Design Patterns

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction 2
- What is a Design Pattern? 3
- Types of Patterns 4

Creational Patterns 5
- Singleton Pattern 6
- Builder Pattern 9
- Factory Pattern 12

Behavioural Patterns 13
- Command Pattern 14
- Memento Pattern 17
- Observer Pattern 19

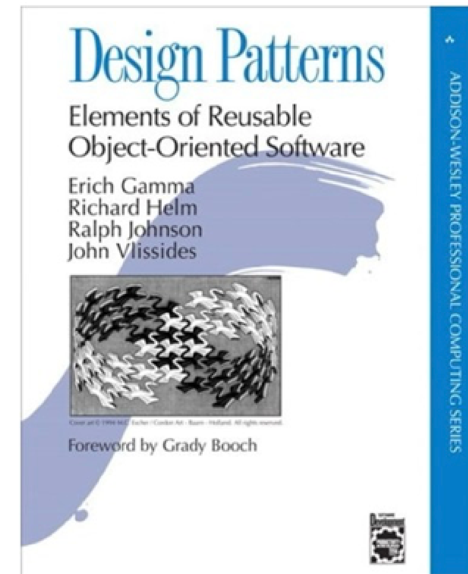
Bibliography 22

Introduction

What is a Design Pattern?

A design pattern is a “a general reusable solution to a commonly-occurring problem within a given context in software design”.

– Gamma et al. 1994. [1]



Benefits

- Proven solutions that address sometimes complex problems.

Cautions

- They must be adapted to suit your application and programming language. They are not off-the-shelf solutions that can be just dropped-in.
- These are explicitly OO solutions. [2], [3]

Types of Patterns

The original set of patterns were subdivided based on the types of problems they addressed.

- Creational Patterns: dynamic creation of objects.
- Behavioral Patterns: identifying communication patterns between objects.
- Structural Patterns: organizing classes to form new structures.

The expectation is that you might encounter a small number of these in any given application.

Some problems are commonly encountered (e.g. decoupling using Observer) and others are rarely used (e.g. Abstract Factory). This is to be expected as conditions change.

We'll present a few patterns that are commonly used in application development. As much as possible, we'll try and present idiomatic Kotlin implementations that you can use in this course. [4]

Creational Patterns

Issues with dynamic object creation.

Singleton Pattern

A singleton is a creational design pattern that lets you ensure that a class has only one instance.

Why is this pattern useful?

- It can be used to control access to a shared resource e.g., a database or a file. Singleton guarantees that there is only a single object instance managing that resource.
- It presents a global access point to that object instance. Just like a global variable, the Singleton pattern lets you access an object from anywhere in the program.

Singleton Pattern

“Standard” implementation:

1. Make the default constructor private, to prevent free object instantiation.
2. Create a static method to instantiate the class, and ensure a single object.
3. Create a static method to return a static reference.

```
public class Singleton {
  private Singleton() {}
  private static instance: Singleton = null

  public static getInstance(): Singleton {
    if (instance == null) {
      instance = Singleton()
    }
    return instance
  }
}

Singleton s = Singleton.getInstance()
```

Singleton Pattern

Kotlin implementation:

Use the Object keyword, which defines a static instance of a class i.e., a singleton. It doesn't need to be explicitly instantiated; this is done 'lazily' as-needed. Methods can be called statically.

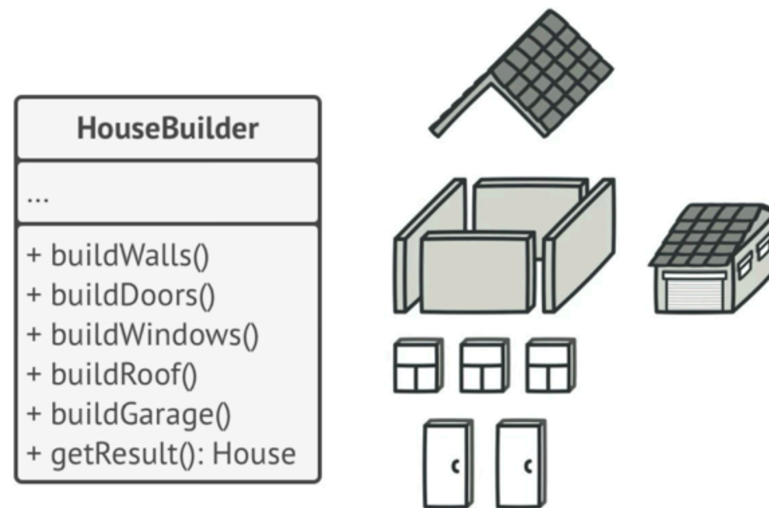
```
object Singleton {
  init {
    println("Singleton accessed")
  }
  fun print() = println("Print called")
}

fun main() {
  Singleton.print()
  // Singleton accessed
  // Print called
}
```

source

Builder Pattern

How do you build complex objects with multiple (optional) initialization steps?

