Introduction to Design Patterns

27

The objective of this chapter is to introduce software design patterns.

27.1 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 understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_27

27.2 Concepts and Context

Software design patterns became widely discussed in 1995 when a book titled *Design Patterns, Elements of Reusable Object-Oriented Software* was published [1]. The authors of this book were inspired by the importance of patterns in other disciplines. Most notably, a book by Christopher Alexander et al., which documented many patterns for the building architecture discipline [2].

A software design pattern is a description of a problem which occurs frequently in various contexts. It describes the *core of a solution* that can be implemented in many different ways while being true to the intent of the design pattern. The literature [1,3,4] promotes design patterns as a way to make your design more flexible. This flexibility would lead to an overall design that is easier to adapt as requirements change over time.

Software design patterns will be discussed using the perspectives presented in three books, all of which describe patterns using object-oriented design models.

1. The *Gang of Four* (GoF) book published in 1995, which started the discussion of using patterns to describe software designs.
2. A book by Craig Larman [3] published in 2002. As you may recall, a few of Larman's patterns were discussed in Chap. 11.
3. A book by Eduardo Fernandez-Buglioni [4] published in 2013. This book describes security patterns.

27.2.1 GoF Design Patterns

As mentioned, this book started the discussion on software design patterns. This book describes patterns using object-oriented models and code. The GoF [1] organized their patterns by their purpose. They identified three broad categories for their design patterns.

Creational: The purpose of these patterns is to create objects. They identified five creational patterns.

Structural: The purpose of these patterns is to deal with the composition of classes or objects. These patterns explain how classes or objects are related to each other. The GoF identified seven structural patterns.

Behavioral: The purpose of these patterns is to describe ways in which classes or objects interact with each other. The GoF identified 11 behavioral patterns.

27.2.1.1 Examples of GoF Software Design Patterns

A few of the 23 patterns described by the GoF include the following.

- Sample Creational Design Patterns

Singleton: Allows only one object instance to be created for the class. This design has a public method which provides access to this one object.

Abstract Factory: Provides an interface, using an abstract class, for creating object instances where objects are related without needing to specify their concrete classes. The abstract factory class contains method signatures for each type of object to be created. A concrete factory class will subclass the abstract factory class to implement the abstract methods. This results in instantiating objects using the appropriate concrete classes. Generally, an application using an abstract factory will only instantiate one of the concrete factory classes.

Factory Method: Provides an interface for creating an object where subclasses decide which class to instantiate. The interface contains a factory method that may include a parameter to indicate the type of object to create.

- Sample Structural Design Patterns

Facade: Provides a unified interface for a bunch of interfaces within a subsystem.

- Sample Behavioral Design Patterns

Command: Encapsulates a request as an object.

Iterator: Allows sequential access to elements (i.e., objects) within a container (or aggregate) object.

27.2.1.2 What Does a GoF Software Design Pattern Look Like?

The GoF template for a design pattern has the following sections.

- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Three GoF design patterns are briefly described below to illustrate how the above template is used in [1]. The first pattern is the creational pattern named Singleton. Table 27.1 provides a summary description for the template sections, while the class diagram in Fig. 27.1 and the Java code in Listing 27.1 provide implementation details.

Table 27.1 GoF example: Singleton pattern [1]

Section	Description
Intent	Ensure a class only has one instance, and provide a global point of access to it
Motivation	It is important that some classes only have exactly one instance. For example, only one file system or window manager provided by an OS. This pattern makes the class itself responsible for keeping track of its sole instance
Applicability	Use a Singleton pattern when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point; when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code
Structure	See the class diagram in Fig. 27.1
Participants	A Singleton class defines a protected constructor and a public static getInstance() operation. It also has a private attribute of type Singleton
Collaborations	Each client class will access the instance through the public getInstance() method
Consequences	Use of this design pattern results in controlled access to a sole instance; a reduced name space; a refinement of operations and representation; the possibility of extending this pattern to support a fixed number of instances besides one, i.e., modify the Singleton pattern if more than one instance becomes necessary
Implementation	See the class diagram in Fig. 27.1
Sample code	See Listing 27.1
Known uses	Access to a persistent data store may involve a single connection, e.g., to a database server
Related patterns	Many patterns can be implemented using the Singleton pattern. For example, Abstract Factory, Builder, and Prototype

