Design Patterns

CS 346 Application Development

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

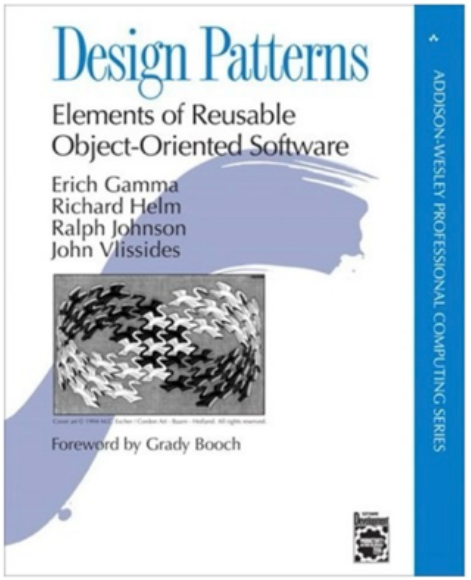
Contents

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

- Solutions to *specific* problems in software design.
- Meant to improve flexibility, robustness of your designs.

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.
- They are often explicitly object-oriented. [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: effective 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 common (e.g. decoupling using `Observer`) and others are rarely used (e.g. `Abstract Factory`).
- Don't assume that you should always use them (or that there's a formal pattern for every situation).

We'll present a few patterns that are commonly used in application development, with Kotlin examples [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.

> **Tip**
>
> Singleton as a means of controlling access to a shared resource has mostly been replaced by Dependency Injection (DI). See that pattern later in these slides.

Singleton Pattern

How to implement `singleton` in Java.

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

  public static getInstance(): Singleton {
    if (instance == null) {
      instance = Singleton(); // static init on access
    }
    return instance;
  }
  public print() { System.out.println("Print called")};
}

Singleton s = Singleton.getInstance();
s.print()
// Print called
```

Singleton Pattern

How to implement `singleton` in Kotlin.

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

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

Tip

The `object` keyword creates a single, static instance of a class i.e., a singleton. It's instantiated 'lazily' as-needed.

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]

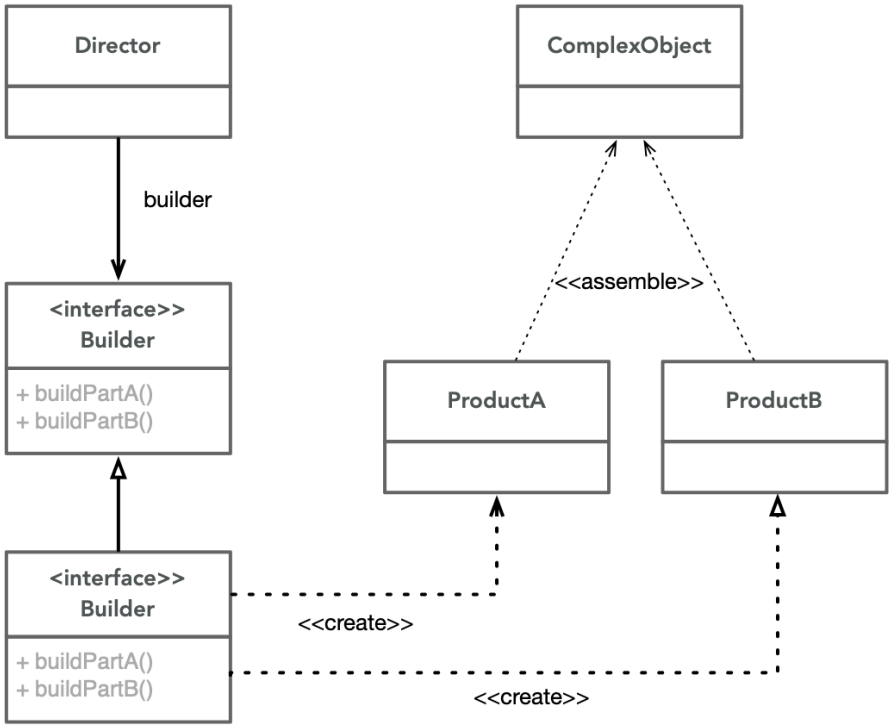


Figure 2: Builder supports staged construction. Image courtesy of Wikipedia

Builder Pattern

Here's an example of the builder pattern:

```
class FoodOrder private constructor(builder: FoodOrder.Builder) {
    val bread: String?
    val condiments: String?
    val meat: String?
    val veggies: String?

    init {
        this.bread = builder.bread
        this.condiments = builder.condiments
        this.meat = builder.meat
        this.veggies = builder.veggies
    }

    class Builder {
        // builder code, next slide
    }
}
```

Builder Pattern

```
class Builder {
  var bread: String? = null
      private set
  var condiments: String? = null
      private set
  var meat: String? = null
      private set
  var veggies: String? = null
      private set

  fun bread(bread: String) = apply { this.bread = bread }
  fun condiments(condiments: String) = apply {
      this.condiments = condiments }
  fun meat(meat: String) = apply { this.meat = meat }
  fun veggies(veggies: String) = apply { this.veggies = veggies }
  fun build() = FoodOrder(this)
}
```

`Builder` has the same fields as the outer class, only accessible through a setter method.

Builder Pattern

Finally to use it, we call the builder and the appropriate methods chained.

The calling method is strictly enforced:

- You cannot set properties directly but only by using Builder methods.
- You cannot instantiate a Food Order, but only return it from the Builder.
- You can put checks in the `build()` method to only allow creating a valid object e.g., check that there's at-least bread defined.

```
fun main() {
    val order = FoodOrder.Builder()
        .bread("Flat Bread")
        .condiments("Mayonnaise")
        .meat("Chicken")
        .veggies("Lettuce, Tomatoes")
        .build()

    println(order.toString())
}
```

Builder Pattern

In Java, `Builder` is used to set required properties before instantiation.

```
Dialog dialog = AlertDialog.Builder(this)
  .setTitle(“File Save Error”)
  .setText(“Error encountered. Continue?”)
  .setIcon(ERROR_ICON)
  .setType(YES_NO_BUTTONS)
  .show();
```

In Kotin, we can use named and default arguments and often avoid using a `Builder`. You’ll sometimes see it when using older APIs.

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

Factory Method Pattern

The Factory Method 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.

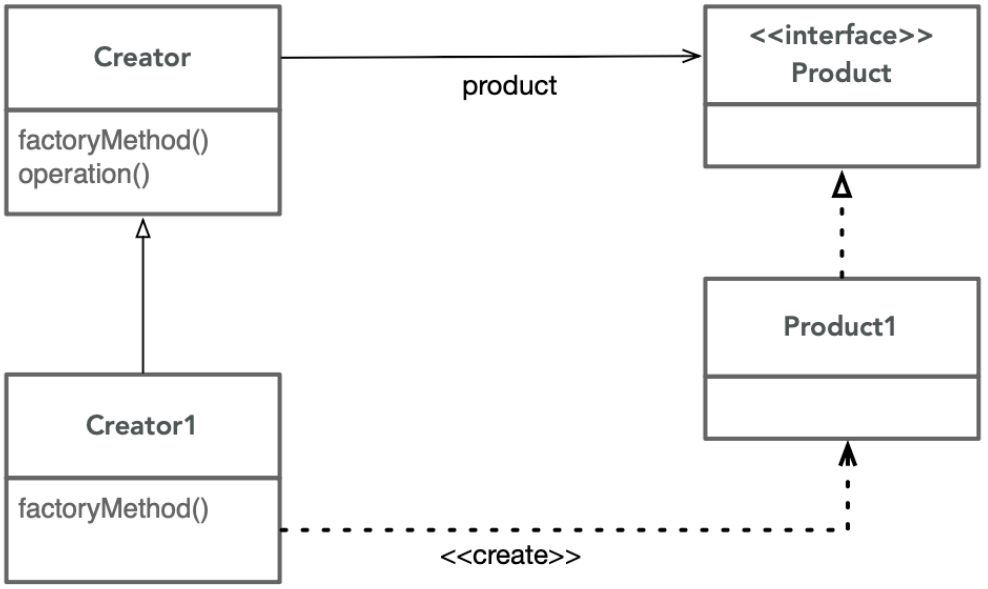


Figure 3: A creator class contains a method that return the correct product, based on conditions.

Factory Method Pattern

An example would be an ordering system for a Pizza Parlor. [5].

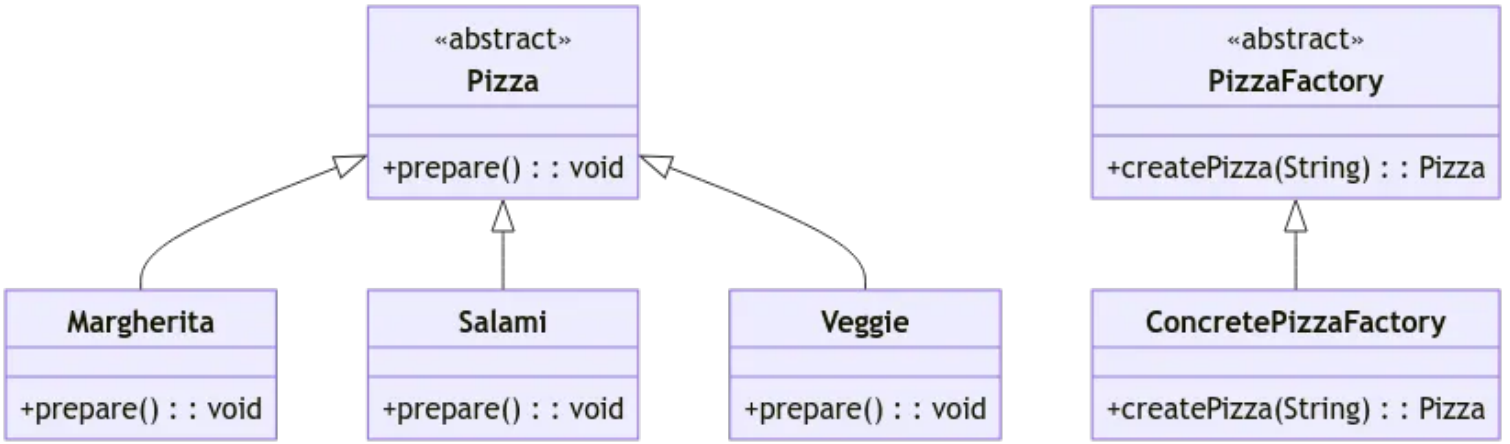


Figure 4: A Pizza Factory determines which type of Pizza to create based on provided arguments.

Our factory method looks like this:

```
class ConcretePizzaFactory : PizzaFactory() {
    override fun createPizza(type: String): Pizza = when (type) {
        "Margherita" → Margherita()
        "Salami" → Salami()
        "Veggie" → Veggie()
    }
}
```

Factory Method Pattern

What's wrong with this? It's inflexible. Adding a new type of pizza requires changes to our `Factory` class and `createPizza()` method.

- Fix this by having separate factories for each type of pizza.
- Adding a new type of pizza merely means adding a new Factory type.

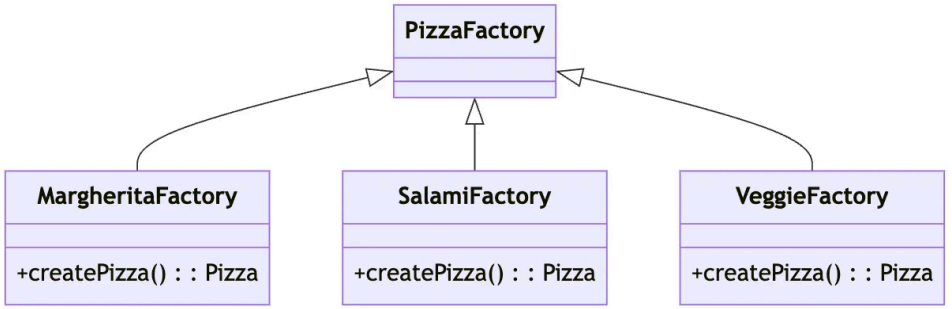


Figure 5: Custom factories are much more flexble.

Here's the simplified code:

```
interface Pizza { fun prepare() }

class Margherita : Pizza {
    override fun prepare() = println("Preparing Margherita Pizza")
}
```

Behavioural Patterns

Communication between objects.

Command Pattern

The Command Pattern is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request [2].

It's often used in user interfaces, where we want to reify actions e.g., you might invoke Save from the menu, or a toolbar, or a button.

- Commands can be manipulated e.g., saved, undone/redone.
- It supports flexibility, since commands can be assesssed at runtime.

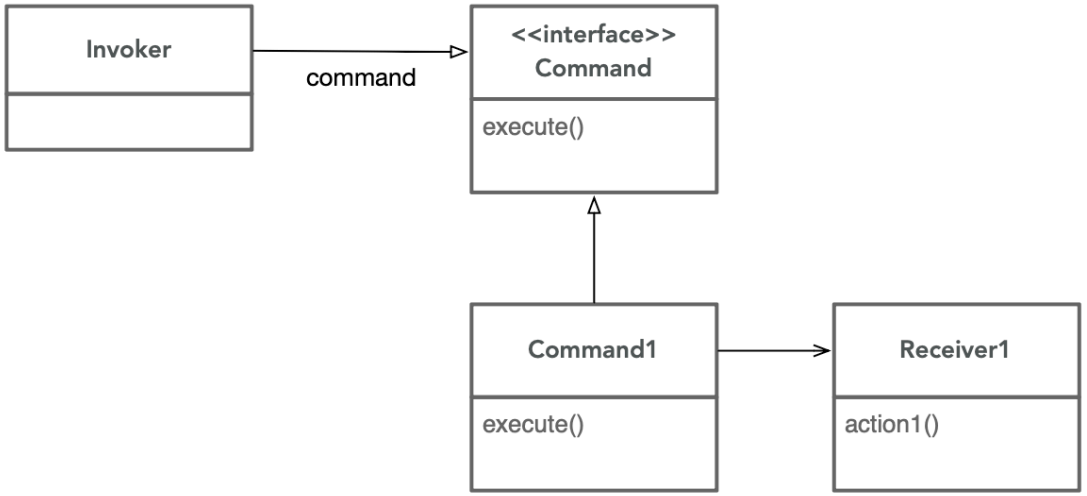


Figure 6: The Command Pattern lets an Invoker decide when and to execute a command, or even what gets executed. Courtesy of Wikipedia.

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

The Memento Pattern 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, create a new Memento.
- If needed later, ask it to revert to the previous Memento. [2]

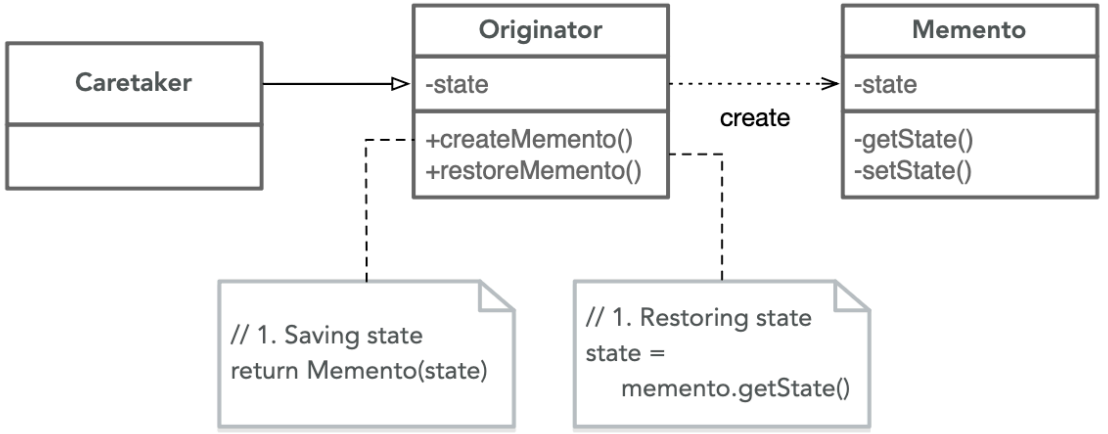


Figure 7: Undo/redo maintains a stack of Mementos, and uses them to undo/redo system actions.

Memento Pattern

