Chapter 7. Design

Table of Contents

Objectives

At the end of this chapter you will have acquired practical and theoretical knowledge and skills about modern software design. After successfully completing this module you should be able to:

- Describe the importance of abstraction and information hiding.
- Describe how abstraction and information hiding are used to handle changes in the software, and in testing the software.
- Show how information hiding and abstraction relates to the software's architecture.
- Define what a design pattern is.
- Give examples of various design classes.

Introduction

The last few chapters have introduced the concepts of analysis and design, along with various modelling techniques appropriate to these activities. We have also discussed some general topics, such as ensuring that your models are readable, that they are produced iteratively and incrementally to better accommodate change both in their requirements and in your own understanding of these requirements. In this chapter we will be discussing some general design guidelines which you may want to keep in mind when designing software systems. We will break these down into general areas such as *abstraction* and *architecture*, each dealt with in their own section.

Abstraction

Abstraction is the activity of reducing the information in a problem to only that information which is important to us. When we perform analysis and design we use many levels of abstraction in order to make both the problem we are solving and the software we are developing understandable and meaningful. At *higher abstraction levels*, the problem and the software are described in less detail and more broadly. At *lower abstraction levels*, more details are given, and the software begins to take on a more concrete form.

Ultimately, the software itself is the lowest level of abstraction that a software engineer will use. It has the highest level of detail.

When performing the act of abstraction, software engineers attempt to create *procedural* and *data* abstractions. **Procedural abstraction** is abstraction applied to functions, methods, and procedures in general. For example, a function's name is an abstraction of the operations which the function performs. The name, in other words, stands for those sequence of operations, and in our designs can be used in its stead.

Data abstraction is abstraction applied to the data discussed in our software and system designs. When we combine data to form an object or class, or combine objects to form a collection, we are performing data abstraction.

Clearly, the software analysts and designers need to choose an appropriate level of abstraction to operate at.

Architecture

Architecture refers to the software being developed. Specifically, the **software's architecture** is the structure of the software: the components that make up the software and how these components are brought together.

Through abstraction, the software's architecture can also be represented at different levels of detail. At lower levels of abstraction, software architecture is concerned with classes, objects, and their interrelationships; at higher levels, the components we consider are the systems and subsystems which make up the software.

Note

The software's architecture is one of the most important aspects of a software system. As such, it should be considered early in the process of analysis and design. A poorly developed architecture is a high-risk factor in software development.

A number of models can be used to represent the software's architecture:

- *Structural models* represent the program's components (such as class diagrams).
- *Framework models* attempts to discover portions of the architecture that can be reused in similar programs. These reusable portions are called frameworks.
- *Dynamic models* show how the architecture may change over time, especially when the software needs to react to external events (such as sequence diagrams).
- *Process models* show the business / technical processes that the software captures (such as data-flow diagrams).
- *Functional models* shows the software's functional hierarchy.

Patterns

Patterns are known solutions to particular problems. Being known solutions, they can be useful guides to generating designs of new systems. Importantly, a design pattern should allow a software engineer to determine if the solution it specifies is suitable to solving a particular problem.

Design patterns are important enough to warrant their own chapter. We will discuss them in far more detail in ???.

Modularity

When developing software, the software is broken into smaller and smaller components, into packages of classes, then into the classes themselves, into the base data-types that make up these classes, into the functions that they call, and so on.

This ability to divide a software system into discrete portions is called **modularity**, which is an important component of abstraction and architectural design. Having modular software allows it to be more easily comprehended by the developers and our customers. However, modularity has a drawback: while increasing modularity can increase our understanding of the software, after a certain point the software will consist of enough modules that we will again have a problem seeing how they all interact.

As with any abstraction tool, it is important to choose the right level of modularity (the right level of abstraction) for the software.

Modularised software is easier to develop and to test; it can more easily accommodate change, since change should be restricted to only a small number of modules.

Note

When we use the term *module*, we are referring to any division in the software, such as a package or a class.

Information hiding

Information hiding is an important aspect of modularity, and if you recall the definition of abstraction (*reducing information content to only what is important*), information hiding is an important aspect to the abstraction of software.

Specifically, consider that the final software system *is* the lowest level of abstraction. All of the software's design details are present at this level. Information hiding allows us to hide information unnecessary to a particular level of abstraction within the final software system, allowing for software engineers to better understand, develop and maintain the software.

We use software modules to implement information hiding: the information contained in the modules should be hidden from those the rest of the software system outside of the module, and access to this hidden information should be carefully controlled. This allows us to maintain a higher level of abstraction in our software, making our software more comprehensible.

If information hiding is done well, changes made to the hidden portions of a module should not affect anything outside of the module. This allows the software engineers to more readily manage change (including changes in the requirements).

Functional independence

Functional independence occurs where modules (such as a package or class) address a specific and constrained range of functionality. The modules provide interfaces *only* to this functionality. By constraining their functionality, the modules require the help of fewer other modules to carry out their functionality.

The functional independence of a module can be judged using two concepts: *cohesion* and *coupling*: **cohesion** is the degree to which a module performs only one function. **coupling** is the degree to which a module requires other modules to perform its function.

