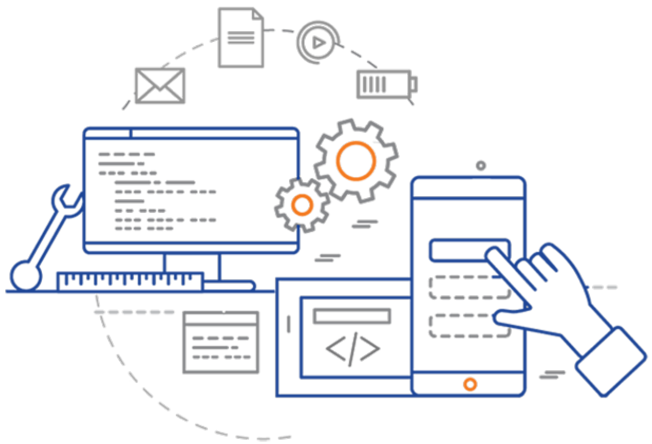


CS 398: Application Development

Week 04 Video: Analysis & Design 4

Design Patterns

Design Patterns

Well-designed solutions to common problems.

Design Patterns

A design pattern is a generalizable software solution to a common problem.

Using a known design pattern provides you with a template for a well-designed solution, which may be superior to a home-grown solution. It also gives you common-ground for design discussions.

Patterns originated with Christopher Alexander, an architect, in 1977. Design patterns in software gained popularity with the book Design Patterns: Elements of Reusable Object-Oriented Software [Gamma 1994].

Criticisms

Design patterns have seen mixed-success. Some criticisms levelled:

- They are not comprehensive, and do not reflect all styles of software or all problems encountered.
- They are old-fashioned and do not reflect current software practices.
- They add flexibility, at the cost of increased code complexity.

These criticisms are not entirely fair.

- While they certainly do not represent all styles of problems that you might encounter, they do cover a surprisingly broad range of them.
- Software design hasn't changed *that* much. For some specific problems, there are design patterns that are widely recognized as the preferred approach. You would be remiss to not consider using that pattern.
- They might add a small amount of complexity, but the benefit is well-structured, correctly working code. This is usually a worthwhile tradeoff!

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

- Creational Patterns control the dynamic creation of objects.
- Structural Patterns are about organizing classes to form new structures.
- Behavioural Patterns are about identifying common communication patterns between objects.

We'll describe patterns at a high-level, and then call out some commonly used patterns for application development.

Creational Patterns

Creational Patterns control the dynamic creation of objects.

Pattern	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.
Singleton	Ensure a class has only one instance, and provide a global point of access to it.

Design Patterns

Builder Pattern

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Imagine that you have a class with a large number of variables that need to be specified when it is created. e.g. a house class, where you might have 15-20 different parameters to take into account, like style, floors, rooms, and so on. How would you model this?

- You could create a single class to do this, but you would then need a huge constructor to take into account all of the different parameters.
- You could create subclasses, but then you have a potentially huge number of subclasses, some of which you may not actually use.

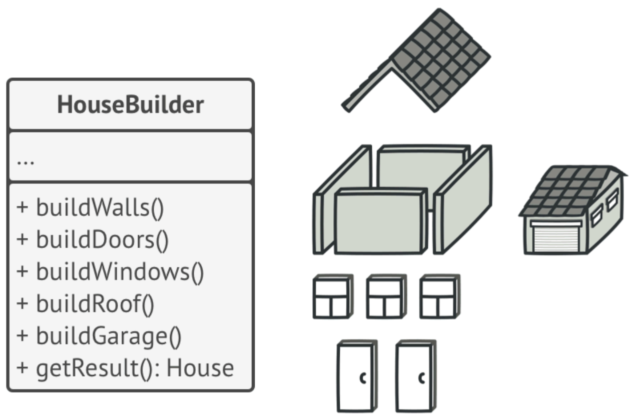
Design Patterns

Builder Pattern

The builder pattern suggests that you put the object construction code into separate objects called **builders**.

The pattern organizes construction into a series of steps.

- After calling the constructor, you call methods to invoke the steps in the correct order.
- You only call the steps that you require, which are relevant to what you are building.



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.

Design Patterns

Builder Pattern

Even if you never utilize the Builder pattern directly, it's used in a lot of complex Kotlin and Android libraries. e.g. the Alert dialogs in Android and other toolkits.

```
val dialog = AlertDialog.Builder(this)
    .setTitle("Title")
    .setIcon(R.mipmap.ic_launcher)
    .show()
```

Design Patterns

Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Why is this pattern useful?