```
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
}
```

Strategy Pattern

The Strategy Pattern allows you to define multiple algorithms and choose one dynamically at runtime. Instead of implementing a single algorithm directly, your program uses context to determine which in a family of algorithms to use.

- e.g., selecting a layout algorithm for a screen based on device-size and orientation.

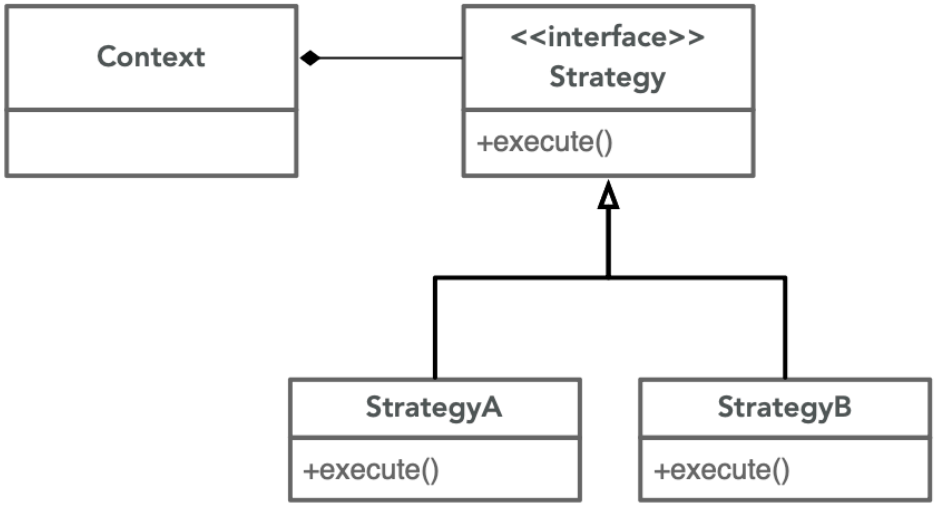


Figure 8: Strategy allows for the selection of an algorithm at runtime. Context can choose StrategyA or StrategyA based on conditions. Image courtesy of Wikipedia

Strategy Pattern

```
interface IBrakes { fun apply() }

class BrakeWithABS : IBrakes {
    fun apply() { println("Brake with ABS applied") }
}

class Brake: IBrakes {
    fun apply() { println("Simple Brake applied") }
}

// client can use the algorithms above interchangeably

abstract class Car(brakes: IBrakes) {
  public void applyBrakes() {
    brakes.apply();
  }
}
```

Observer Pattern

Observer lets you define a subscription mechanism between objects.

The object that you wish to monitor is called the `Subject`. Objects that want to track changes to the Subject's state are called `Observers`. Observers register their interest in the subject, who adds them to an internal subscriber list, and notifies them when state changes occur [2].

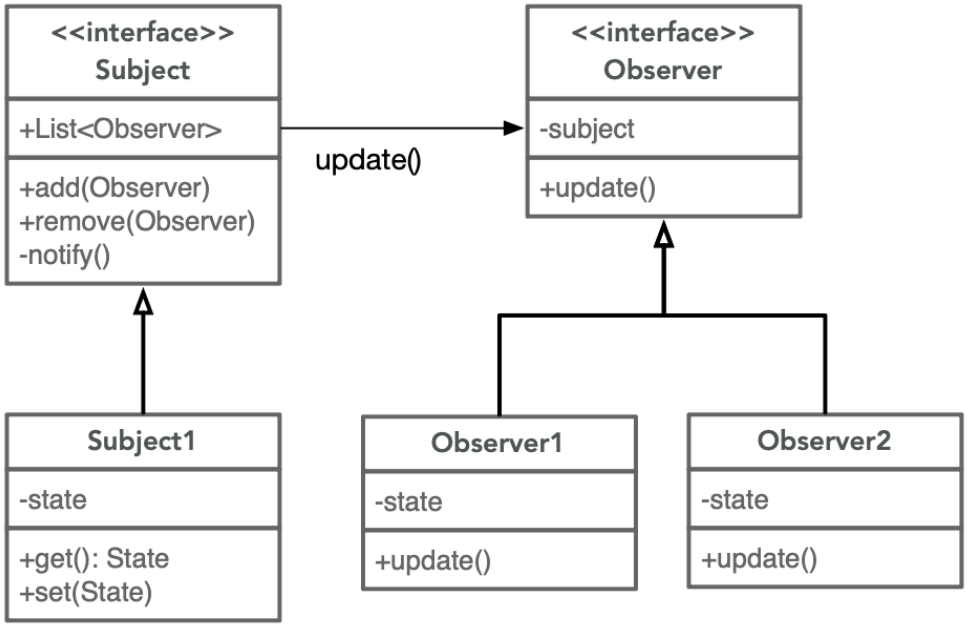


Figure 9: The Subject notifies Observers when appropriate. Image courtesy of Wikipedia

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
}
```

Structural Patterns

Relationships between entities.

Adapter Pattern

The Adapter Pattern solves a common problem of having two different modules communicate when they were not designed to work together. The adapter design pattern solves problems around class reuse, when two classes weren't designed to work together.

