SOFTWARE ARCHITECTURE & DESIGN

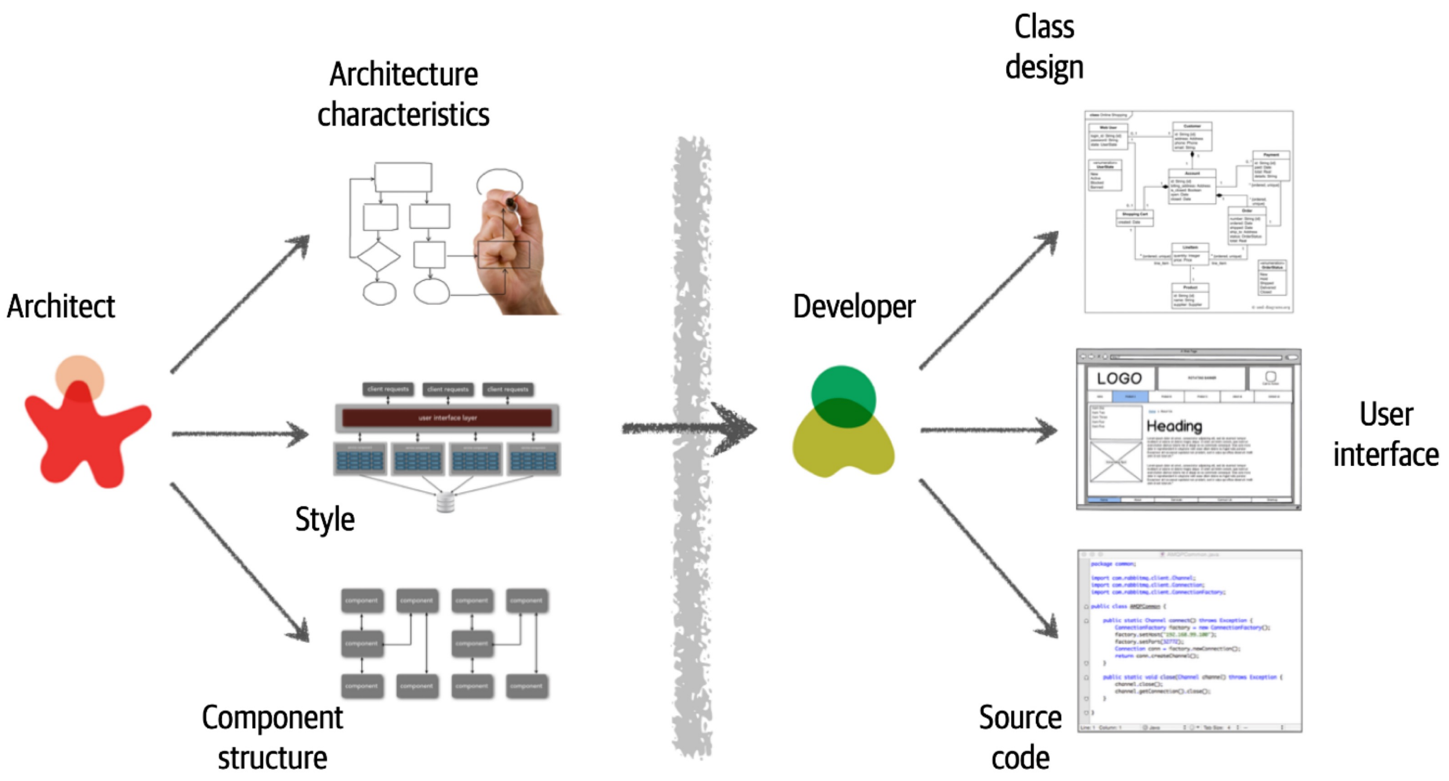
CS 346: Application Development

MEETING OUR GOALS

We have orthogonal goals, that we somehow need to align.

1. Features
   - We want our program to accomplish something for our users.
   - We will rely on UCD and our iterative process to ensure that they are *useful*.
2. Quality
   - Robustness: stability of the solution
   - Flexibility: can we add new features or extend existing features?
   - Reusability: how can we reuse our code?

Software architecture (and design) can help us address both functional and quality goals.



Architecture & Design operate at different levels of abstraction, but address similar goals.

From Mark Richards & Neal Ford. 2020. **Fundamentals of Software Architecture: An Engineering Approach.** O'Reilly.

SOFTWARE ARCHITECTURE

CS 346: Application Development

SOFTWARE ARCHITECTURE

Software architecture is primarily concerned with the **building blocks of a system and how they interact**. These building blocks implement the **functional requirements** that a software system must satisfy. There are also many **non-functional requirements**, and these are also important: for example, performance, time to market, costs, maintainability, reusability, modifiability, availability, and simplicity…

— Oliver Vogel et al (2011)

Architecture is about *structure* and how the choice of structure affects the features that we build, and the qualities that resulting solution will have. e.g., extensibility, reusability, scalability, and so on.

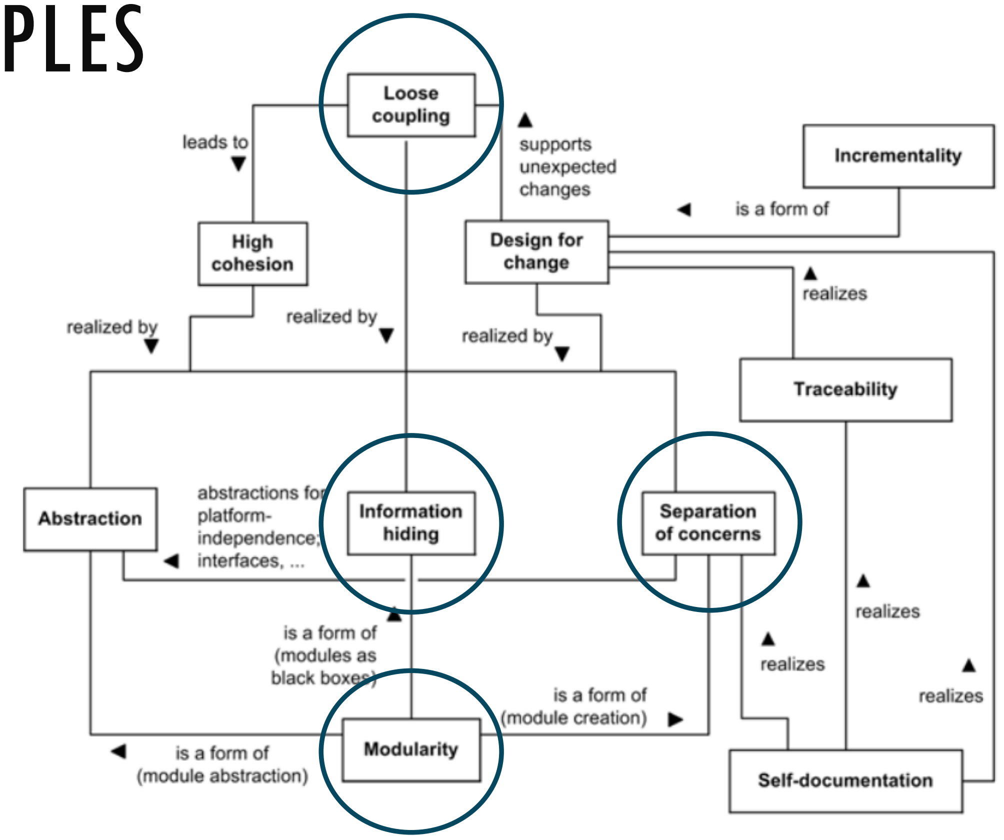
ARCHITECTURAL PRINCIPLES

How do we meet both functional goals, and non-functional goals?

