Design Principles

CS 346: Application Development

SOLID Principles

Revisiting principles from CS 246.

SOLID

SOLID was introduced by Robert (“Uncle Bob”) Martin around 2002.

The **SOLID Principles** tell us how to arrange our functions and data structures into classes, and how those classes should be arranged (“class” meaning “a grouping of functions and data”)

Their goal is the creation of mid-level software structures that:

- Tolerate change (flexibility, extensibility),
- Are easy to understand (readability), and
- Are the basis of components that can be used in many software systems (reusability).

There are five SOLID principles, and we’ll walk through them.

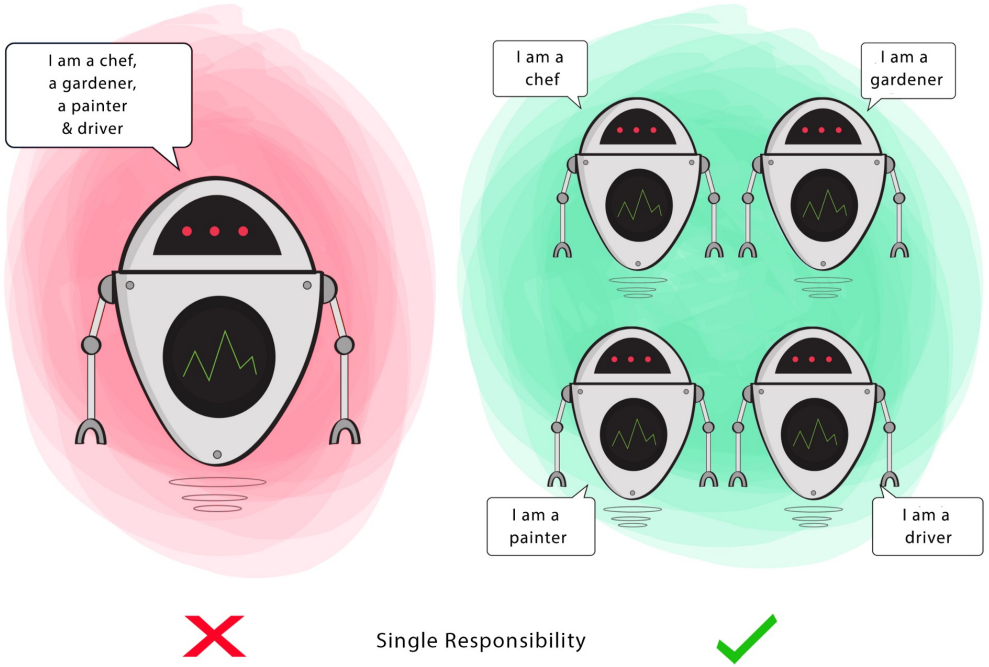
- Diagrams are taken from Ugonna Thelma: The S.O.L.I.D. Principles in Pictures.

1. Single Responsibility

The Single Responsibility Principle (SRP): we want classes to do a single thing.

This ensures that classes are focused but also reduces pressure to change that class.

- A class has responsibility over a single block of functionality.
- There is only one reason for a class to change.
- Applies to components, and other “units” of code, not just classes.