1. **Ensure that a class has just a single instance**. The most common reason for this is to control access to some shared resource—for example, a database or a file.
2. **Provide a global access point to that instance**. Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

Design Patterns

Singleton

All implementations of the Singleton have these two steps in common:

1. **Make the default constructor private**, to prevent other objects from using the new operator with the Singleton class.
2. **Create a static creation method** that acts as a constructor.

e.g.

```
var s = Singleton.getInstance()
```

```
public class Singleton {

  private static Singleton instance = null;
  private Singleton() {
  }

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
        if (instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Design Patterns
Singleton

In Kotlin, it's significantly easier.

The ‘object’ keyword in Kotlin defines a static instance of a class. Effectively, an object is a singleton and we can just call its methods statically.

Like any other class, you can add properties and methods if you wish.

```
object Singleton {
    init {
        println("Singleton class invoked.")
    }
    fun print(){
        println("Print method called")
    }
}

fun main(args: Array<String>) {
    Singleton.print()
    // echos "Print method called" to the screen
}
```

Structural Patterns

Structural Patterns are about organizing classes to form new structures.

Pattern	Description
Adapter, Wrapper	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality
Proxy	Provide a surrogate or placeholder for another object to control access to it.

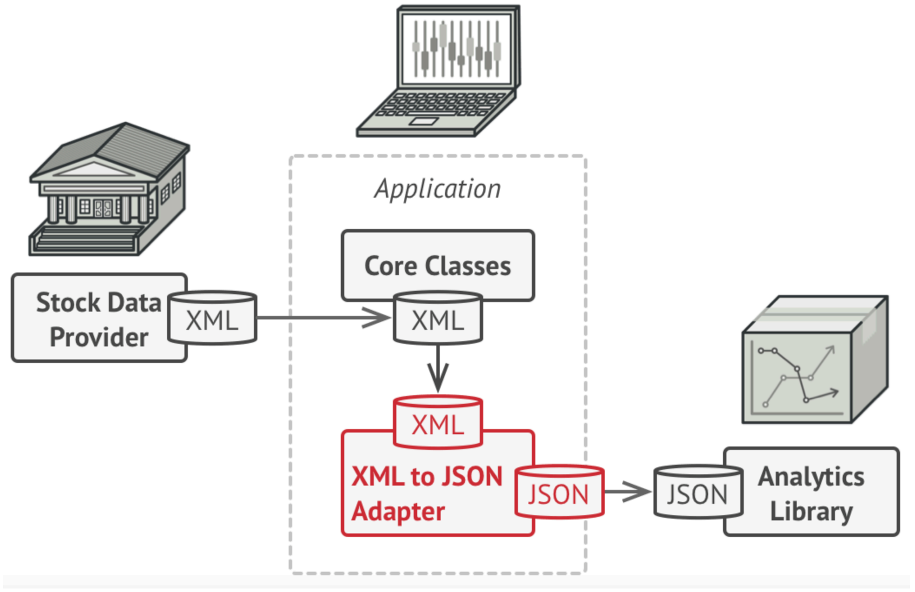
Design Patterns

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Imagine that you have a data source that is in XML, but you want to use a charting library that only consumes JSON data.

An adapter is an intermediate component that converts from one interface to another. In this case, it could handle the complexities of converting data between formats.

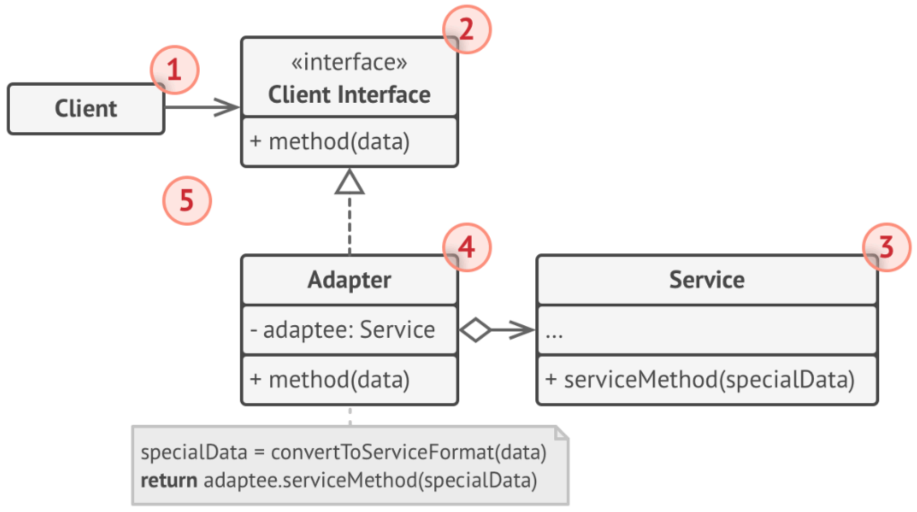


Design Patterns

Adapter

The simplest way to implement this is using **object composition**: the adapter is a class that utilizes an interface provided to the main application (client).

1. The **client** is the class containing business logic (i.e. an application class that you control).
2. The **client interface** describes the interface that you have designed for your application.
3. The **service** is some useful library or service (typically which is closed to you).
4. The **adapter** is the class that you create to serve as an intermediary between these interfaces.
5. The **client application** isn't coupled to the adapter because it works through the client interface.