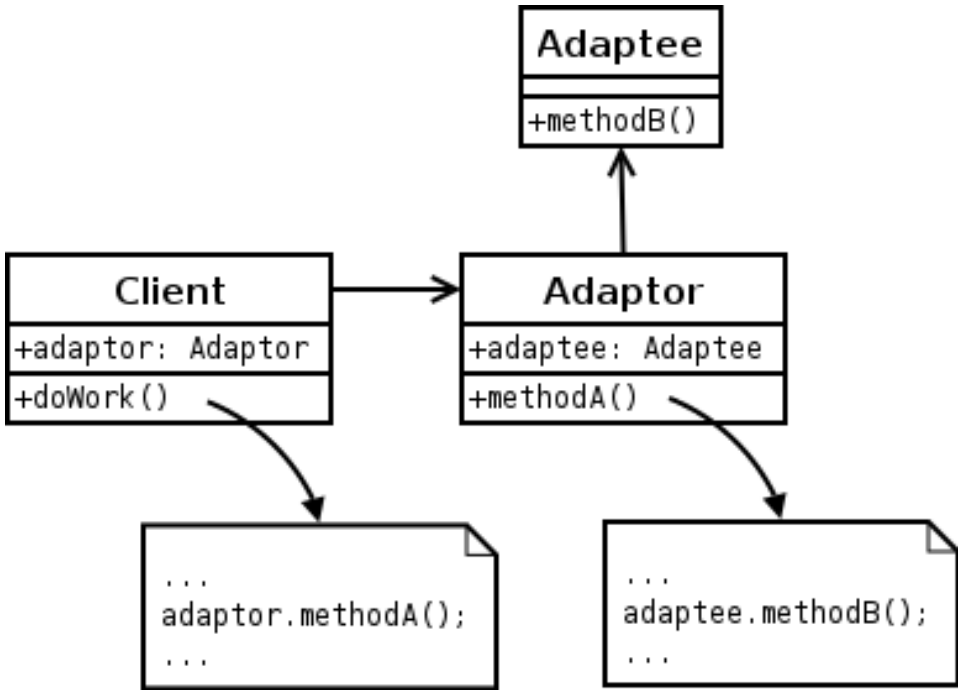


Figure 10: The adapter contains an instance of the target class, and passes through calls from the client to that target. Image courtesy of Wikipedia

Adapter Pattern

Imagine a simple interface for phone chargers (adapted from Wikipedia).
We want to support Apple (lightning) and Android phones (MicroUSB)[1].

```
interface
ILightningPhone {
    fun recharge()
    fun useLightning()
}

interface
IMicroUsbPhone {
    fun recharge()
    fun useMicroUsb()
}
```

```
class Iphone : ILightningPhone {
    var connector = false

    override fun useLightning() {
        connector = true
        println("Lightning connected")
    }

    override fun recharge() {
        if (connector) {
            println("Recharge started")
            println("Recharge finished")
        } else { /* Handle error */
    }
}
```

[1] Yes I'm aware that everyone uses USB-C now, but it's a still a useful example!

Adapter Pattern

Here's how we normally use them. So far, no surprises.

```
fun rechargeMicroUsbPhone(phone: IMicroUsbPhone) {
  phone.useMicroUsb()
  phone.recharge()
}

fun rechargeLightningPhone(phone: ILightningPhone) {
  phone.useLightning()
  phone.recharge()
}

fun main() {
  val android = Android()
  rechargeMicroUsbPhone(android)

  val iPhone = Iphone()
  rechargeLightningPhone(iphone)
}
```

Adapter Pattern

What about an iPhone using a MicroUSB adapter?

```
// Accepts iPhone, but exposes MicroUSB interface
class LightningToMicroUsbAdapter implements IMicroUsbPhone {
    var lightningPhone: ILightningPhone

    LightningToMicroUsbAdapter(lightningPhone: ILightningPhone) {
        this.lightningPhone = lightningPhone
    }

    override fun useMicroUsb() {
        println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    override fun recharge() {
        lightningPhone.recharge();
    }
}
```

Bridge Pattern

The Bridge Pattern is very similar to an Adapter. It aims to decouple an abtraction from its implementation, and evolve those hierarchies independently from each other. The decision on which implemenation to apply isn't made at compile-time, but run-time instead. e.g., pImpl in C++.