The Builder Pattern lets you construct complex objects step by step. A builder class generates the initial object, and subsequent methods can be called to customize it. The object is inaccessible until a method is called to finalize it. [2]



The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

Builder Pattern

Standard implamantation:

- Static constructor that creates an empty object instance,
- Methods that populate or customize the object,
- A final method that does returns the completed object.

```
val dialog = AlertDialog.Builder(this)
  .setTitle("File Save Error")
  .setText("Error encountered. Continue?")
  .setIcon(ERROR_ICON)
  .setType(YES_NO_BUTTONS)
  .show()
```

Builder Pattern

Kotlin implementation:

- Use named and default arguments instead of setting things manually.
- In practice, these features greatly simplify how we create objects.

```
val dialog = AlertDialog(
  title = “File Save Error”,
  error = Error encountered. Continue?”,
  icon = ERROR_ICON,
  type = YES_NO_BUTTONS
)

dialog.show()
```

Factory Pattern

The Factory Pattern is a Creational pattern that allows you to base the creation of an object until runtime. The Factory, in the name, refers to a software structure whose job it is to create objects dynamically.

An example would be an ordering system for a Pizza Parlor. Customers can order their preferred pizza, and our Factory object will create the appropriate type of pizza based on their selection.[5]

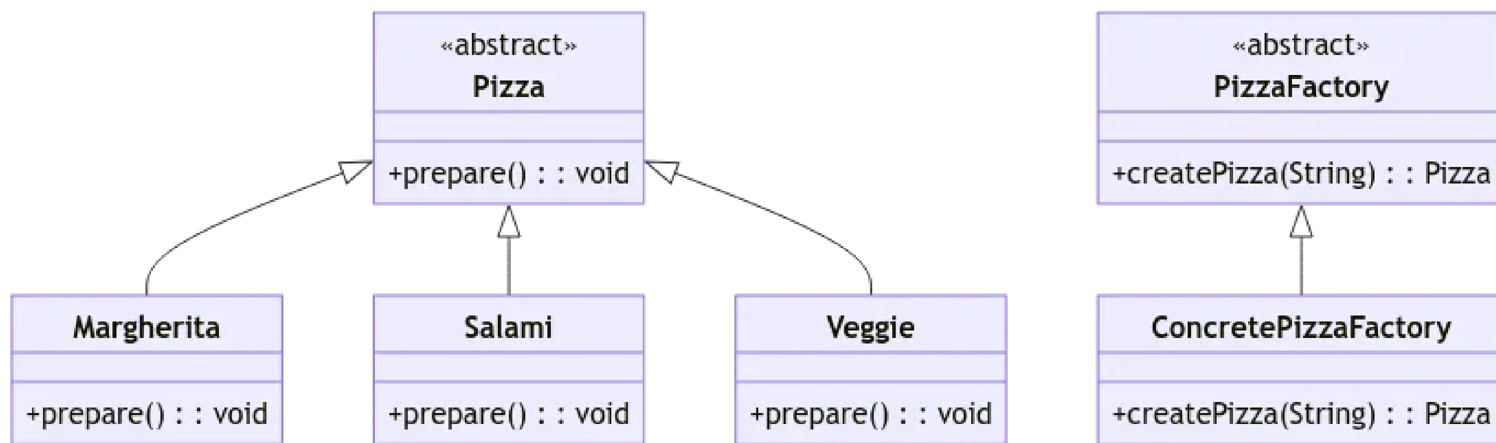


Figure 3: A Pizza Factory determines which type of Pizza to create based on provided arguments.

Behavioural Patterns

Communication between objects.

Command Pattern

Imagine that you are writing a user interface, and you want to support a common action like Save. You might invoke Save from the menu, or a toolbar, or a button. Where do you put the code, without duplicating it?

The command pattern is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request (a command could also be thought of as an action to perform) [2].

Several classes implement the same functionality.

Command Pattern

Mapping arguments (commands) passed in from the command-line to actions within the application:

```
// Entry point
fun main(args: Array<String>) {
  val command = CommandFactory.createFromArgs(args)
  command.execute()
}
// Factory Method
object CommandFactory {
  fun createFromArgs(args: Array<String>): Command =
    if (args.isEmpty()) {
      when (args[0]) {
        "add" -> AddCommand(args)
        "del" -> DelCommand(args)
        "show" -> ShowCommand(args)
        else -> HelpCommand(args)
      }
    }
}
```

Command Pattern

The command-class structure. All commands share a common `Command` interface. Commands can be passed around the system and `executed`.

```
interface Command {
  fun execute()
}
class AddCommand(val args: Array<String>) : Command {
  override fun execute() {
    assert(args.size == 2)
    println("Add: ${args[1]}")
  }
}

class DelCommand(val args: Array<String>) : Command {
  override fun execute() {
    assert(args.size == 2)
    println("Delete: ${args[1]}")
  }
}
```

Memento Pattern

Memento captures an object's state so you can save/restore it later.

This is helpful for object versioning (e.g., tracking changes over time), and is a common pattern to use when implementing undo-redo.

How does it work?

