider
Contraindications	There are a few cases in which accepting lower cohesion is justified One case is the grouping of responsibilities or code into one class or component to simplify maintenance by one person—although be warned that such grouping may also make maintenance worse Another case for components with lower cohesion is with distributed server objects
Benefits	• Clarity and ease of comprehension of the design is increased • Maintenance and enhancements are simplified • Low coupling is often supported • The fine grain of highly related functionality supports increased reuse because a cohesive class can be used for a very specific purpose

11.2.1.4 Information Hiding

From a software design perspective, information pertains to data and algorithms. A good design should hide data and algorithm details from design elements that do not need to know this information. For example, a software application may need to save data to a persistent data store (e.g., a relational database). Only the design element responsible for saving data needs to know that it's using a database. All other design elements would simply call a method/function (to save data) without knowing details of how this is done. This allows the persistent data store to be changed (e.g., to an XML file) without having to change other parts of the design.

11.2.1.5 Performance

When two software designs provide a solution to the same problem space, and when these two solutions are similar in their simplicity, coupling, cohesion, and information hiding, then differences in performance may decide which design should be implemented. While it is an obvious statement, it is worth saying—a design that has better performance is a better design. Better performance means faster if we're measuring time and less if we're measuring space. Typically, time is measured using Big O notation and space is measured in bytes, kilobytes (KB), megabytes (MB), or gigabytes (GB).

11.2.1.6 Security

Security is not just a concern when we are programming, it is a concern during the entire software development life cycle. As discussed in Chap. 8, this book focuses on information security—the creation and application of security controls within a software solution. Information security within a software design is about the confidentiality, integrity, and availability of data/information.

Chapter 8 discussed security from a program design perspective. The five secure programming practices described in Chap. 8—data input validation, data output validation, exception handling, fail-safe defaults, and type-safe languages—should also be included when developing a software design.

The SEI CERT describes four security practices in their Top 10 Coding Practices that are directly related to software design [10]. These four practices are the following.

- Architect and design for security policies (discussed in Chap. 24).
- Keep it simple (see simplicity in Sect. 11.2.1)
- Practice defense in depth (discussed in Chap. 24)
- Use effective quality assurance techniques (discussed in Chaps. 23 and 24).

11.2.2 Abstraction: The Art of Software Design

Transitioning your thought process from program design to software design involves the use of abstraction. When designing software, you think of ways to abstract away

certain details. This is typically done by grouping certain details together based on shared characteristics or purposes. These shared characteristics or purposes are then a way to generalize your design. For example, a programmer may think about a soccer ball, tennis ball, baseball, and softball as distinct types of items. A software designer would generalize these items by classifying all of them as a type of ball that share certain characteristics (e.g., they are all spheres with a center point and radius) and purposes (e.g., they are all struck by an object).

As a software designer applies abstraction to a problem, different design elements are identified. Each of these design elements represents a certain level of abstraction. For example, a relatively detailed software design element may represent one or more elements found in the code. In an object-oriented design, methods and classes are directly implemented in code. In a structured design, functions are directly implemented in code. Thus, a class diagram (for object-oriented design) and a hierarchy chart (for structured design) represent software design models that are at a detailed level of abstraction. A software designer will produce more general design elements by combining detailed design elements. For example, a bunch of classes in an object-oriented design may be combined into a package. Or, a bunch of functions in a structured design may be combined into a component. As the designer continues to identify elements that represent more abstract designs, the designer produces a very high-level design often called a software architecture.

It is important to remind ourselves that software design abstractions should be created to represent both the structure and behavior of the software. The examples in the previous paragraph focused only on abstractions that describe software structure.

To illustrate the different thinking done by a programmer versus a software designer versus a software architect, the example from the first paragraph in this section is restated. A *programmer* may think about a soccer ball, tennis ball, baseball, and softball as distinct types of items used to play a specific type of game. A *software designer* would generalize these items by classifying all of them as a type of ball that share certain characteristics (e.g., they are all spheres) and purposes (e.g., they are all struck by another object). A *software architect* would abstract away even more details by recognizing that equipment is needed to play certain games. A ball represents one type of equipment needed to play these games.