We can apply some software engineering principles that improve our ability to maintain and extend our software.

We'll focus on these:

- Loose coupling & high cohesion
- Modularity
- Separation of concerns
- Information hiding



- Oliver Vogel et al. 2011. **Software Architecture.** Springer.

COUPLING-COHESION

Loose Coupling: reduce coupling between components as much as possible; functionality should be isolated within a component.

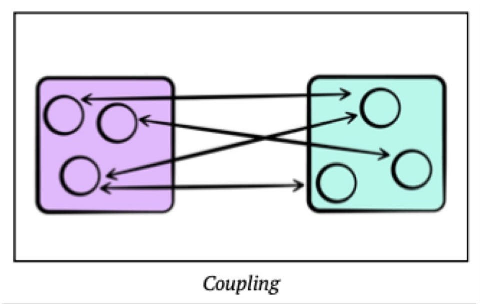
High Cohesion: parts of a class (or module) should be self-contained.

These work together:

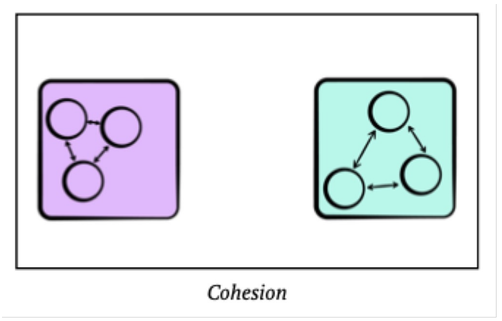
- Each module is easier to understand (if functionality is self-contained)
- Each module is easier to modify (if changes are contained)

Examples:

- **High coupling**: classes can access each other's (private) data. (BAD)
- **Medium coupling**: classes communicate via a global data structure. (BETTER)
- **Low coupling**: classes communicate through an interface/methods. (BEST)



Coupling refers to how closely linked components or modules are to each other.



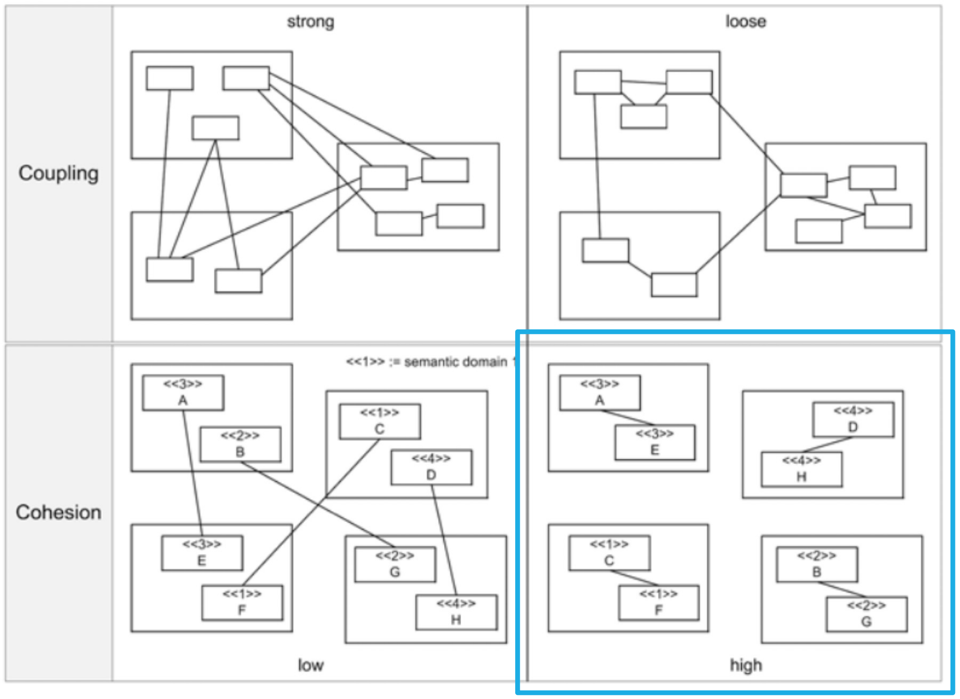
Cohesion is a measure of how closely related the parts of a module are.

You can identify the level of coupling between modules or classes by looking at the methods calls that are made.

High number of calls <u>within</u> a class? **High cohesion.**

High number of calls <u>between</u> classes? **High coupling.**

We generally want to be on the 'lower-right-hand corner' of this diagram.



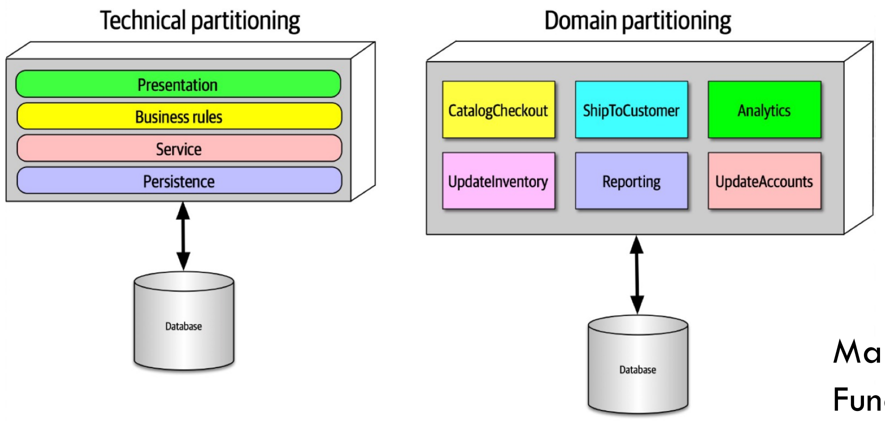
- Oliver Vogel et al. 2011. **Software Architecture**. Springer.

MODULARITY

Modularity refers to the logical grouping of source code into related groups. This can be realized as namespaces (C++), or packages (Java or Kotlin). Modularity reinforces a separation of concerns and encourages reuse of source code.

- **Technical partitioning**: group functionality according to technical capabilities.
- **Domain partitioning**: group functionality according to the domain/area of interest.

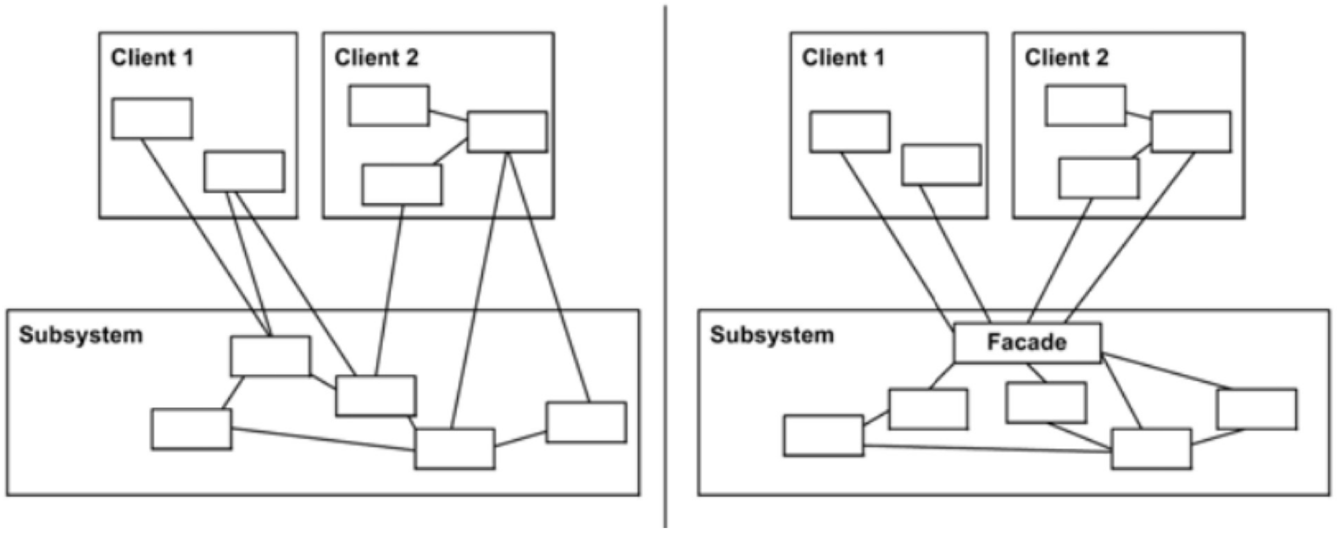
Applications often leverage technical partitioning



Mark Richards & Neal Ford. 2020. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly.