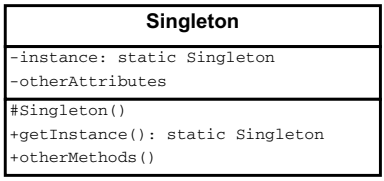

Fig. 27.1 GoF Singleton class diagram

Listing 27.1 GoF Singleton Sample Code

```
public class Singleton
{
  private static Singleton instance = null;
  //class may have other attributes!

  protected Singleton()
```

```
  {
    //initialize other attributes, if needed
  }
  public static Singleton getInstance()
  {
    if (instance == null)
      instance = new Singleton();
    return instance;
  }
  //class may have other operations!
}
```

The second GoF pattern is the structural pattern named Facade. Table 27.2 provides a summary description for the template sections, while the package diagram in Fig. 27.2 and the Java code in Listing 27.2 provide implementation details. The package diagram shows a Facade class providing a common interface to four classes (named One, Two, Three, and Four). The unnamed classes in the package diagram represent other classes likely to be part of the subsystem. The Java code in Listing 27.2 shows a Compiler class used to compile a source code file while hiding the details of the compiler subsystem.

Listing 27.2 GoF Facade Sample Code

```
public class Compiler //Facade to a compiler subsystem
{
    private Scanner scan;
    private Parser parser; //compiler subsystem class
    private ProgramNodeBuilder builder; //compiler subsystem class

    public Compiler()
    {
        builder = new ProgramNodeBuilder(...);
        parser = new Parser(...);
    }
    public void compile(File codeFile, BytecodeStream output)
    {
        scan = new Scanner(codeFile);
        parser.parse(scan, builder);
        CodeGenerator generator = new CodeGenerator(builder);
        generator.generate(output);
    }
}
```

The third pattern is the behavioral pattern named Iterator. Table 27.3 provides a summary description for the template sections, while the class diagram in Fig. 27.3 shows an Iterator interface being used to represent the common operations of an iterator, regardless of the underlying data structure (i.e., ConcreteAggregate) containing the data.

Table 27.2 GoF example: Facade pattern [1]

Section	Description
Intent	Provide a unified interface to a set of interfaces in a subsystem. A Facade defines a higher level interface that makes the subsystem easier to use
Motivation	Structuring a system into subsystems helps reduce complexity. We want to minimize communication and dependencies between subsystems
Applicability	Use the Facade pattern when you want to provide a simple interface to a complex subsystem; there are many dependencies between clients and the subsystem. A facade will decouple the clients from the subsystem
Structure	See the package diagram in Fig. 27.2
Participants	A Facade knows which subsystem classes are responsible for a request and delegates client requests to appropriate subsystem objects. Figure 27.2 shows four subsystem classes—One, Two, Three, Four—that are known by the Facade class. A request from another subsystem may be forwarded by the Facade to any of these four classes. Each subsystem class implements specific subsystem functionality; handling work assigned by the Facade object. These subsystem classes have no knowledge of the Facade class, i.e., they have no references to it
Collaborations	Each client communicates with the subsystem by sending requests to the Facade. The Facade forwards each request to an appropriate subsystem object. The clients using the Facade do not have access to subsystem objects
Consequences	Use of this design pattern shields clients from subsystem complexity and promotes weak coupling. However, this pattern does not prevent applications from directly using subsystem objects
Implementation	See the package diagram in Fig. 27.2
Sample code	See Listing 27.2
Known uses	When access to a complex subsystem needs to be simplified
Related patterns	Facade objects are often Singletons. An Abstract Factory may be used by Facade to provide an interface for creating subsystem objects. Finally, the Mediator design pattern is similar to Facade. However, the Mediator's purpose is to abstract arbitrary communication between colleague objects; it centralizes functionality that does not belong in any of the colleague classes. Another difference: a Mediator's colleagues are aware of and communicate with the Mediator object (instead of communicating with each other directly)

27.2.2 Larman Design Patterns

Like the GoF book, the Larman book describes patterns using object-oriented models and code. Larman introduces nine patterns characterized as general responsibility

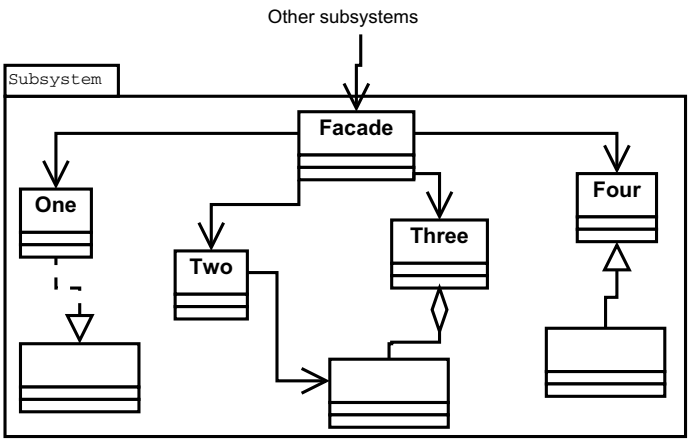

Fig. 27.2 GoF facade package diagram

assignment software patterns (GRASP) [3]. Two of his GRASP patterns—*Low Coupling* and *High Cohesion*—were described in Chap. 11.