Note

The goal of functional independence is to maximise cohesion while minimising coupling.

Having many functionally independent modules helps a software system be resilient to change: because functionally independent modules rely on fewer other modules, there is less chance of changes to these modules spreading to those which are functionally independent.

Functional independence makes modules easier to develop and test. Changes made to how they perform their function are less likely to affect the software as a whole.

Functional independence is one of the goals of using information hiding and modularity. Consider this: there can be no good information hiding if the software has not been broken into modules. If the software has not been broken into modules, there can not ever be functionally independent modules. If no information is hid from other modules of the software, if every module always depended on all the others to perform its function, any change made to the software will always result in changes having to be made elsewhere in the software in order to handle these changes.

Stepwise refinement

Stepwise refinement is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and, incrementally refining the software through each level of abstraction, providing more detail at each increment. At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed.

At the early steps of the refinement process the software engineer does not necessarily know how the software will perform what it needs to do. This is determined at each successive refinement step, as the design and the software is elaborated upon.

Refinement can be seen as the compliment of abstraction. Abstraction is concerned with hiding lower levels of detail; it moves from lower to higher levels. Refinement is the movement from higher levels of detail to lower levels. Both concepts are necessary in developing software.

Refactoring

Refactoring is the activity of changing the software's internal structure without changing its external behaviour. It is specifically concerned with *improving* the software's internal structure.

Refactoring provides higher quality software, and eases the work of the software engineer. During refactoring, the software is examined for:

- *Redundant code*: portions of the software that perform the same function should be merged. Having the functionality repeated in multiple areas makes the software harder to maintain.
- *Unused design elements*: if portions of the software are not being used, and removing it will not change the software's behaviour, those portions should not be in the software.
- *Poorly constructed data structures*: these should be modified to improve, for instance, information hiding and functional independence.

As with all of the design concepts we have discussed, refactoring should be done to make code easier to understand, easier to develop and test, and easier to change. A good indication that you need to refactor a particular software application is when it has become difficult to either add new functionality to it, or to fix a bug in it.

Note

Refactoring is *not* concerned with fixing bugs or adding functionality: it is concerned with improving design concepts in the software. This, however, should make it easier to find and fix software bugs. Importantly, refactoring does *not* include changing the running time of an application: as with bug fixing or adding functionality, a change to the running time is clearly changing the software's behaviour, and so not in the purview of refactoring.

Design classes

As the design progresses, classes can often be fit into various roles. Five common roles are presented below.

- **User interface classes.** Instances of these classes are used to provide all the interaction between the user and the software. Often, interaction with the software occurs through the use of a metaphor (think of the desktop metaphor, or the drawing board metaphor in *Computer Aided Design* software), and user interface classes may represent elements of this metaphor.
- **Domain classes.** These are the classes that are used to implement some specific portion of the problem domain that the software is attempting to solve. Classes that represent books and catalogues are examples of domain classes for software used in a library.
- **Process classes.** Are lower-level domain classes used to implement the software.
- **Persistent classes.** Are classes that represent data stores, and data that will persist even when the program is not executing. They are useful for hiding the details of obtaining specific data from databases and files.
- **System classes.** Provide the functionality the software requires to operate and communicate with the environment in which it will be functioning.

When used in software design, these classes can be represented using appropriately named stereotypes. For example, user interface classes can be represented in class diagrams using the «user interface» stereotype, persistence classes with «persistent», and system classes with «system». However, the stereotypes should only be used if they will add useful meaning to the model. If knowing that a particular class will be used in the user interface is not useful, do not add the stereotype.

These classes should have the following properties:

- The class should do all that its name implies, and do only what its name implies.
- The class and each of its method should provide only one way to do the same thing.
- The class should be functionally independent. That is, it should have high cohesion and low coupling.

Review

Questions

Review Question 1

Why is the importance of *abstraction*?

Discussion of this question can be found at the end of this chapter.

Review Question 2

What is the main technique for implementing information hiding?

Discussion of this question can be found at the end of this chapter.

Review Question 3

Complete the following:

Refactoring is the activity of changing the software's [_____] without changing its [_____]. It is specifically concerned with improving the software's [_____].

During refactoring, software is examined for three things, namely:

1. [_____]
2. [_____]
3. [_____]

Discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Abstraction allows us to focus on the relevant portions of a problem and our design. It allows us to simplify our representation of how actions are carried out (procedures) and how data is represented (such as by using classes and objects).

Abstraction also allows us to represent our software at various levels of detail.

Discussion of Review Question 2

Software modules are the main tool used to implement information hiding. They segregate a portion of the information contained within a software system from the rest of the system, and control access to this information. Examples of software modules include C++ and C# namespaces, Java packages, and classes and objects.

Discussion of Review Question 3

Refactoring is the activity of changing the software's **internal structure** without changing its **external behaviour**. It is specifically concerned with improving the software's **internal structure**.

During refactoring, software is examined for three things, namely:

1. Redundant code
2. Unused design elements
3. Poorly constructed data structure