Related principles that support and work with modularity:

- **Separation of concerns** also includes keeping responsibility contained within a module.
- **Information hiding** makes a particular module responsible for its own data exclusively. (This obviously applies to classes as well; modules are just a different level of abstraction).



- Oliver Vogel et al. 2011. **Software Architecture**. Springer.

WHAT IS A STYLE?

An architectural style (aka *pattern*) is an **overall structure that describes how our components are organized and structured, and how they communicate.**

Like design patterns, an architectural style is a general solution that has been found to work well at solving specific types of problems.

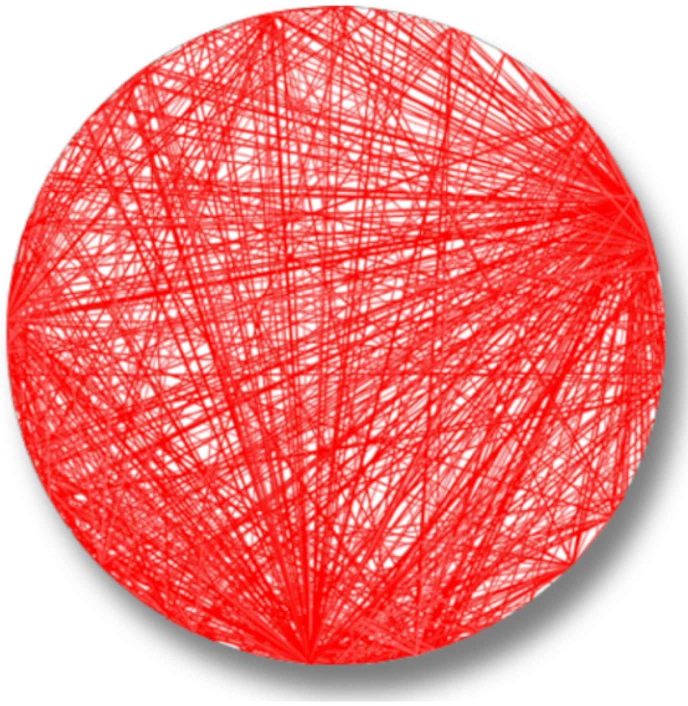
An architectural style describes both the **topology** (organization of components) and the associated architectural **characteristics** (qualities) for that topology.

ANTI-STYLE: BIG BALL OF MUD

A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle.

These systems show unmistakable signs of **unregulated growth**, and **repeated, expedient repair**. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.

-- Foote & Yoder 1997.



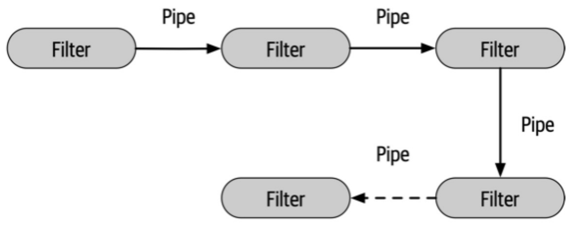
MONOLITHIC: PIPELINE

A **pipeline** architecture is appropriate when we want to transform data sequentially. It consists of pipes and filters.

Pipes form the communication channel between filters. Each pipe is unidirectional, accepting input, and producing output.

Filters perform operations on data that they are fed. Each filter performs a single operation, and they are stateless.

- **Producer**: The outbound starting point (also called a source).
- **Transformer**: Accepts input, optionally transforms it, and then forwards to a filter (this resembles a *map* operation).
- **Tester**: Accepts input, optionally transforms it based on the results of a test, and then forwards to a filter (aka *reduce*).
- **Consumer**: The termination point, where data can be saved.



Advantages

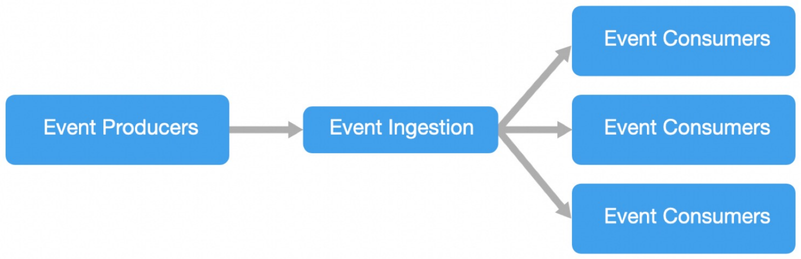
- Easy to extend by adding another step in the chain.
- Filters are stateless so they can be tested independently.
- Broadly applicable to any sequence of operations e.g., shell programming.

MONOLITHIC: EVENT DRIVEN

An **event-driven architecture** is designed around the production, transmission and consumption of events between loosely-coupled components.

Unlike a pipeline architecture, which tends to assume linear ordering of consumers, an event-driven architecture expects multiple consumers for a particular event and works best when ordering isn't critical.

- Producers and consumers have no knowledge of one another.
- Consumers are also independent and have no knowledge of what each other is doing with an event.



Advantages

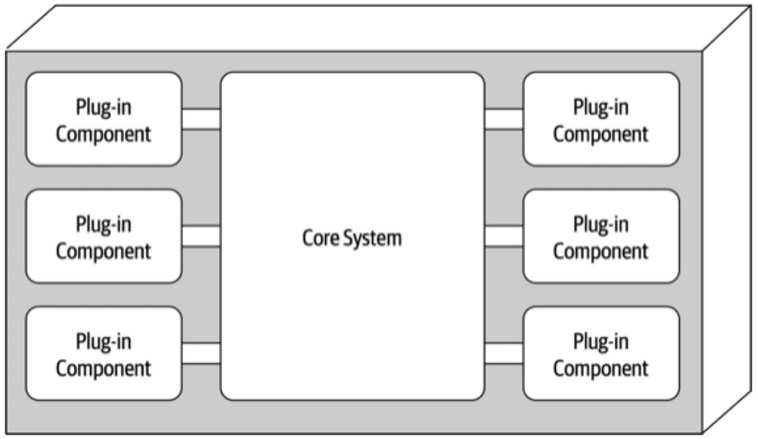
- Useful in systems that generate and handle large volumes of data.
- Can be used in monolithic applications or distributed applications (with restrictions).
- e.g. a system that manages hardware events or interrupts.

MONOLITHIC: MICROKERNEL

A **microkernel architecture** (aka plugin architecture) is a popular pattern that provides the ability to easily extend application logic to external, pluggable components.

This architecture works by focusing the primary functionality into the core system and providing extensibility through the plugin system.

- Plugins can be from different developers.
- e.g. IntelliJ uses a plugin architecture to allow you to install third-party extensions.