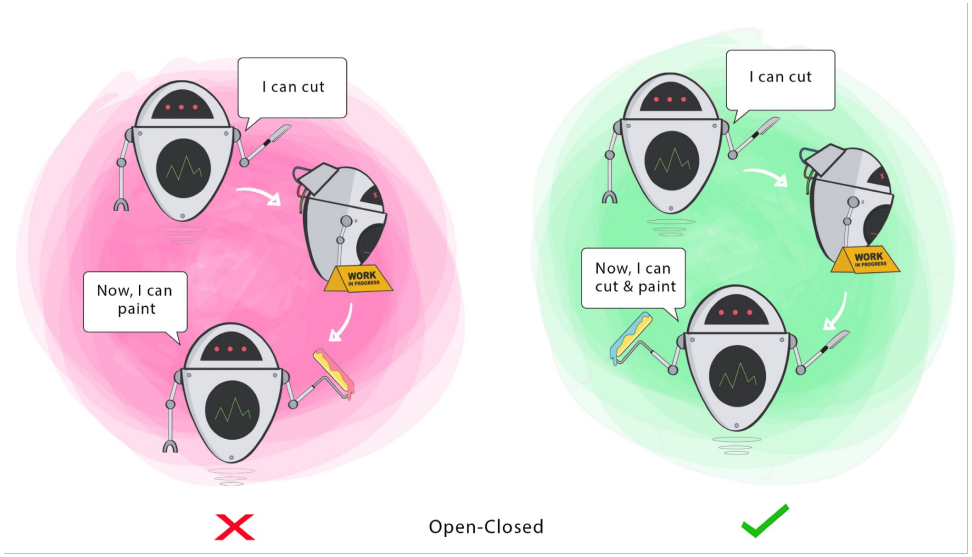


2. Open-Closed Principle

"**A software artifact should be open for extension but closed for modification**. In other words, the behaviour of a software artifact ought to be extendible, without having to modify that artifact. "

- – Bertrand Meyers (1988)

- Subclassing is the primary form of code reuse.
- A particular module (or class) should be reusable without needing to change its implementation.



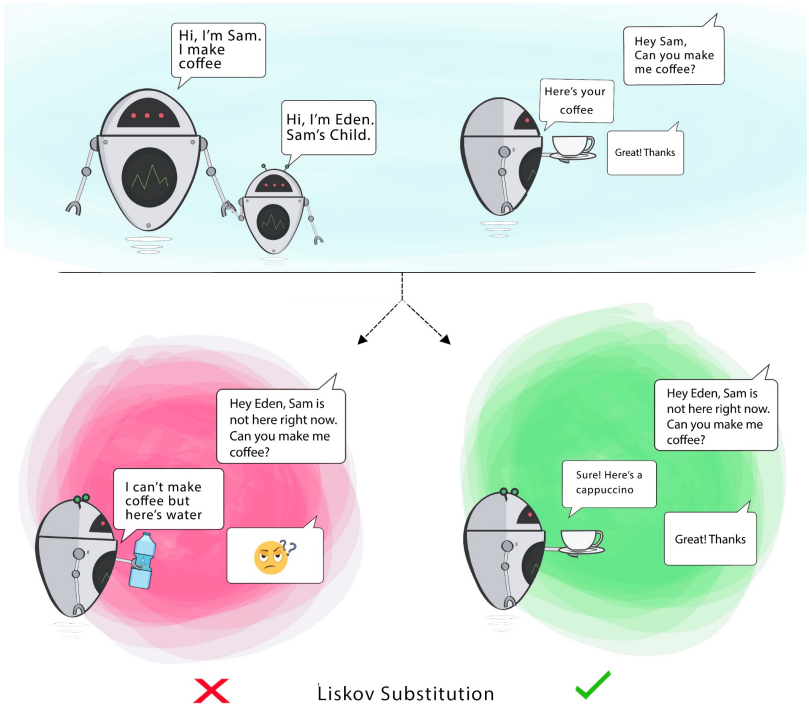
3. Liskov-Substitution Principle

"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2, then S is a subtype of T".

– Barbara Liskov (1988)

It should be possible to substitute a derived class for a base class, since **the derived class should retain the base class behaviour**.

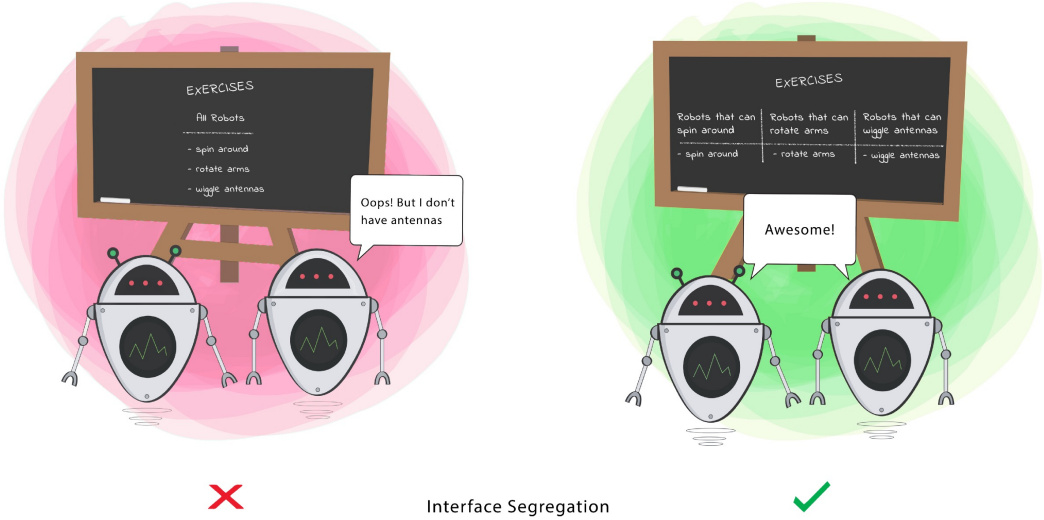
In other words, a child should always be able to substitute for its parent.