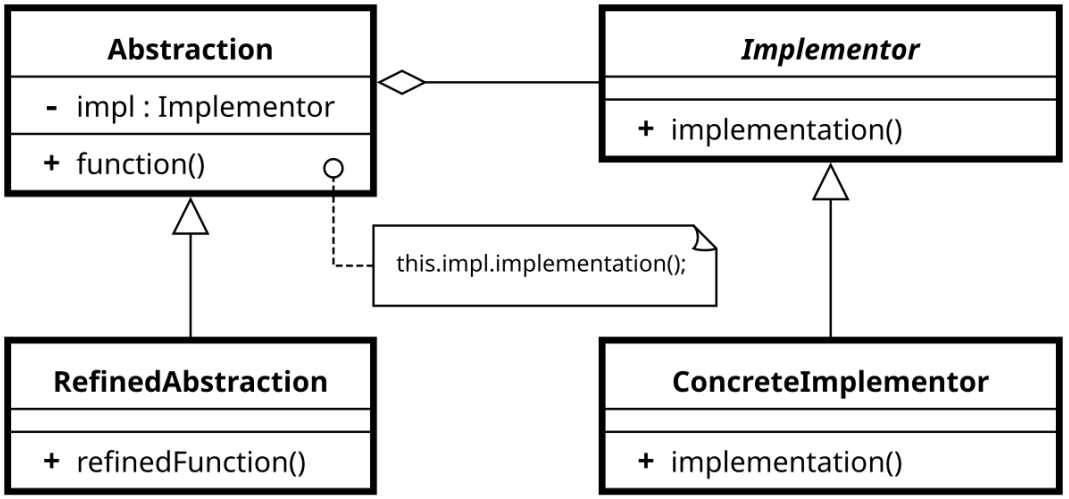


Figure 11: Bridge separates implementation and abstraction, so that the abstraction delegates responsibility to the implementation at runtime. Image courtesy of Wikipedia

Bridge Pattern

```
internal fun interface Logger {
    fun log(message: String?)

    // static methods
    companion object {
        fun info(): Logger {
          return Logger { message → println("info: $s") }
        }

        fun warning(): Logger {
            return Logger { message → println("warning: $s") }
        }
    }
}
```

Bridge Pattern

A banking application separates operations from the logging of those operations.

```
abstract class AbstractAccount {
    private Logger logger = Logger.info();

    public void setLogger(Logger logger) { this.logger = logger }

    protected void operate(String message, boolean result) {
        logger.log(String.format("%s result %s", message, result));
    }
}
//  ...

account.withdraw(75);

if (account.isBalanceLow()) {
    account.setLogger(Logger.warning());
}
```

Decorator Pattern

The Decorator Pattern allows behaviour to be added to an instance of a class, without modifying the class itself. It's an effective alternative to subclassing in extending an object's behaviour.