Advantages

- Great flexibility and extensibility
- Can add plug-ins while the application is running
- Plug-in modules can be tested in isolation.
- Good portability

MONOLITHIC: LAYERED

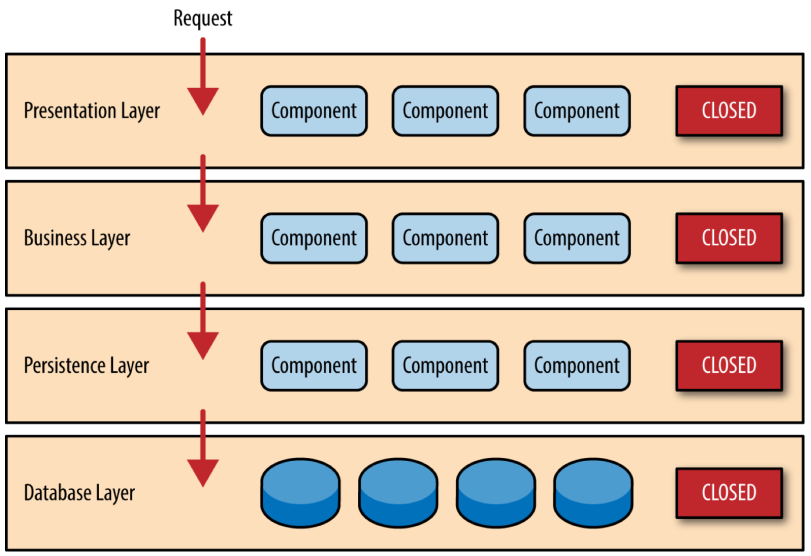
We'll focus on this pattern for Desktop + Mobile GUI development.

A **layered** or *n-tier* architecture is a very common architectural style that organizes software into horizontal layers, where each layer represents a ***logical*** division of functionality.

Each layer has specific functionality that is presents to the layer above (i.e. lower layers provide services up the stack).

Dependencies extend down: lower-levels provide functionality that is consumed by higher-levels.

e.g. presentation uses business logic, but business layer doesn't know anything about the UI.



Advantages
•Layers remain isolated ("separation of concerns")
•High testability: components belong to specific layers.
•High ease of development.

SOFTWARE DESIGN

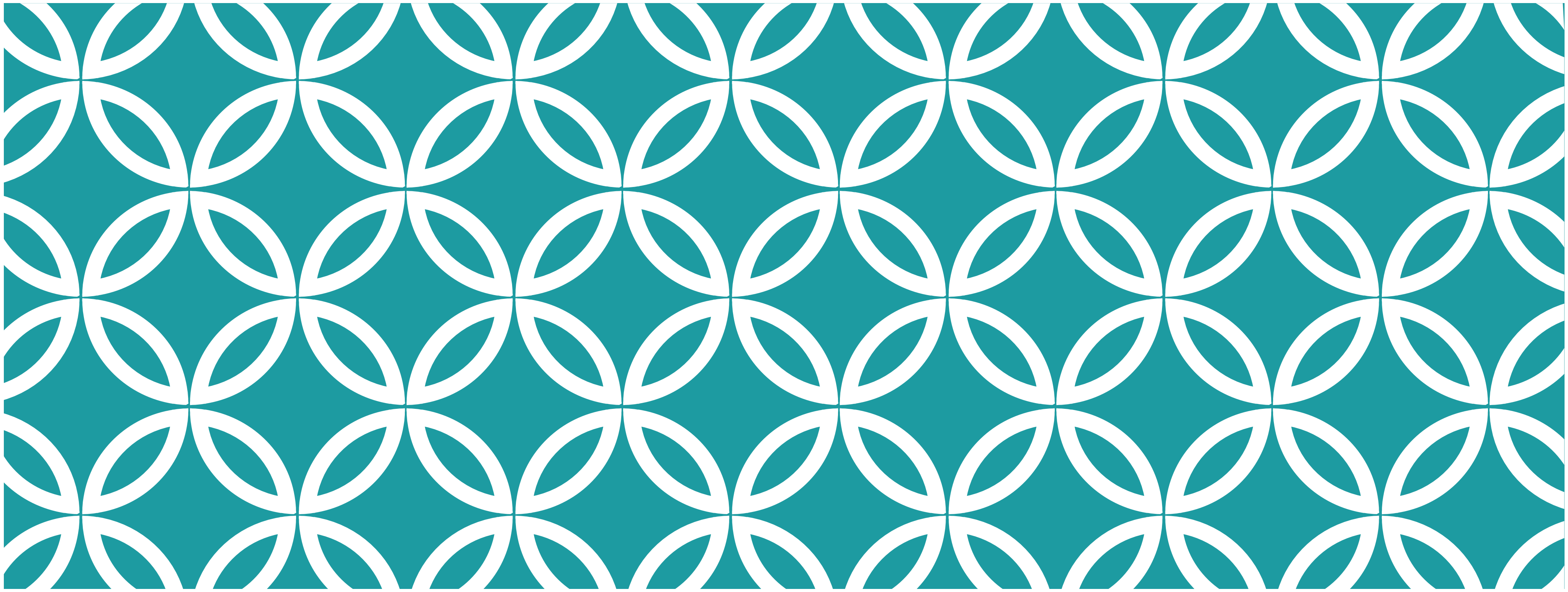
CS 346: Application Development

WHAT IS SOFTWARE DESIGN?

The term “software design” is overloaded.

- A **UX designer** will treat design as the process of working with users to identify requirements, and iterating on the interaction and experience design with them to fine tune how they want the experience to work (*we did this in the requirements phase!*)
- A **software engineer** will want to consider ways of designing modules and source code that emphasize desirable characteristics like scalability, reliability and performance.
- A **software developer** may want to consider structure of their code, readability and maintainability, and correctness of the results (among other things).

In this course, we’ll consider design to be the **implementation decisions that are made *prior to writing code*,** that lead to a “good” (effective, appealing) design.



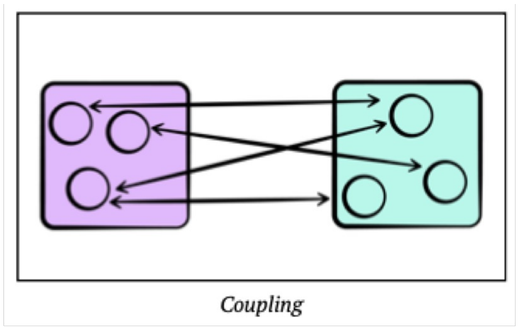
DESIGN PRINCIPLES

CS 346: Application Development

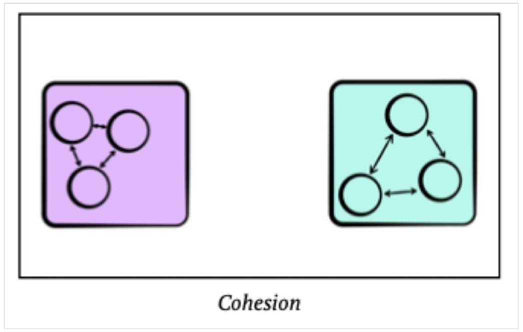
COUPLING/COHESION (FLEXIBILITY)

When designing components, create self-contained entities (high-cohesion) with minimal dependencies on one-another (loose-coupling).

As much as possible, minimize dependencies and enforce separation across layers!



Coupling refers to how closely linked components or modules are to each other.



Cohesion is a measure of how closely related the parts of a module are to each another.

ENCAPSULATE WHAT VARIES (FLEXIBILITY)

Identify the aspects of your application that vary and separate them from what stays the same. The goal is to minimize the effects of changes in code.