4. Interface Substitution

It should be possible to change classes independently from the classes on which they depend.

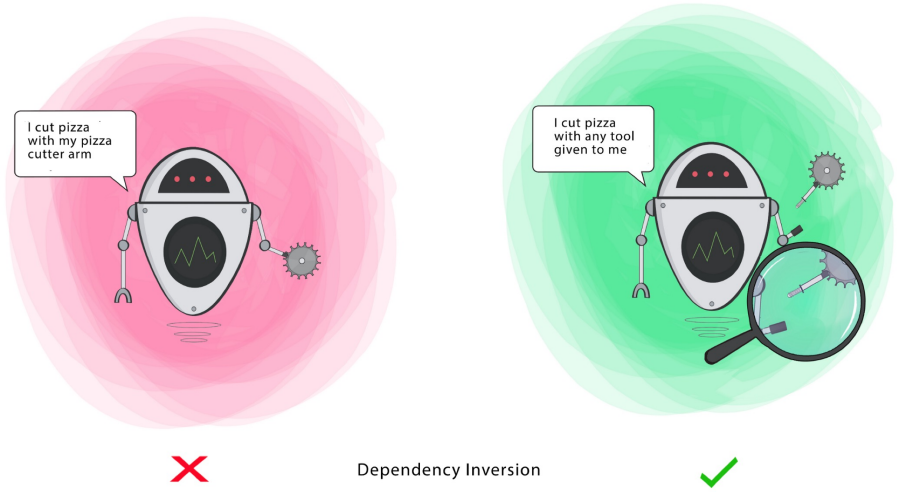
- Also described as "**program to an interface, not an implementation**". This means focusing your design on what the code is doing, not how it does it.
- If you code to an interface, it allows flexibility, and the ability to substitute other valid implementations.



5. Dependency Inversion

The most flexible systems are those in which **source code dependencies refer to abstractions (interfaces) rather than concretions (implementations)**. This reduces the dependency between these two classes.

- High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g. interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.



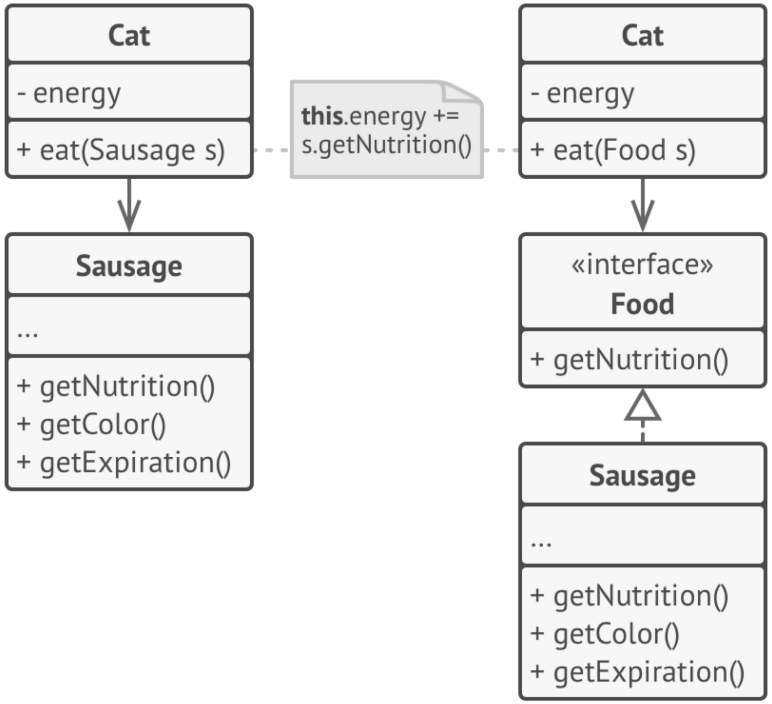
Additional Design Principles

Can we generalize these?