- Before making changes to an object's state, tell it to save itself.
- If needed later, ask it to revert to the previous saved version. [2]

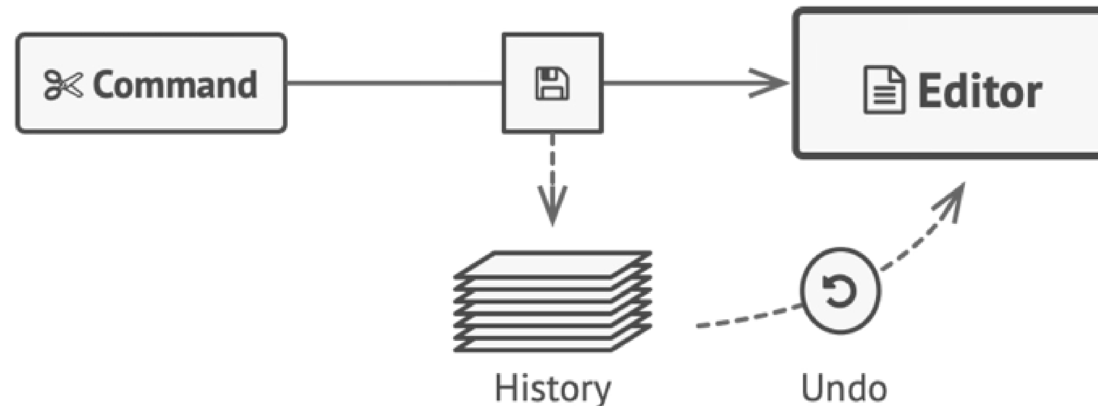


Figure 5: Undo/redo maintains a stack of Mementos, and uses them to undo/redo system actions.

Memento Pattern

```kotlin
class Book(var title: String, var author: String, var year: Int) {
  private data class Memento(val title: String, val author: String)

  private object UndoManager {
    private val mementos = mutableListOf<Memento>()
      fun save(title: String, author: String) {
        mementos.add(Memento(title = title, author = author))
      }
      fun restore() {
        mementos.last()
      }
  }

  fun save() {
    UndoManager.save(title = this.title, author = this.author)
  }

  fun restore() {
    val memento = UndoManager.restore()
    this.title = memento.title
    this.author = memento.author
  }
```

Observer Pattern

Observer is a behavioural design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're monitoring.

The object that has some interesting state is often called the `subject`. Objects that want to track changes to the publisher's state are called `observers`. Subscribers register their interest in the subject, who adds them to an internal subscriber list, and notifies them when state changes occur [2].

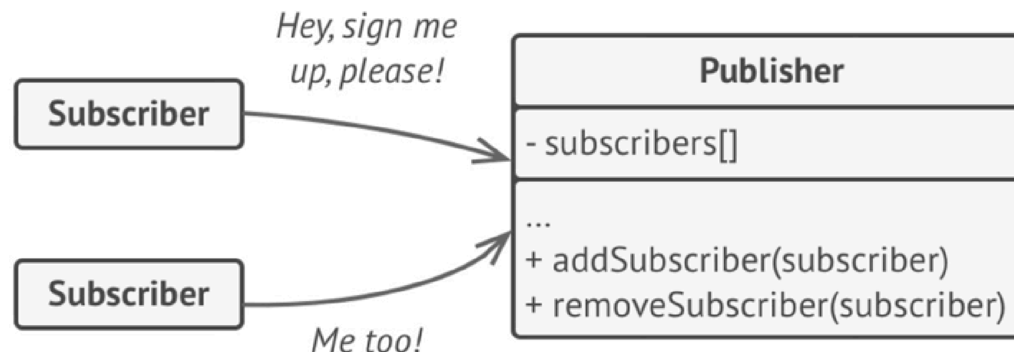


Figure 6: The publisher agrees to notify registered observers when state changes.

Observer Pattern

Kotlin implementation: Publisher classes

```
interface IPublisher {
  val list: ArrayList<ISubscriber>

  fun add(sub: ISubscriber) = list.add(sub)
  fun remove(sub: ISubscriber) = list.remove(sub)

  fun sendUpdateEvent() {
    list.forEach { it.update() }
  }
}

class Newsletter: IPublisher {
  override val list = ArrayList<ISubscriber>()
  var article = ""
    set(value) {
      field = value
      sendUpdateEvent()
    }
  }
```

Observer Pattern

Kotlin implementation: Observer classes

```
interface ISubscriber {
  fun update()
}

class View(target: IPublisher) : ISubscriber {
  init {
    target.add(this)
  }

  override fun update() {
    println(“Updated”)
  }
}

fun main() {
  val publisher = Newsletter()
  val screen = View(publisher)
  publisher.article = “New article” // Updated
}
```

Bibliography

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley Professional, 1994.

[2] A. Shvets, *Dive Into Design Patterns*. Refactoring.Guru, 2022. [Online]. Available: https://refactoring.guru/design-patterns/book

[3] R. C. Martin, *Clean Architecture - A Craftsman's Guide to Software Structure and Design*. Pearson, 2017.

[4] A. Soshin, *Kotlin Design Patterns and Best Practices*. Packt Publishing, 2022.

[5] samibel, "Understanding the Factory Method Design Pattern in Kotlin." [Online]. Available: https://medium.com/@samibel/understanding-the-factory-method-design-pattern-in-kotlin-b17a28d35d14