You can do this by encapsulating classes, or functions. In both cases, your goal is separate and isolate the code that is likely to change from the rest of your code. This minimizes what you need to change over time.

```
fun getOrderTotal(order) {
    var total = 0
    for (item in order.lineItems) {
        total += item.price * item.quantity

        if (order.country == "US")
            total += total * 0.07 // US sales tax
        else if (order.country == "EU")
            total += total * 0.20 // European VAT
    }
    return total
}
```

$\Rightarrow$

```
fun getOrderTotal(order) {
  total = 0
  for (item in order.lineItems) {
    total += item.price * item.quantity
  }
  total += total * getTaxRate(order.country)
  return total
}

fun getTaxRate(country) {
   return when (country) {
      "US" -> 0.07 // US sales tax
      "EU" -> 0.20 // European VAT
      else -> 0
   }
}
```

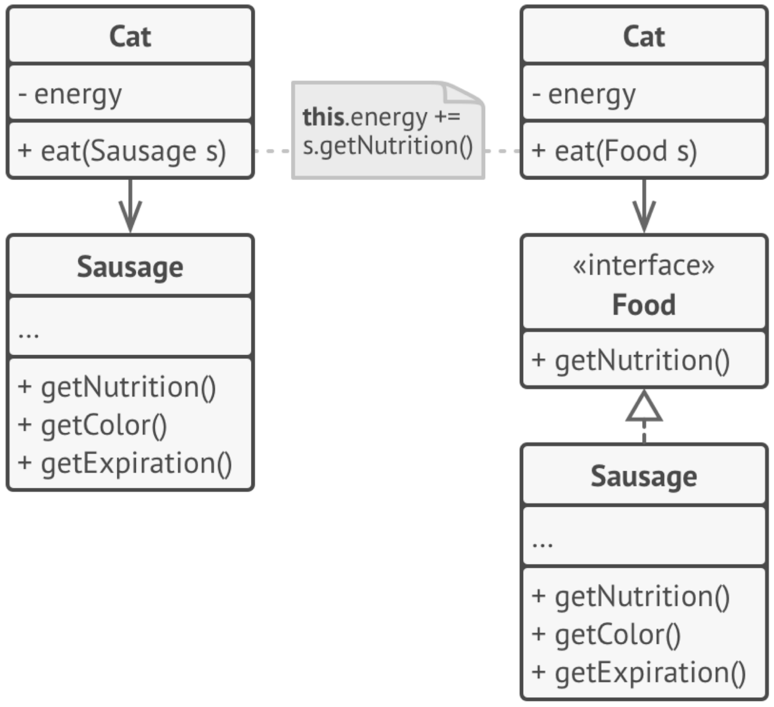
PROGRAM TO AN INTERFACE (EXTENSIBILITY)

Program to an interface, not an implementation.

Dependencies between classes should be based on abstractions, not on concrete classes. This allows for maximum flexibility.

When classes rely on one another, you want to minimize the dependency - we say that you want *loose coupling* between the classes. Do do this, you 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.



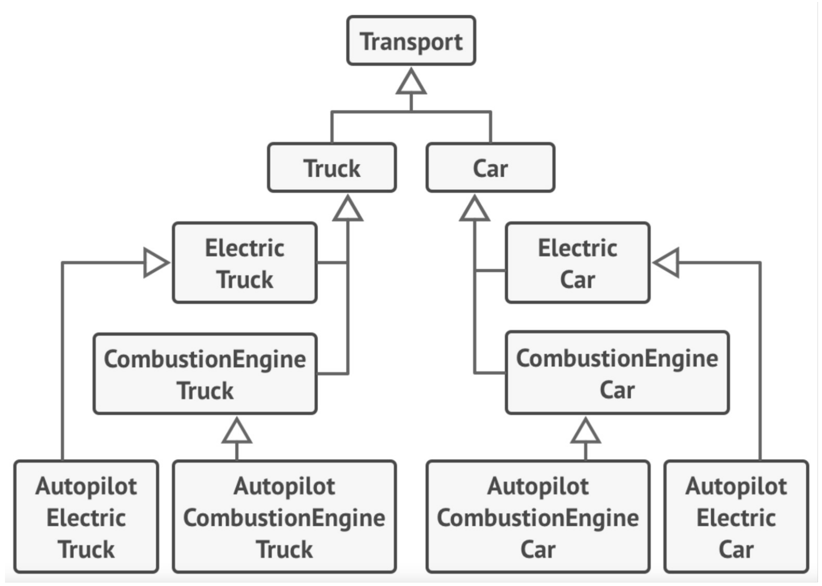
FAVOR COMPOSITION (FLEXIBILITY)

Favour Composition over Inheritance

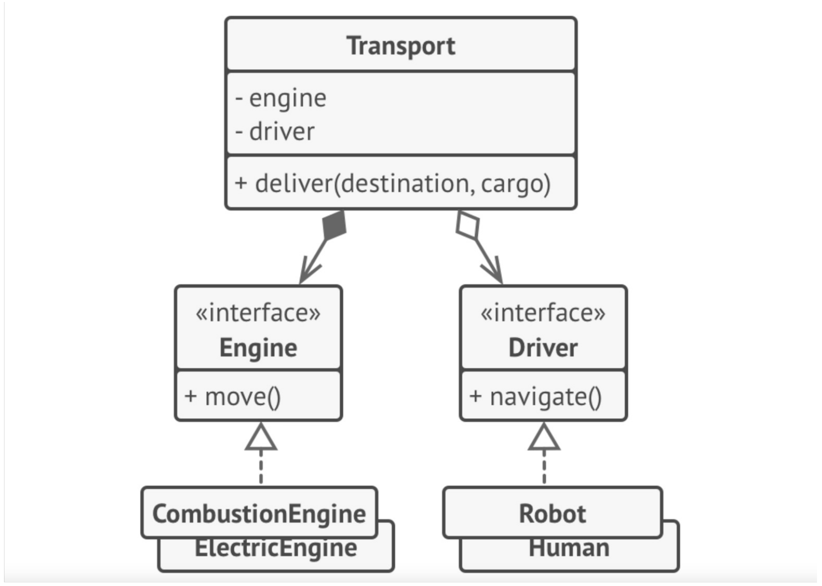
Inheritance is a useful tool for reusing code. In principle, it sounds great - derive from a base class, and you get behaviour for free! However, there can be negative side effects to inheritance.

- **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).

Reuse known-working code

- This doesn't just mean code that you write….
- Use tested libraries when possible.

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

SOLID PRINCIPLES

CS 346: Application Development

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”)

The goal of the principles 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.

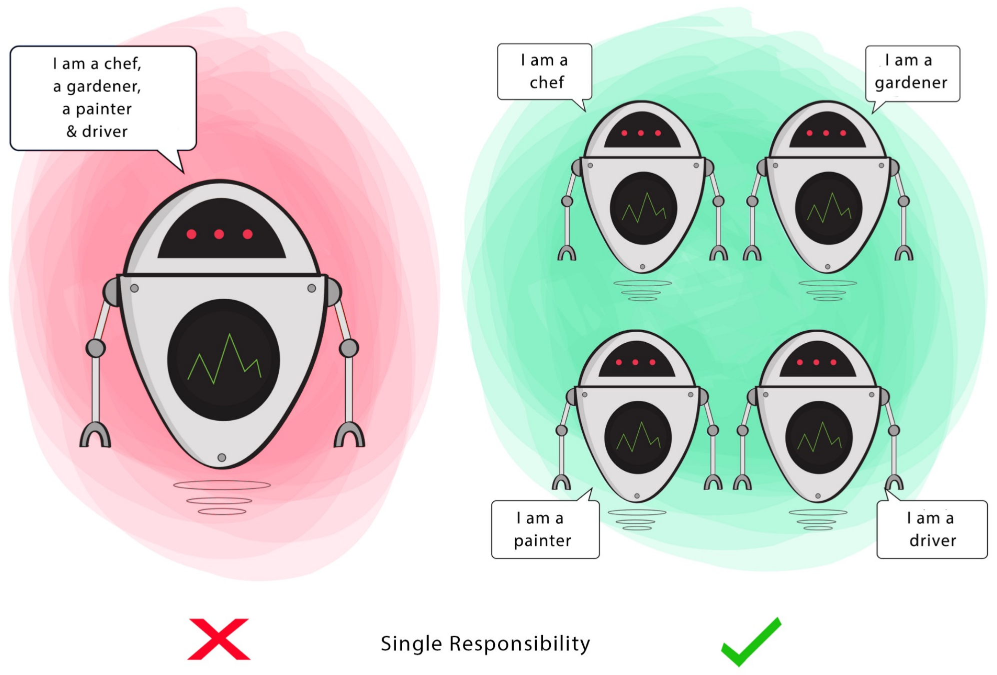
- Diagrams are taken from Ugonna Thelma: The S.O.L.I.D. Principles in Pictures.