Program to an Interface (Extensibility)

Base dependencies between classes on common behavior defined in interfaces; don't assume implementation classes.

- This allows for maximum flexibility.
- When classes rely on one another, you want to minimize the dependency - we want *loose coupling*. To achieve this, extract an abstract interface and use that to describe the desired behaviour between the classes.
- e.g. our cat on the left can *only* eat sausage. The cat on the right can eat anything that provides nutrition, *including* sausage.

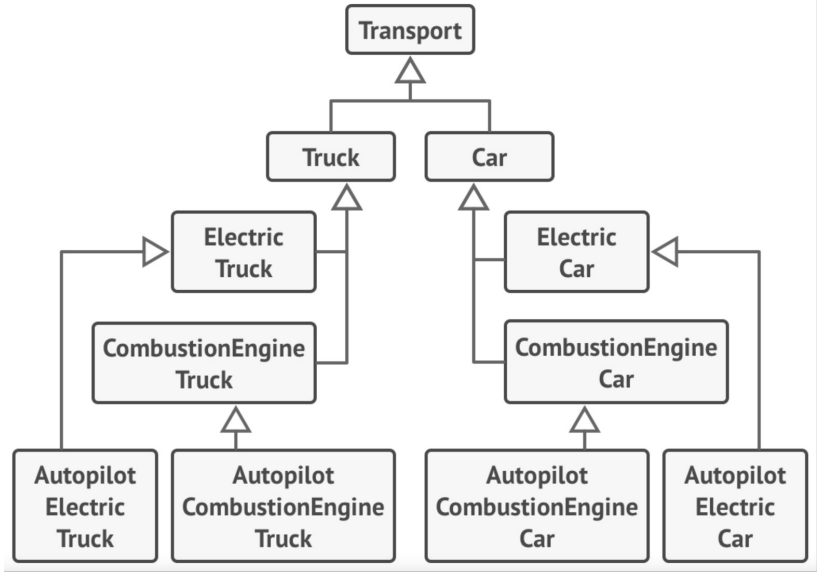


Favor Composition (Flexibility)

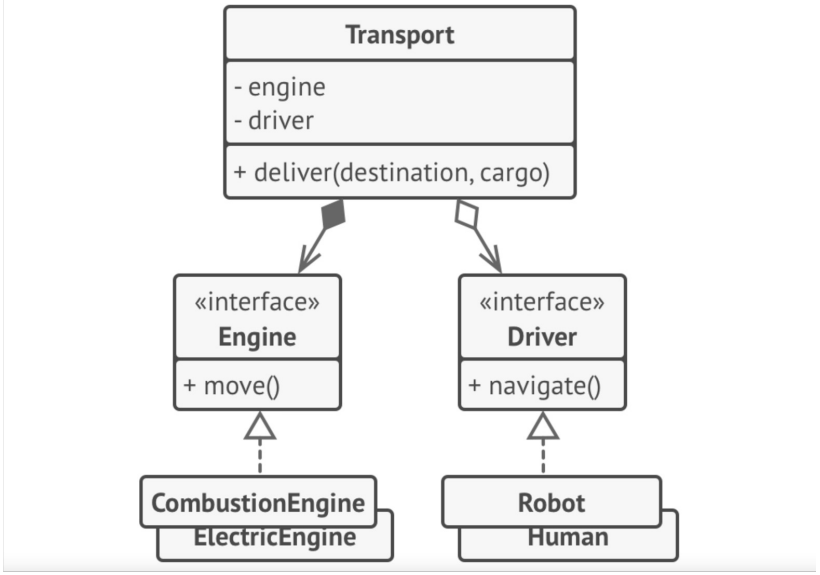
Inheritance is a useful tool for reusing code. In principle, it sounds great - derive from a base class, and you get behaviour for free! However, there can be negative side effects to inheritance.

- **A subclass cannot reduce the interface of the base class**. You have to implement all abstract methods, even if you don't need them.
- When overriding methods, you need to make sure that your new behaviour is compatible with the old behaviour. In other words, **the derived class needs to act like the base class**.
- **Inheritance breaks encapsulation**, because the details of the parent class are potentially exposed to the derived class.
- **Subclasses are tightly coupled to superclasses**. A change in the superclass can break subclasses.
- Reusing code through inheritance can lead to parallel inheritance hierarchies, and **explosion of classes**.