```
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)
```

Behavioural Patterns

Behavioural Patterns are about identifying common communication patterns between objects.

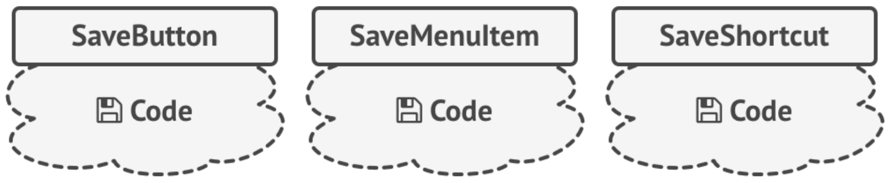
Pattern	Description
Command	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Observer	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.

Design Pattern
Command

Command is a behavioural 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).

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 that actually handles saving the data?

If you attach it to the object that the user is interacting with, then you risk duplicating the code. e.g.

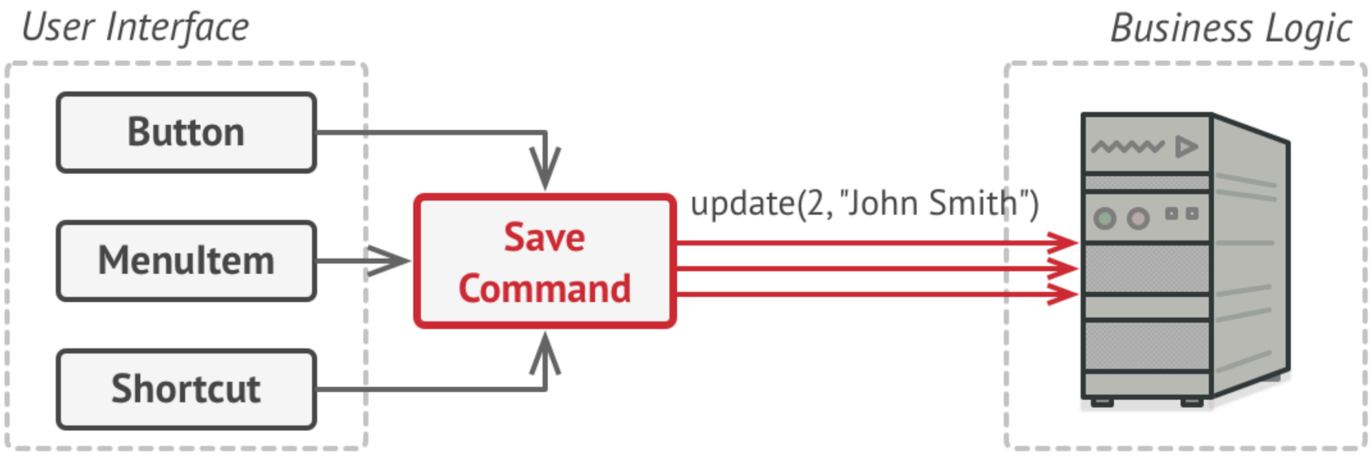


Several classes implement the same functionality.

Design Pattern

Command

The Command pattern suggests that you encapsulate the details of the command that you want executed into a separate request, which is then sent to the business logic layer of the application to process.

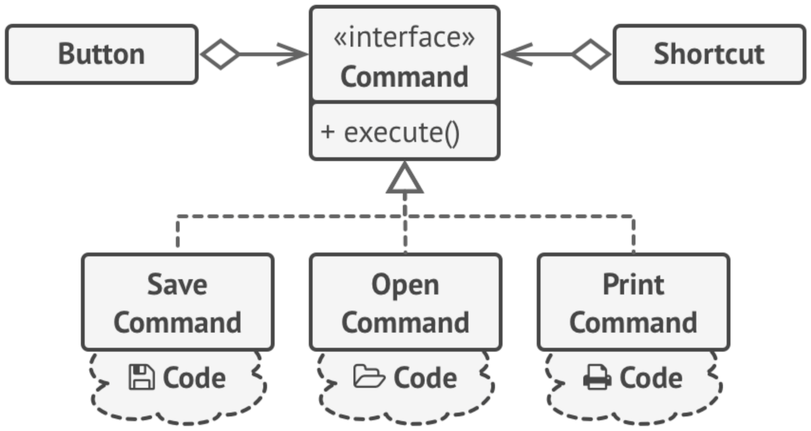


Design Pattern
Command

The command class relationship to other classes.

In other words, the specific instance of the Command contains the code that should be executed. Relevant UI elements would have a reference to the same Command instance, and could tell it to invoke itself as needed.

SaveButton -> saveCommand.execute()

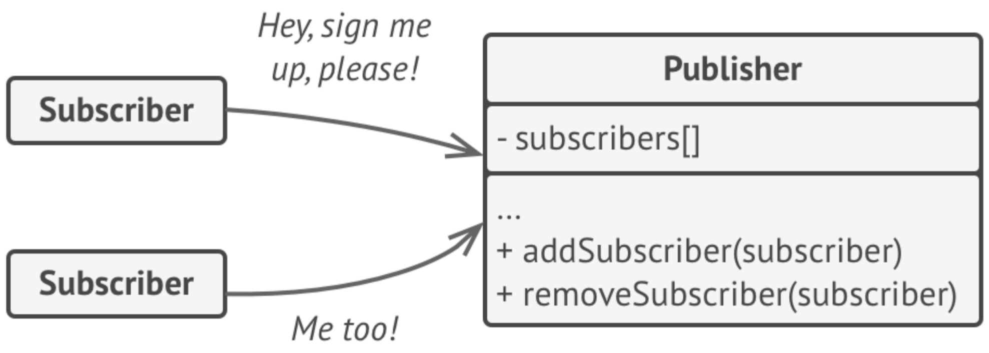


Design Pattern
Observer

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 observing. This is also called *publish-subscribe*.