1. SINGLE RESPONSIBILITY

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

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

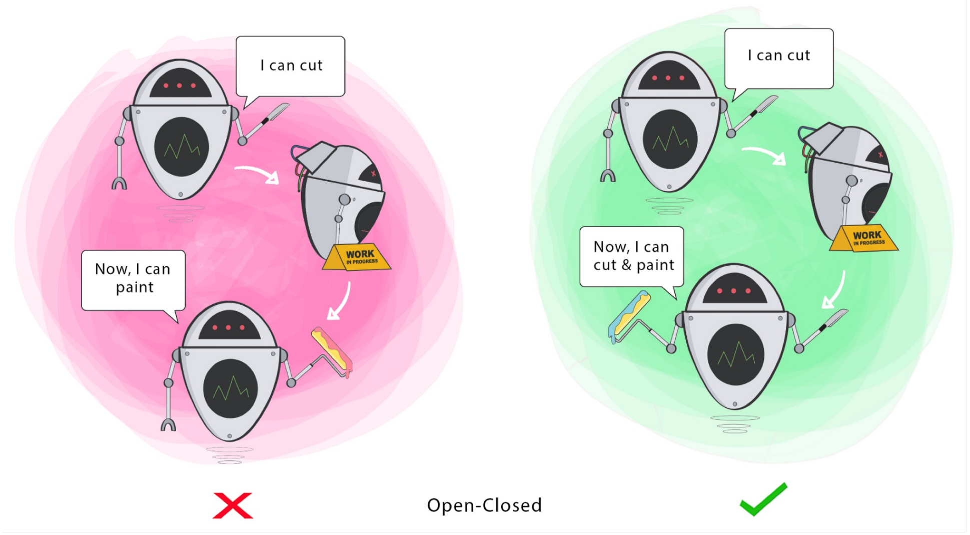
- A class has responsibility over a single block of functionality.
- There is only one reason for a class to change.
- Applies to components, and other "units" of code, not just classes.



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.

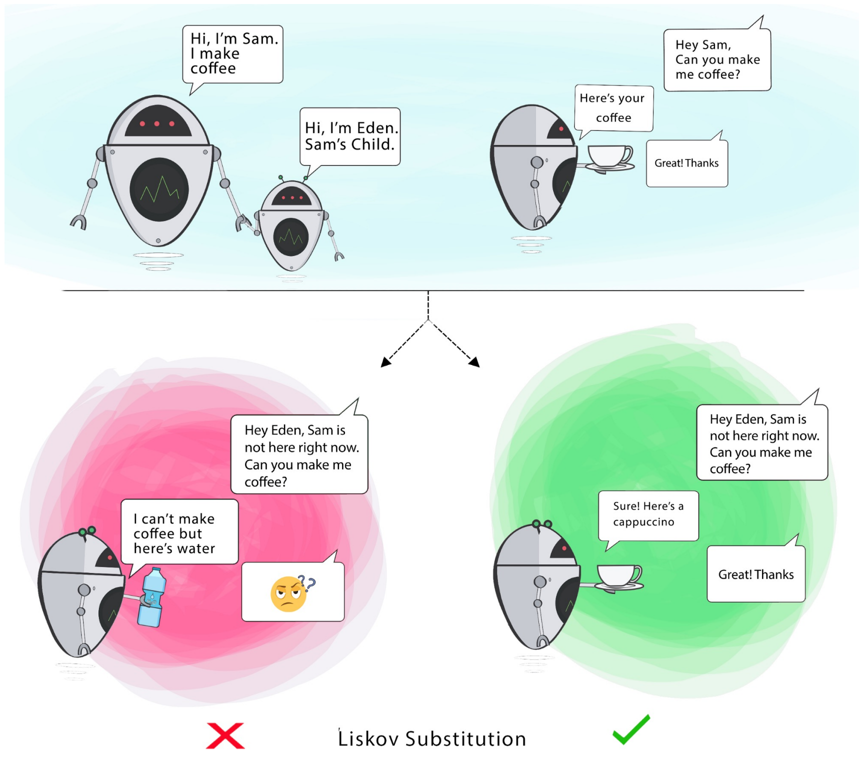
3. LISKOV-SUBSTITUTION PRINCIPLE

"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.

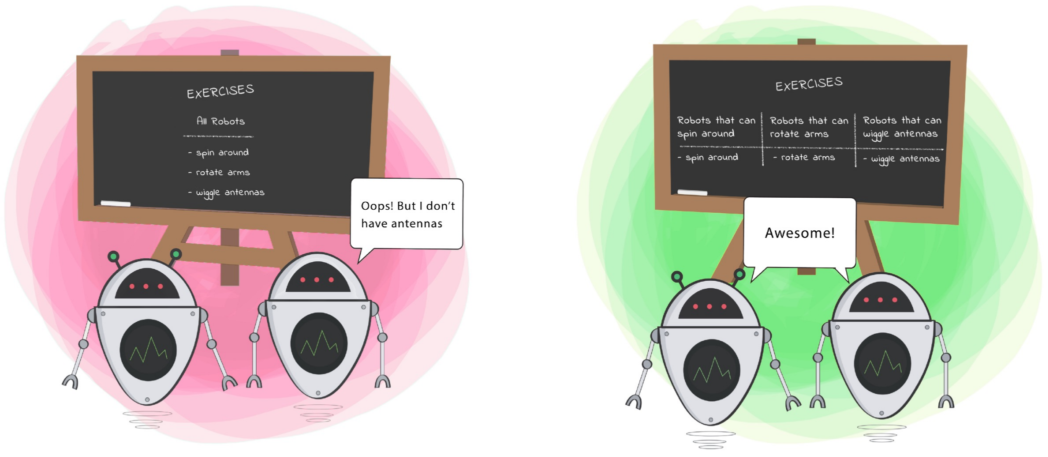

Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
Liskov Substitution

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.