A useful alternative to inheritance is composition. Where inheritance represents an **is-a** relationship (a car is a vehicle), composition represents a **has-a** relationship (a car has an engine). Imagine a catalog application for cars and trucks.



Inheritance leads to class explosion, and unused intermediate classes.



Composition (aggregation) greatly reduces the complexity, and models based on supported behaviours.

— Alexander Shvets. 2019. **Dive Into Design Patterns**.

Favor immutability (Robustness)

Avoid side effects (aka unintended consequences).

Prefer a functional style as much as possible.

- Write functions that return modified data (and do not modify data that is passed in).
- Avoid using global variables (except as constants that do not change).

Avoid “Happy Path” Programming

Bake error handling into your design.

You should anticipate errors and design mechanisms that allow your application to continue processing, even when these errors occur.

Have a strategy for handling errors:

- **Favour immutable functions, with no side effects**. This reduces the chance of runtime errors.
- **Check function return values** to ensure that results are valid. Use Kotlin’s NULL handling correctly.
- **Never throw an exception without understanding where it will be handled**, otherwise this will just percolate up the call stack to the user (and crash).
- **Perform validation on user inputs** to avoid users entering invalid information that could cause issues.
- **Determine what recovery action is appropriate** for the type of error! e.g. retry in the case of a network error, or abort the operation in the case of an invalid file operation

Design patterns

Useful patterns in Kotlin.

Recap: Design patterns

A design pattern is a *generalizable software solution to a common problem*. Design patterns gained popularity with Design Patterns: Elements of Reusable Object-Oriented Software [Gamma et al 1994].

Why use design patterns?

- They represent a pattern that is (was) known to work well for a particular problem and context.
- They can result in a more extensible, flexible solution.

Criticisms:

- They are not comprehensive, and do not reflect all styles of software or all problems encountered.
- They trade increased complexity now for the promise of flexibility later (YAGNI?)

Types of patterns

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

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

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).

Creational Patterns

Creational Patterns control the dynamic creation of objects.

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.
Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Prototype	Specify the kinds of objects to create using a prototypical instance and create new objects from the 'skeleton' of an existing object.
Singleton	Ensure a class has only one instance and provide a global point of access to it.

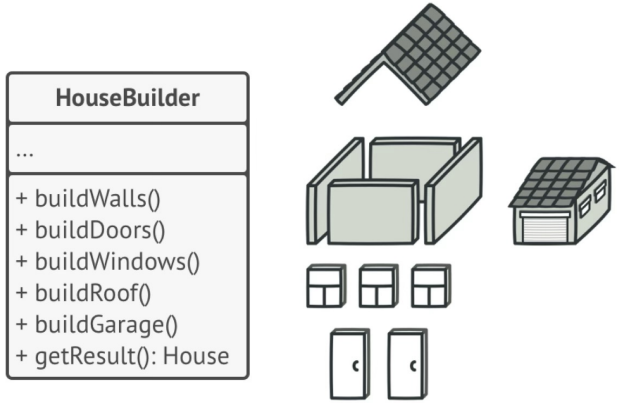
Builder Pattern

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

Builder lets you construct complex objects step by step. Produce different types and representations of an object using the same construction code.

A builder class generates the initial object, and subsequent methods can be called to customize it.

- After calling the constructor, 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.

Builder Pattern Example (Java)

How do you implement it in other languages? e.g., Java

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

Builder Pattern Example (Kotlin)

Kotlin supports named and default arguments, simplifying this.

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

Singleton Pattern

You want to control access to a shared or restricted resource.

A **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?

- Ensures 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.
- Provides 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.

Singleton Example (Java)

Here's an implementation:

1. Make the default constructor private, to prevent other objects from creating an instance directly.

2. Create a static method that to instantiate the class. This method ensures that the object is only instantiated once.

As needed, use the static method to return a static reference to the object.

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

    public static getInstance(): Singleton {
        if (instance == null) {
            instance = Singleton()
        }
        return instance
    }
}
// get a reference to it
Singleton s = Singleton.getInstance()
```

Singleton Example (Kotlin)

In Kotlin, implementation is 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.

You do not need initialize it — it’s lazy initialized as needed.

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


fun main(args: Array<String>) {
    Singleton.print()
    // echos “Singleton class invoked.”
    // echos "Print method called"
}
```

Behavioral Patterns

Behavioural Patterns are about identifying common communication patterns between objects.