The object that has some interesting state is often called **subject** (or publisher). Objects that want to track changes to the publisher's state are called **observers** (subscribers) of the state of the publisher.

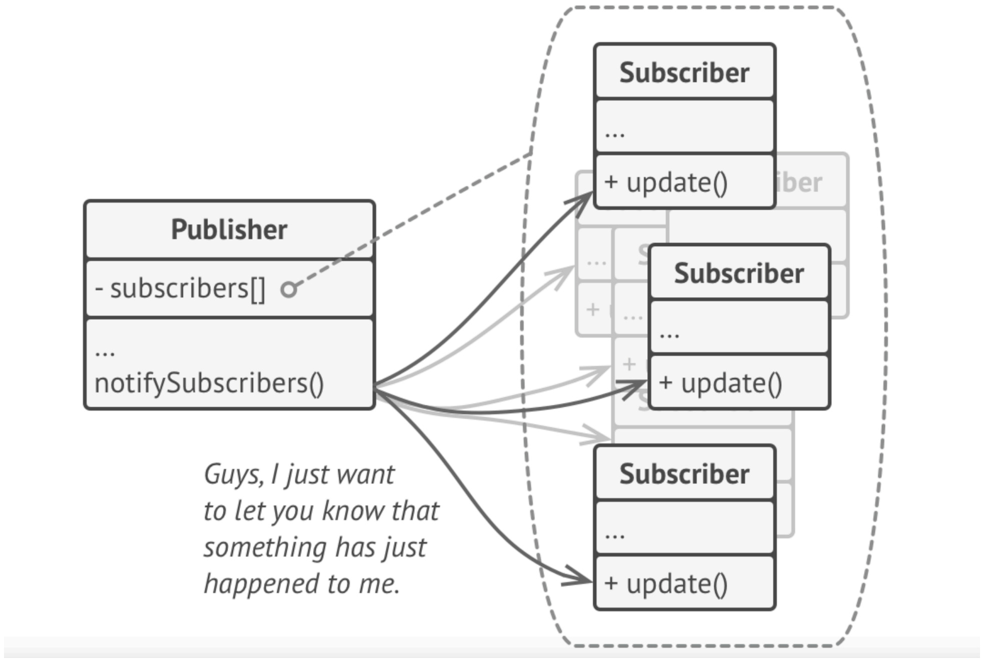
Subscribers register their interest in the subject, who adds them to an internal subscriber list.



Design Pattern
Observer

When something interest happens, the publisher notifies the subscribers through a provided interface. The subscribers can then react to the changes.

A modified version of Observer is the Model-View-Controller (MVC) pattern, which puts a third intermediate layer between the Publisher and Subscriber, which manages user input. That layer is not required for this pattern.



Resources

Diagrams and examples were taken in-part from the following sources.

- Eric Gamma et al. 1994. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley Professional. ISBN 978-0201633610.
- Robert C. Martin. 2003. **Agile Software Development: Principles, Patterns and Practices**. Pearson. ISBN 978-0135974445.
- Alexander Shvets. 2019. **Dive Into Design Patterns**. Refactoring.Guru (self-published).
- Alexey Soshin. 2018. **Hands-On Design Patterns with Kotlin**. Packt Publishing.
- Joel Spolsky. 2000. **Things You Should Never Do, Part 1**. https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
- Ugonna Thelma. 2020. **The S.O.L.I.D. Principles in Pictures**. https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898