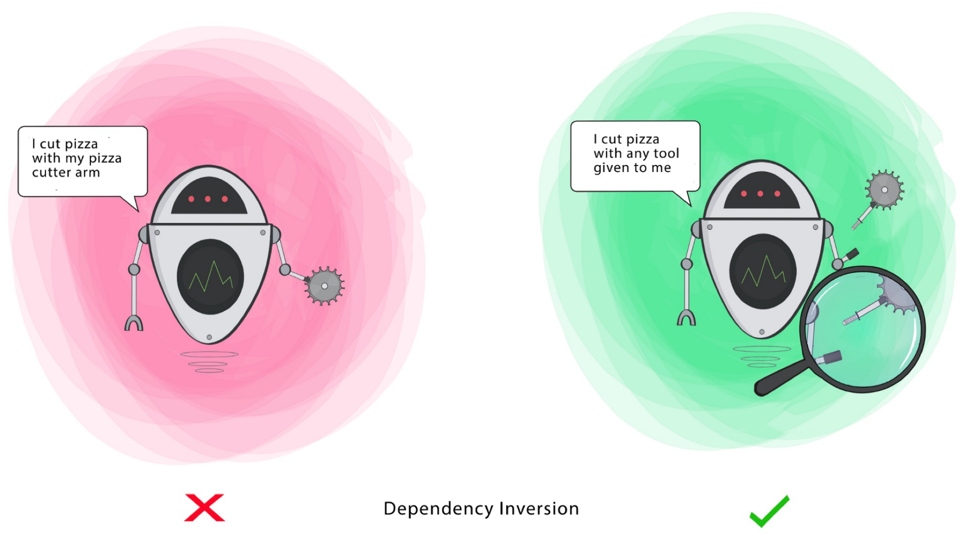


✗ Interface Segregation ✓

5. <u>D</u>EPENDENCY 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.


I cut pizza with my pizza cutter arm
I cut pizza with any tool given to me
Dependency Inversion

DESIGN PATTERNS

CS 346: Application Development

RECALL: 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].

- 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.
- 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?)

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
ADDISON-WESLEY
Dive Into
DESIGN PATTERNS
Alexander Shvets
Kotlin Design Patterns and Best Practices
Second Edition
Build scalable applications using traditional, reactive, and concurrent design patterns in Kotlin
Alexey Soshin
Foreword by Anton Arhipov, Kotlin Developer Advocate at JetBrains

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.
- Behavioural Patterns : identifying communication patterns between objects.

The expectation is that you might need a small number of these in any application. Some problems are commonly encountered (e.g. decoupling using *Observer*) and others are rarely used (e.g. *Abstract Factory*).

We'll review a few that are particularly helpful for building applications.

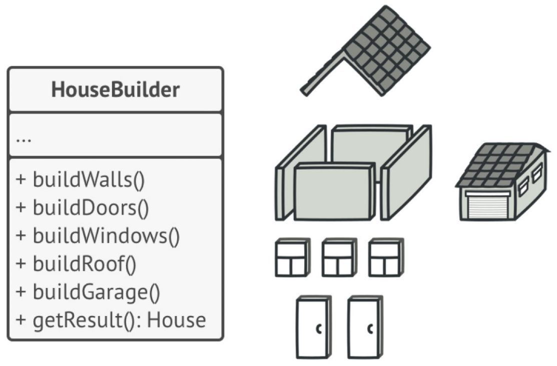
BUILDER PATTERN (CREATIONAL)

Problem: how do you build complex objects with multiple (optional) initialization steps?

Builder 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.

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 EXAMPLE

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

Java requires the Builder pattern because it doesn't support default arguments. You need to be able to call setters first, in order, and then indicate when you are "done".

```
class AlertDialog(
   var title= File Save Error",
   var text = "Error. Continue?",
   var icon = ERROR_ICON,
   var type = YES_NO_BUTTONS
)
val dialog = AlertDialog(title = "Error")
dialog.show()
```

This is much simpler in Kotlin, since we can define defaults for every parameter. When you construct your dialog, just override the properties that need to change.

No builder required!

SINGLETON PATTERN (CREATIONAL)

Problem: 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?

1. 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.
2. Provides a global access point to that instance. Just like a global variable, the Sin-gleton 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

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

1. **Make the default constructor private,** to prevent other objects from creating an instance directly.
2. **Create a static method** that acts as a constructor. This method ensures that the object is only instantiated once.
3. 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
  }
}

var s = Singleton.getInstance()
```

Java singleton implementation

SINGLETON EXAMPLE

In Kotlin, implementation is *much* easier.

The ‘object’ keyword in Kotlin defines a static instance of a class. Effectively, **an object <u>is</u> 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.
- You cannot instantiate it (compile error).

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


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

Kotlin singleton implementation

FACTORY METHOD (CREATIONAL)

Problem: You don't know which subclass to create (i.e. need to defer to *runtime*).

The **Factory Method** defines a way to decide which subclasses to instantiate by defining method whose role is to examine conditions and make that decision for the caller. i.e. instantiation is deferred to a Factory Method.

How to use it?

1. Create a class hierarchy for the classes that you need to instantiate, including a base class (or interface) and all subclasses.
2. Create a Factory class that instantiates and returns the correct subclass.

FACTORY METHOD EXAMPLE

```
sealed class Piece(val position: String)                           // base class
class Pawn(position: String) : Piece(position)                     // derived classes
class Queen(position: String) : Piece(position)


fun generatePieces(notation: List<String>): List<Piece> {          // factory method
    return notation.map { piece ->
        val pieceType = piece.get(0)
        val position = piece.drop(1)
        when(pieceType) {
            'p' -> Pawn(position)
            'q' -> Queen(position)
            else -> error("Unknown piece")
        }
    }
}


val notation = listOf("pa3", “qc5")                                 // method returns list of pieces
val list = generatePieces(notation)
```

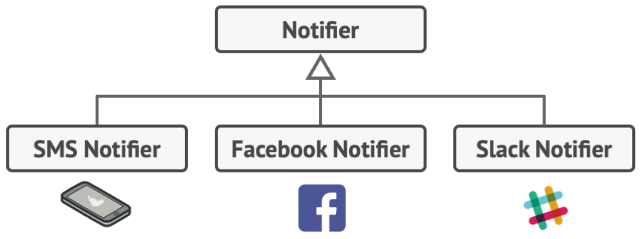
DECORATOR (STRUCTURAL)

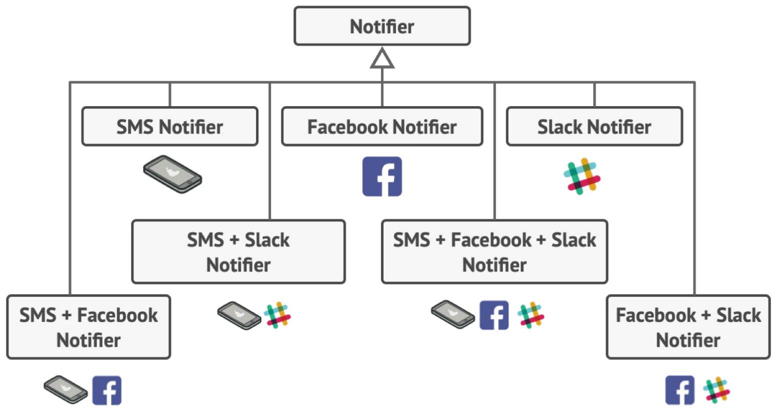
Problem: You want to describe combinations of parameters or classes, without the complexity of having a subclass for each combination.

A **decorator** allows you to chain together the classes that you want to process a request.

- e.g. You are building a message notifier, which allows your application to send out messages.

It's easy to see how to create a specialized notifier, but what if you want to have a message that is sent to *multiple* notifiers at the same time?





DECORATOR EXAMPLE

```
fun main() {
    val logger = Logger()
    val cache = Cache()
    val request = Request("http://example.com")
    val response = processRequest(request, logger, cache)
    println("Results: ${response}")
}


// what if I don’t want all of these processors to run?
fun processRequest(request: Request, logger: Logger, cache: Cache): Response {
    logger.log(request.toString())
    val cached = cache.get(request) ?: run {
        val response = Response("You called ${request.endpoint}")
        cache.put(request, response)
        response
    }
    return cached
}
```