11.2.2.1 Modeling Abstraction: Software Design Models

As we transition to thinking more about software design and less about program design, we move away from the details of coding. To do this effectively, we will need to introduce and use software design models that allow us to easily represent higher levels of abstraction.

Selected software design models will be introduced in Chaps. 12 and 13. These models will allow us to abstract away some of the details while accurately representing the structure and/or behavior of the software code. Additional software design models will be introduced, as needed, throughout the remaining chapters of this book.

11.2.2.2 Modeling Abstraction: Software Design Patterns

Software design patterns were first documented by Gamma, Helm, Johnson, and Vlissides, often referred to as the *Gang of Four (GoF)*, in their seminal book [11]. Their first paragraph in Chap. 1 Introduction eloquently states the challenges and opportunities for software designers.

> Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get 'right' the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

While all of the patterns this author has seen have been written with object-oriented design and programming in mind, much of the lessons learned and applied to OOD and OOP may be reworded and applied to structured design and programming. Below is the GoF quote, reworded so that it applies to software design regardless of the programming paradigm (e.g., object-oriented, imperative, functional, logical) being used. The *italicized words* identify the changes made to the GoF introductory paragraph.

> Designing software is hard, and designing reusable software is even harder. You must find pertinent *structures and behaviors*, factor these into *design elements* at the right granularity, *define interfaces and hierarchies*, and establish key relationships among *these design elements*. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced software designers will tell you that a reusable and flexible design is difficult if not impossible to get 'right' the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Chapter 27 provides an in-depth introduction to software design patterns. Chapters 28 and 29 apply selected software design patterns to the Address Book Application case study.

11.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as the process of generalizing a concept by removing details that are not needed to properly convey the perspective being emphasized.

Exercises

Discussion Questions

1. A simple measure that may be used to describe simplicity (or complexity) of program code is a count of the source lines of code. Can you think of reasons why counting source lines of code is *not* a good measure for simplicity?
2. Coupling and cohesion are two design concepts that are easy to mistake for each other. Are there examples outside of software design that may be used to better understand the distinction between these two design criteria?
3. Information hiding includes hiding the algorithm being used from other software elements that do not need to know how the processing is being performed. Besides the example used in this chapter—hiding the type of persistence storage being used—what are some other examples of hiding the details of an algorithm from other software elements?
4. Describe a problem statement (i.e., scenario) and have students discuss the use of the six characteristics of a good software design. Are there situations where a balance (i.e., trade-off) needs to be done between two or more of the design characteristics?
5. Use your development of an application that you started in Chap. 3 or 4 for this question. Discuss the six characteristics of a good software design and their use within your current program design and code. Which of these characteristics do you feel your program design adheres to and which do you feel needs to be improved? Explain your answers.
6. Based on the six characteristics of a good software design and your experiences, do you think that a perfect software design can be developed? Explain your answer.

References

1. Wiktionary.org: simplicity, coupling, cohesion, information hiding, performance, and security (2019) In: Wiktionary the free dictionary. Wikimedia Foundation. https://en.wiktionary.org/wiki/Wiktionary:Main_Page. Accessed 09 2019
2. Wikipedia.org: Coupling (computer programming) (2017) In: Wikipedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/Coupling_(computer_programming). Accessed 19 2017
3. Pressman RS (2005) Software engineering: a practitioner's approach, 6th edn. McGraw-Hill, New York
4. Dhama H (1995) Quantitative models of cohesion and coupling in software. J Syst Softw 29
5. Wikipedia.org: Cohesion (computer science) (2017) In: Wikipedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/Cohesion_(computer_science). Accessed 19 2017
6. Bieman JM, Ott LM (1994) Measuring functional cohesion. IEEE Trans Softw Eng 20(8)
7. Pfleeger SL (2001) Software engineering: theory and practice, 2nd edn. Prentice-Hall, Upper Saddle River

8. The Joint Task Force on Computing Curricula, ACM, IEEE Computer Society (2013) Computer science curricula 2013: curriculum guidelines for undergraduate degree programs in computer science
9. Larman C (2002) Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process, 2nd edn. Prentice Hall, Upper Saddle River
10. Software Engineering Institute (2019) Top 10 secure coding practices. Carnegie Mellon University. https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices. Accessed 11 2019
11. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Boston