Command	Encapsulate a request as an object, supporting 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.
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.
State	State allows an object to change its behaviour when state changes.
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.

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).

Several classes implement the same functionality.

Command Pattern Example (Kotlin)

```kotlin
// 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 Example (Kotlin)

```kotlin
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

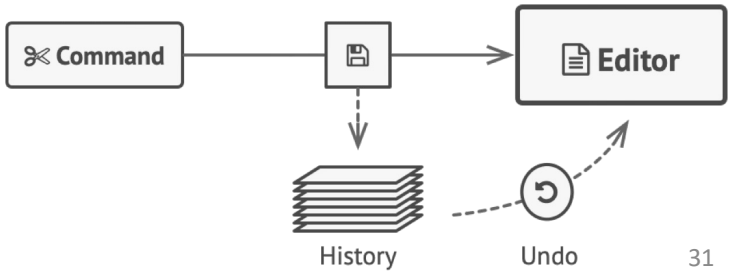
Memento captures an objects state so you can save/restore it later.

This is helpful for object versioning (e.g., tracking changes over time).

It's commonly used when reverting the state of a system with 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.



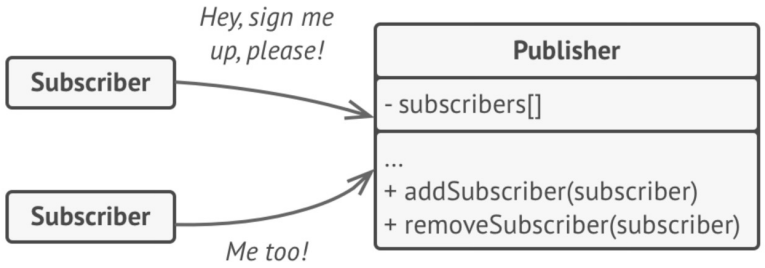
Memento Example (Kotlin)

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

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

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

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

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*.

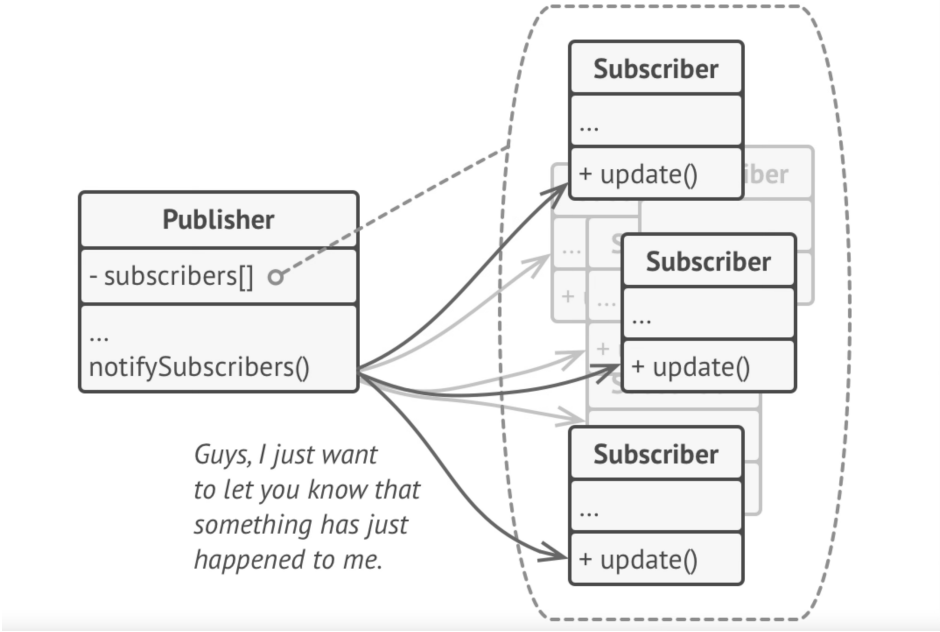
- The object that has some interesting state is often called **subject** (or publisher). Objects that want to track changes to the publisher's state are called **observers** (subscribers) of the state of the publisher.
- Subscribers register their interest in the subject. who adds them to an internal subscriber list.



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 to process user input (*not shown here*).



Observer Examples (Kotlin)

```kotlin
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()
        }
}
```

```kotlin
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
}
```