```
fun main() {
   val request = Request("http://example.com")
 val processor: Processor = LoggingProcessor(Logger, RequestProcessor()))
   println("Results: ${processor.process(request)}")
}


interface Processor { fun process(request: Request): Response }


class LoggingProcessor(val logger: Logger, val processor: Processor) : Processor {
    override fun process(request: Request): Response {
        logger.log(request.toString())                      // do appropriate work
        return processor.process(request)                   // pass to next Processor
    }
}


class RequestProcessor(): Processor {
    override fun process(request: Request): Response {
        return Response("You called ${request.endpoint}”)   // do appropriate work
    }
}
```

STRATEGY PATTERN (BEHAVIORAL)

Problem: You have behavior that isn't specified until runtime. How to you abstract this so that you can set it dynamically?

The strategy pattern is a way to swap algorithms at runtime. It's often modelled as a set of interchangeable classes.

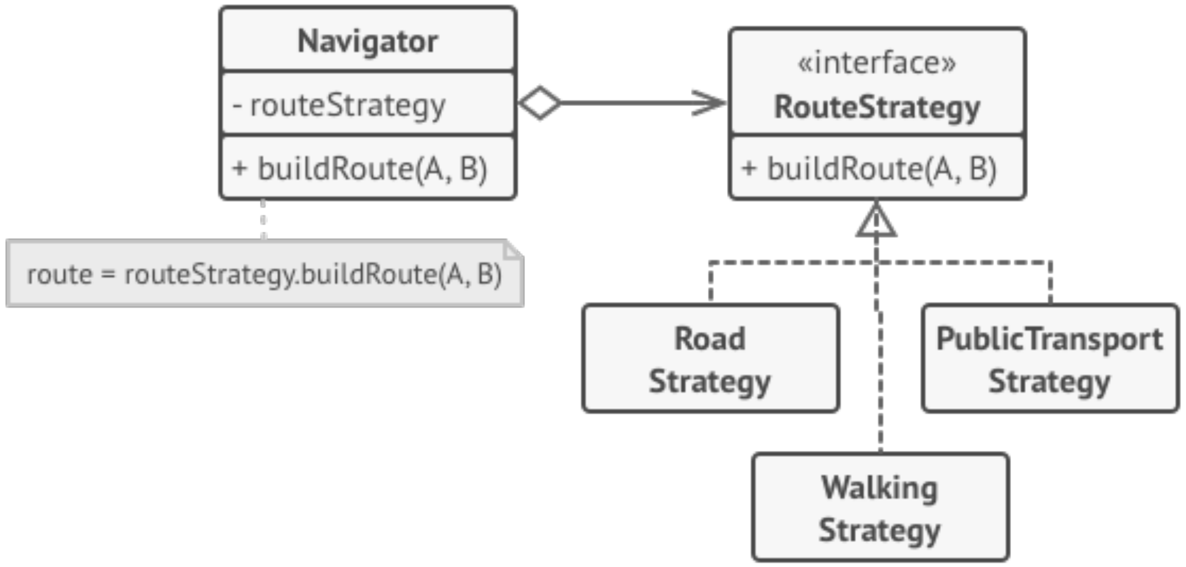
Why is this pattern useful?

- It allows you to add new algorithms or modify existing ones without modifying existing code.
- Provides extensibility and flexibility to your solution.

How does it work?

- Extract algorithms into separate classes called *strategies*.
- The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object.

STRATEGY EXAMPLE



https://refactoring.guru/design-patterns/strategy

```kotlin
interface FormField {
    val name: String
    val value: String
    fun isValid(): Boolean
}

class EmailField(override val value: String) : FormField {
    override val name = "email"
    override fun isValid(): Boolean {
        return value.contains("@") && value.contains(".")
    }
}

class UsernameField(override val value: String) : FormField {
    override val name = "username"
    override fun isValid(): Boolean {
        return value.isNotEmpty()
    }
}

class PasswordField(override val value: String) : FormField {
    override val name = "password"
    override fun isValid(): Boolean {
        return value.length >= 8
    }
}

fun main() {
    val emailForm = EmailField("nobody@email.com")
    val usernameForm = UsernameField("none")
    val passwordForm = PasswordField("*")
}
```

Kotlin

- “Standard” approach using classes
- Rigid and hard to extend e.g. add optional password field?

Idiomatic Kotlin

- Extract what changes
- Replace classes with functions/interfaces

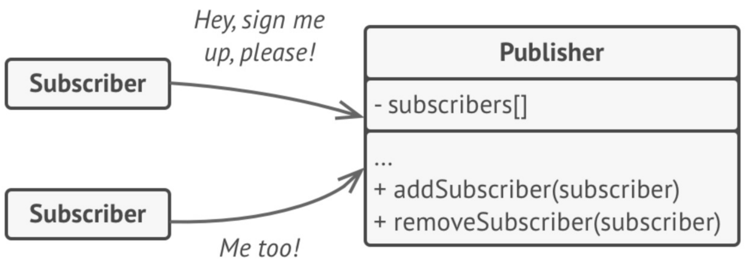
```kotlin
fun interface Validator {
    fun isValid(value: String): Boolean
}

val emailValidator = Validator { it.contains("@") && it.contains(".") }
val usernameValidator = Validator { it.isNotEmpty() }
val passwordValidator = Validator { it.length >= 8 }

class FormField(val name: String, val value: String, private val validator: Validator) {
    fun isValid(): Boolean {
        return validator.isValid(value)
    }
}

fun main() {
    val emailForm = FormField("email", "nobody@email.com", emailValidator)
    val usernameForm = FormField("username", "empty", usernameValidator)
    val passwordForm = FormField("email", "***", passwordValidator)
}
```

OBSERVER (BEHAVIOURAL)

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.

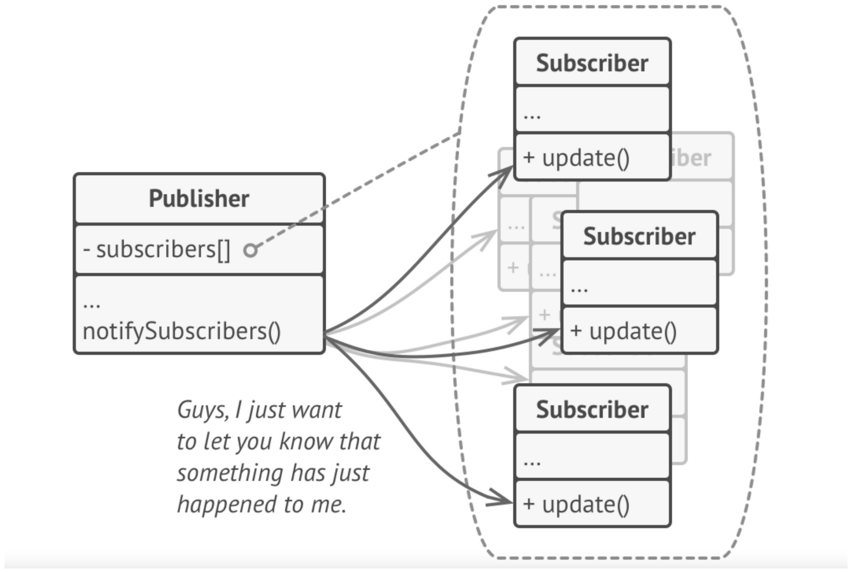
Subscribers register their interest in the subject, who adds them to an internal subscriber list.



OBSERVER (BEHAVIOURAL)

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*).



```
interface IObservable {
    val observers: ArrayList<ISubscriber>

    fun add(observer: IObserver) {
       observers.add(observer)
    }

    fun remove(observer: IObserver) {
        observers.remove(observer)
    }

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

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

```
interface ISubscriber {
    fun update()
}


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


    override run update() {
        println(“Updated”)
    }
}
```

https://www.baeldung.com/kotlin/observer-pattern