27.2.2.1 Examples of Larman Software Design Patterns

Three of the nine patterns described by Larman include the following.

Creator: This is similar to the GoF Factory patterns. It describes a design solution where a class is responsible for creating object instances of another class.

Low Coupling: This pattern assigns responsibility in a way that decreases coupling between classes. See Table 11.2 for details on this GRASP pattern.

High Cohesion: This pattern assigns responsibility in a way that increases cohesion within a class. See Table 11.3 for details on this GRASP pattern.

27.2.2.2 What Does a Larman Software Design Pattern Look Like?

Larman uses the following template to describe his GRASP patterns.

- Problem
- Solution
- Discussion
- Contraindications
- Benefits
- Related patterns or principles

Larman's Creator pattern in GRASP is summarized in Table 27.4.

Table 27.3 GoF example: Iterator pattern [1]

Section	Description
Intent	Provide a way to access elements of an aggregate object sequentially without exposing its underlying representation
Motivation	An aggregate object should give you a way to access its elements without exposing its internal structure. For example, you may want to traverse the aggregate object in different ways, but you do not want to make the interface larger by adding operations for different traversals
Applicability	Use an Iterator pattern to access an aggregate object's contents without exposing its internal representation; support multiple traversals of aggregate objects; and to provide a uniform interface for traversing different aggregate structures
Structure	See the class diagram in Fig. 27.3
Participants	The class diagram in Fig. 27.3 shows two interfaces and two concrete classes. Iterator is an interface for accessing and traversing elements, while ConcreteIterator is a class that implements the Iterator interface. This class keeps track of the current position in traversal of the ConcreteAggregate. Aggregate is an interface for creating an Iterator object, while ConcreteAggregate is a class that implements the Iterator method to return an instance of the proper ConcreteIterator
Collaborations	A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
Consequences	This design pattern supports variations in the traversal of an aggregate. It simplifies the aggregate interface and allows more than one traversal to be pending on an aggregate
Implementation	See the class diagram in Fig. 27.3
Sample code	Not provided
Known uses	Very common in object-oriented systems; most collections (i.e., data structures) provide one or more iterators. This pattern is implemented in the Java API; see the Interator interface and the Collection, List, Map, and Set interfaces for different types of aggregates
Related patterns	Iterators are often applied to recursive structures

27.2.3 Fernandez Design Patterns

The Fernandez book describes patterns for building more secure software. Instead of categorizing patterns like the GoF, Fernandez uses a matrix to classify his patterns [4]. The rows in his matrix represent different levels of architecture. These levels are application, operating system, distribution, transport, and network. The distribution, transport, and network levels refer to the application, transport, and network layers in the Internet Protocol Stack. The columns in the matrix are life cycle phases and include domain analysis and design. The column for the design life cycle phase is further split into purposes, e.g., filtering, access control, and authentication. The

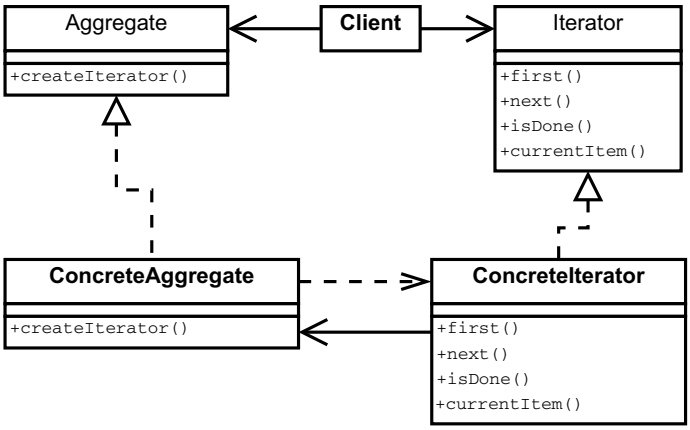

Fig. 27.3 GoF iterator class diagram

Table 27.4 Summary of GRASP Creator [3]