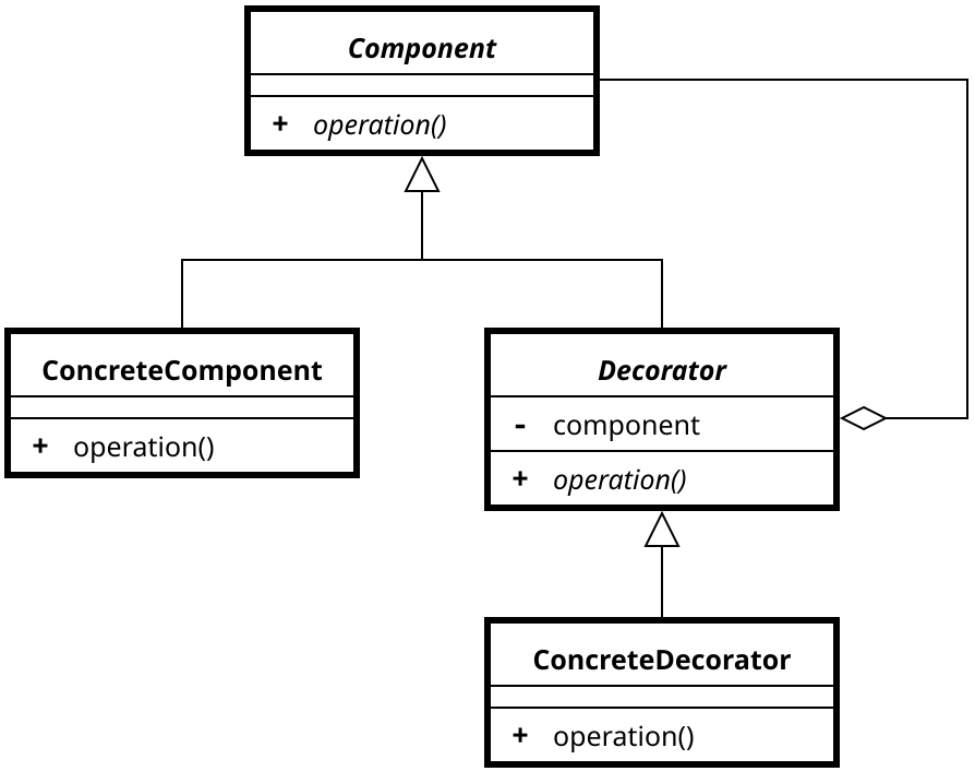


Figure 12: Decorator injects a class to manage access to an underlying component's functionality. Image courtesy of Wikipedia

Decorator Pattern

The classic example of a decorator is a user-interface element that is extended and customized using a decorator. e.g., a button that is decorated with an image to become an ImageButton, or a Window that is decorated with a scrollbar to become a ScrollableWindow.

```
// Window interface
interface Window {
  fun draw() // draws the Window
  fun getDescription(): String // returns a description
}

// Window without scrollbars
class SimpleWindow : Window {
  override void draw() { /* draw window */ }

  override getDescription() {
    return "simple window";
  }
}
```

Decorator Pattern

```
// abstract decorator class
// note that it implements Window

abstract class WinDecorator(val winToDecorate: Window) : Window
  {
    override fun draw() {
      winToDecorate.draw(); // delegation
    }

    override fun getDescription(): String {
      return winToDecorate.getDescription(); // delegation
    }
}
```

Decorator Pattern

```
// decorator which adds scrollbar functionality
// this is similar to bridge and adapter patterns

class ScrollDecorator(val windowToDecorate: Window) : WinDecorator {
    override fun draw() {
      super.draw();
      drawScrollBar();
    }

    fun drawScrollBar() {
      // Draw the vertical scrollbar
    }

    fun getDescription(): String {
        return ("${super.getDescription()} including scrollbars");
    }
}
```

Dependency Injection Pattern

Dependency Injection is a programming technique in which an object or function receives other objects or functions that it requires, as opposed to creating them internally. The main uses for this pattern are to:

- To decouple a service and consumer. DI promotes loose coupling.
- To support different configurations of services from a single codebase i.e., we describe components in terms of interfaces, and choose the implementation at runtime.

This pattern is used extensively as a way to promote testability since we can isolate dependencies by substituting modules at the time of testing e.g., replacing a service with a mock for unit tests.

Dependency Injection Pattern

There are multiple ways to accomplish this pattern:

- **Constructor injection**, where dependencies are provided through a client's class constructor.
- **Method injection**, where dependencies are provided to a method only when required for specific functionality.
- **Interface injection**, where the dependency's interface provides an injector method that will inject the dependency into any client passed to it.

Here's an example of a dependency being introduced in our Client class:

```
class Client {
  var service: Service

  init {
    // dependency is hard-coded.
    // this.service is tightly coupled to our client
    this.service = new ExampleService();
  }
}
```

Dependency Injection Pattern

We can easily remove this strong dependency by introducing the service instance through the constructor. This lets us easily pass in a different implementation.

```
interface IService { /* details aren't important here */ }

public class Client {
  var service: Service

  // The dependency is injected through a constructor.
  Client(service: IService) {
      this.service = service
  }

  fun login(token: Token) {
    if (service ≠ null) {
      service.login(token)
    }
  }
}
```

Dependency Injection Pattern

Other methods i.e., injecting through a method, are relatively simple.

```
class Client {
  fun performAction(Service service) {
    if (service == null) {
      throw new IllegalArgumentException("service is null");
    }
    service.execute();
  }
}
```

DI is often implemented using a framework that takes over responsibility for service registration and managing dependencies. For this reason, it is often linked to the principle of Inversion of Control [6], where program execution is directed by an external agent i.e. the DI framework.

We'll discuss these topics further in Testing, and Service lectures.

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

[6] M. Fowler, “Inversion of Control Containers and the Dependency Injection pattern.” [Online]. Available: https://martinfowler.com/articles/injection.html