Solution	Assign class B the responsibility to create one or more instances of class A if any of the following are true • B aggregates, contains, or records A objects • B closely uses A objects • B has the data needed to initialize A objects
Problem	As we know, creating objects is a common activity in object-oriented systems. But who should be responsible for creating a new instance of some class?
Discussion	Look for class that needs a connection to created object. This connection may be reflected in a class diagram using any of these relationship types: • Aggregate aggregates Part • Container contains Content • Recorder records Recorded
Contraindications	When creation involves significant complexity, use the Factory pattern
Benefits	Lowers coupling, which implies lower maintenance costs
Related patterns or principles	Low Coupling, Factory, and Whole-Part

security patterns described by Fernandez are described using object-oriented design models.

27.2.3.1 Examples of Fernandez Software Design Patterns

Four of the patterns described by Fernandez include the following.

Table 27.5 Summary of symmetric encryption design pattern [4]

Example	Alice sends sensitive data to Bob. Evan can intercept this data and read the sensitive data
Context	Applications exchange sensitive information over insecure channels
Problem	Sensitive data may be read by unauthorized users while in transit (or at rest)
Solution	The sender transforms plaintext data into ciphertext using a secret key and then transmits the ciphertext over an insecure channel. The receiver transforms the ciphertext into plaintext data using the same secret key
Implementation	Both sender and receiver need to agree on the cryptography algorithm and secret key
Example resolved	Alice encrypts sensitive data then sends ciphertext to Bob. Evan can still intercept this data, but cannot read the sensitive data
Consequences	The key needs to be secret and must be shared between sender and receiver in secure manner. Selection of cryptographic algorithm
Known uses	The following cryptographic algorithms use symmetric encryption: GNuPG, OpenSSL, Java Cryptographic Extension, .NET framework, XML encryption, and Pretty Good Encryption (PGP)
See also	Other security patterns related to this include: secure channel communication pattern, asymmetric encryption, and patterns for key management

Symmetric Encryption: Describes the use of encryption to make a message unreadable unless you have the key. The same key is used to encrypt and decrypt the message.

Asymmetric Encryption: Describes the use of encryption to make a message unreadable unless you have the key. A public key is used to encrypt the message while a private key is used to decrypt the message.

Digital Signature with Hashing: Describes a way to allow the sender of a message to prove the message originated from them and not someone else. Also describes how the receiver of a message can verify the integrity of the message, i.e., that it has not been altered during transmission.

Secure Model–View–Controller: Describes Model–View–Controller components as tiers (or layers) of a system. Each tier enforces security applicable to the tier. Fernandez also describes a Secure Three-Tier Architecture pattern. This describes the three tiers using the terms presentation (i.e., view), business logic (i.e., controller), and data storage (i.e., model).

27.2.3.2 What Does a Fernandez Software Design Pattern Look Like?

Fernandez uses the following template to describe his security patterns.

Table 27.6 Summary of asymmetric encryption design pattern [4]

Example	Alice needs to send sensitive data to Bob, but they do not share a secret key. Evan can intercept this data and read the sensitive data
Context	Applications exchange sensitive information over insecure channels
Problem	Sensitive data may be read by unauthorized users while in transit (or at rest)
Solution	The sender transforms plaintext data into ciphertext using receiver's public key and then transmits the ciphertext over an insecure channel. The receiver transforms ciphertext into plaintext data using their private key
Implementation	Both sender and receiver need to agree on the cryptography algorithm, e.g., RSA
Example resolved	Alice looks up Bob's public key and uses it to encrypt sensitive data. Alice then sends the ciphertext to Bob. Evan can still intercept this data, but cannot read the sensitive data. Bob receives the ciphertext and uses his private key to decrypt the data
Consequences	Anyone can look up someone's public key. Selection of cryptographic algorithm and key length impacts performance and level of security
Known uses	The following cryptographic algorithms use symmetric encryption: GNuPG, Java Cryptographic Extension, .NET framework, XML encryption, and Pretty Good Encryption (PGP)
See also	Other security patterns related to this include secure channel communication pattern

- Example
- Context
- Problem
- Solution
- Implementation
- Example resolved
- Consequences
- Known uses
- See also

The four design patterns listed above are described in Tables 27.5 (Symmetric Encryption), 27.6 (Asymmetric Encryption), 27.7 (Digital Signature with Hashing), and 27.8 (Secure Model–View–Controller).

27.2.4 Summary of Design Pattern Templates

Table 27.9 shows a summary of the software design templates used by the three software design patterns books referenced in this chapter. As this table illustrates,

Table 27.7 Summary of digital signature with Hashing design pattern [4]

Example	Alice wants to send nonsensitive data to Bob and Bob wants to make sure the data came from Alice. Evan can intercept this data and modify it
Context	Applications exchange information over insecure channels and may need to confirm integrity and origin of the data
Problem	Need to authenticate the origin of the message (data)
Solution	The sender computes a message digest on the plaintext data using a hash function, transform the plaintext data into ciphertext using sender's private key, and then sends both the message digest and ciphertext. The receiver decrypts the ciphertext using the sender's public key, computes the message digest on decrypted ciphertext, and then compares the computed digest with digest received from sender
Implementation	Both sender and receiver need to agree on the cryptographic hash function (e.g., SHA-256) and cryptographic asymmetric algorithm (e.g., RSA)
Example resolved	Alice now uses an asymmetric algorithm and a hash function to send nonsensitive data to Bob. Bob verifies that his computed digest matches what Alice sent him. Evan can intercept this data, but cannot decrypt the data or use the hash digest
Consequences	Sender cannot deny that they sent the message (assuming their private key is only known by them)
Known uses	The following cryptographic algorithms use symmetric encryption: GNuPG, Java Cryptographic Extension, .NET framework, and XML signature
See also	Other security patterns related to this include secure channel communication pattern

there is no consensus regarding how software design patterns are described. Even so, design patterns offer a clear benefit when consistently used within an organization. First, it allows software developers to use a common set of terms to describe aspects of their design. Second, software design patterns are typically developed by experienced professionals, making them useful to software development teams with a range of experiences. Finally, design patterns allow a development team to quickly discuss and assess the use of different design approaches.

27.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand the ways in which a software design pattern is described.
- You understand the benefits of using software design patterns.

Table 27.8 Summary of secure Model–View–Controller pattern [4]

Example	The address book case study
Context	Applicable to stand-alone and distributed software systems in homogeneous or heterogeneous environments
Problem	Need to secure all tiers of a system, since having an insecure tier/layer invites attacks. Should provide services that are available through mostly transparent security features to address the confidentiality, integrity, and availability (CIA) of the system. For some systems, recording actions performed by each user (i.e., supporting non-repudiation) would also be an important consideration to address the risk of an attack from a legitimate user
Solution	Apply an appropriate security service to each layer/tier, e.g., use encryption on data sent between the layers. Design the human–computer interaction (view component) to require authentication and authorization of users. Design into the business layer (controller component) a unified access control model. Design into the storage layer (model component) use of encryption for sensitive data
Implementation	Define a global authorization model and select authentication approaches based on needs of the application/system. Also, need to select an encryption approach
Example resolved	See Chaps. 25 (OOD) or 26 (SD) for a security design for the ABA
Consequences	Centralized management of security services, including authorization constraints, authentication information, and logging repositories. Each layer should apply security restrictions, which are (hopefully) transparent to the user. Considerations for availability, non-repudiation, and performance of security services
Known uses	Web services and distributed applications
See also	Other security patterns related to this include secure channel communication pattern

Table 27.9 Summary of design pattern templates

GoF (Original)	Larman (GRASP)	Fernandez (Security)
Intent	Problem	Example
Motivation	Solution	Context
Applicability	Discussion	Problem
Structure	Contraindications	Solution
Participants	Benefits	Implementation
Collaborations	Related patterns/principles	Example resolved
Consequences		Consequences
Implementation		Known uses
Sample code		See also
Known uses		
Related patterns		

Exercises

Discussion Questions

1. Using an existing design solution you've developed, discuss the use of each pattern (listed below) in your design.

   a. Singleton
   b. Facade
   c. Creator

2. Using an existing design solution you've developed, discuss the use of each security design pattern (listed below) in your design.

   a. Symmetric Encryption
   b. Asymmetric Encryption
   c. Digital Signature with Hashing
   d. Secure Three-Tier Architecture

3. Using Table 27.9, discuss the similarities and differences in how design patterns are described by the GoF, Larman, and Fernandez.

Hands-on Exercises

1. Use an existing design solution you've developed, improve your design by adding one or more of the design patterns described in this chapter.

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Boston
2. Alexander C, Ishikawa S, Silverstein M (1977) A pattern language: towns, buildings, construction. Oxford University Press, Oxford
3. Larman C (2002) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall
4. Fernandez EB (2013) Security patterns in practice: designing secure architectures using software patterns, 1st edn. Wiley, New York