Design Principles & Design Patterns

Design Principles

Code Smells

- Bad Design and Hacks cause Code Smell
- Accumulate over time, sequence of bad decisions
- Cause the Development cycle to increase as the project size grows.
- Common solution used is to add more programmer, further increasing the Code Rot (in case engineers not properly trained).

Design Principles
Code Smell

- **Rigidity**
  - Tendency of a System to be Hard to Change
  - If a simple change in the system requires a complete rebuild
  - Result of tight coupling (dependency)
  - Increase in Development - Test cycle

Design Principles
Code Smell

- **Fragility**
  - A change in one module causes other seemingly unrelated modules to mis-behave.
  - Long Distance behavioural dependencies
    - Bug fix in one module causes completely unrelated modules to break and crash.
    - Unexpected coupling
  - Loss of trust

Design Principles
Code Smell

- **Immobility**
  - A system is immobile when it’s internal components cannot be easily extracted and re-used in new environments.
  - Caused by tightly coupled class and module design
  - Prevents reuse

Design Principles
Code Smell

- **Needless Complexity**
  - Systems that carry a lot of anticipatory design are needlessly complex.
  - Keep focus on the current set of requirements and keep the code free from code smells.

Design Principles

Universal Principles for Software Design

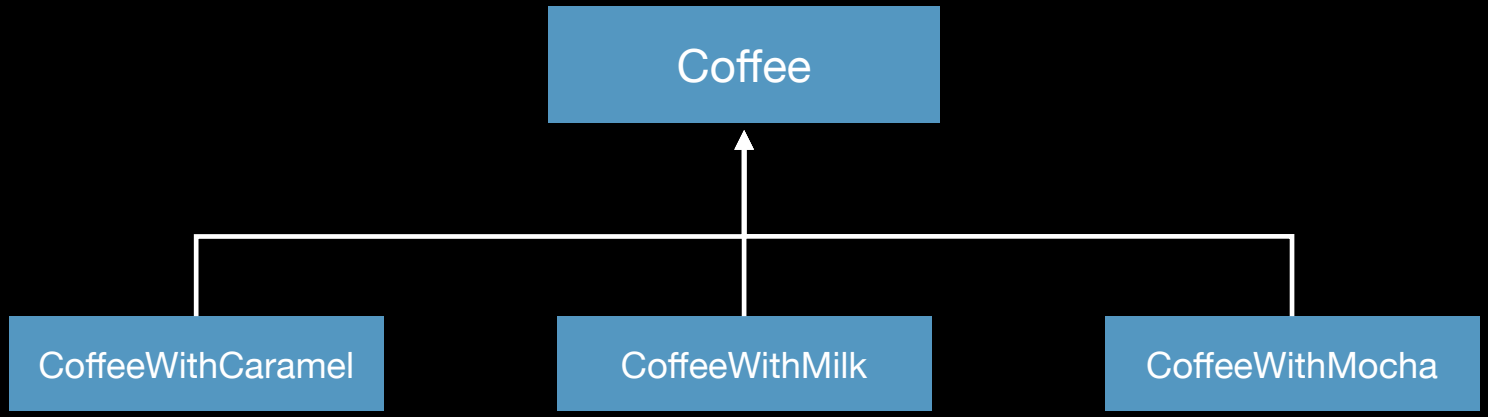
- Encapsulate What Varies
  - Identify the aspects of your application that vary and separate them from what remains same.

```
public Pancake orderPancake(String type) {
    Pancake pancake;

    if( type.equals("classic")) {
        pancake = new ClassicPancake();
    } else if (type.equals("blueberry")){
        pancake = new BlueberryPancake();
    } else if (type.equals("chcolate chip")) {
        pancake = new ChocolateChipPancake();
    }
    pancake.cook();
    pancake.plate();
    return pancake;
}
```

Design Principles

Universal Principles for Software Design

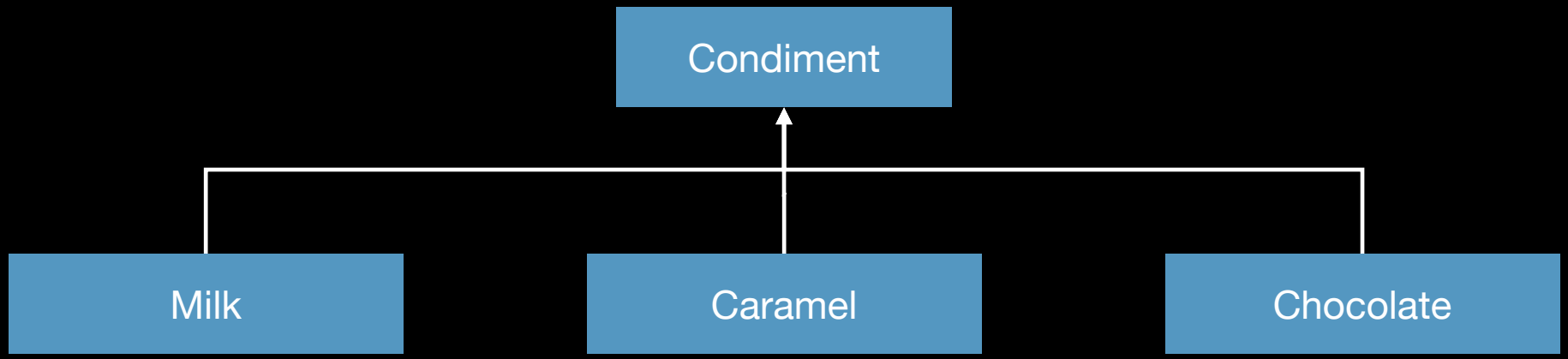
- Favor composition over Inheritance
  - Example:



Design Principles

Universal Principles for Software Design

- Favor composition over Inheritance
  - Example:



Design Principles

Universal Principles for Software Design

- Favor composition over Inheritance
  - Example:

Design Principles

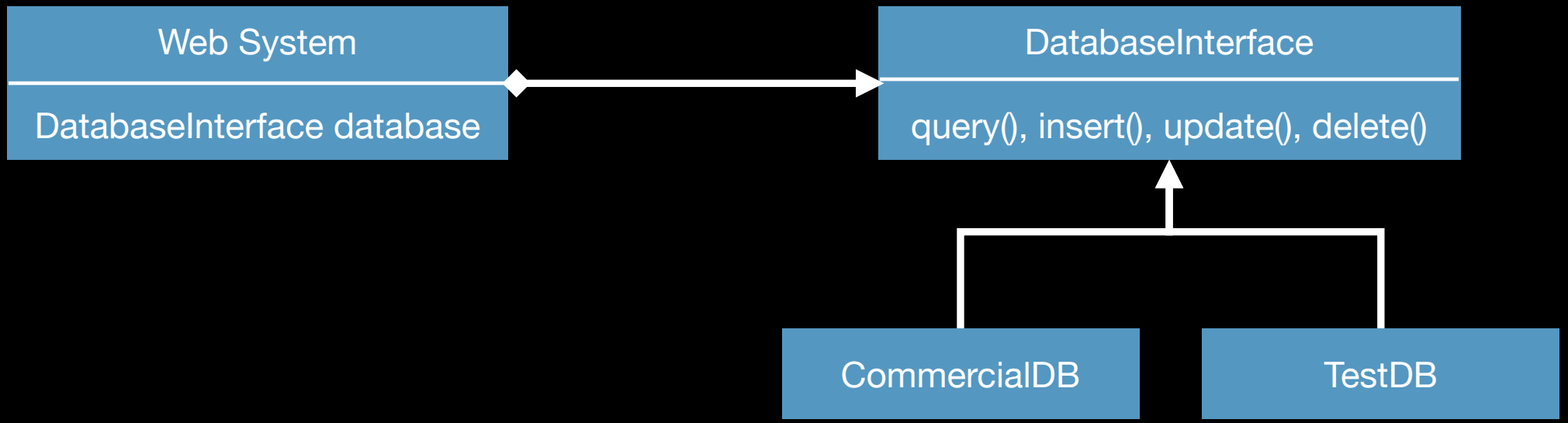
Universal Principles for Software Design

- Program to Interfaces
  - Where possible, components should use abstract types or interfaces rather than concrete types.
- Example

Design Principles

Universal Principles for Software Design

- Program to Interfaces
- Example



SOLID Principles

SOLID Principles

- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

SOLID Principles

Single Responsibility Principle

“A Function, Class should only have one Responsibility.”

- A Function, Class or Module should have one and only one reason to change.
- Examples of Responsibilities:
  - Business Logic
  - UI
  - Persistence
  - Logging

SOLID Principles

Single Responsibility Principle

- How to Identify More than one responsibility?
  - If Statements
  - Switch Statements
  - Moster Methods
  - God Class

SOLID Principles

Single Responsibility Principle

- Example of a class following SRP:

```java
public class ConsoleLogger {

    void logInfo(String message) {
        System.out.println("Info: " + message);
    }

    void logWarning(String message) {
        System.out.println("Warning: " + message);
    }

    void logError(String message) {
        System.out.println("Error: " + message);
    }
}
```

SOLID Principles

Open Closed Principle

“Functions, Classes should be closed for modification, but open for extension.”

- Closed for modification
  - Adding a new feature does not require modifying existing code.
- Open for extension
  - Component should be extendable to add new behaviors.

SOLID Principles

Liskov Substitution Principle

“If S is a sub-type of T, then objects of type T may be replaced with objects of type S, without changing the correctness of the program.”

- Need to think carefully when defining Hierarchical relationships.
- Sub-class is not always a good option
- Incorrect hierarchical relationships fail the Liskov substitution principle.

SOLID Principles

Liskov Substitution Principle

- Incorrect relationships between types cause unexpected bug or code effects.
- When thinking about hierarchical relationships, we think in terms of is-a relationship.
  - Square is a Rectangle
  - Ostrich is a Bird
- We should think in terms of substitutability
  - Is Bird object substitutable by Ostrich object?

SOLID Principles

Liskov Substitution Principle

- Violation of LSP

```java
public class Bird {
    void fly() {
        System.out.println("Flying");
    }
}
```

```java
public class Ostrich extends Bird {
    @Override
    void fly() {
        super.fly();
        //nothing to do
    }
}
```

```java
public static void main(String[] args) {
    Bird bird = new Ostrich();
    bird.fly();
}
```

SOLID Principles

Liskov Substitution Principle

- Violation of LSP
  - Type Checking

```
for(Task task : tasks) {
    if(task instanceof Bug) {
        Bug b = (Bug) task;
        b.fixed();
    }

    t.taskInProgress();
}
```

SOLID Principles

Interface Segregation Principle

“Clients should not be forced to depend on methods that they do not use”

- Split large interfaces into smaller more focused interfaces.

SOLID Principles

Interface Segregation Principle

- Violation of ISP

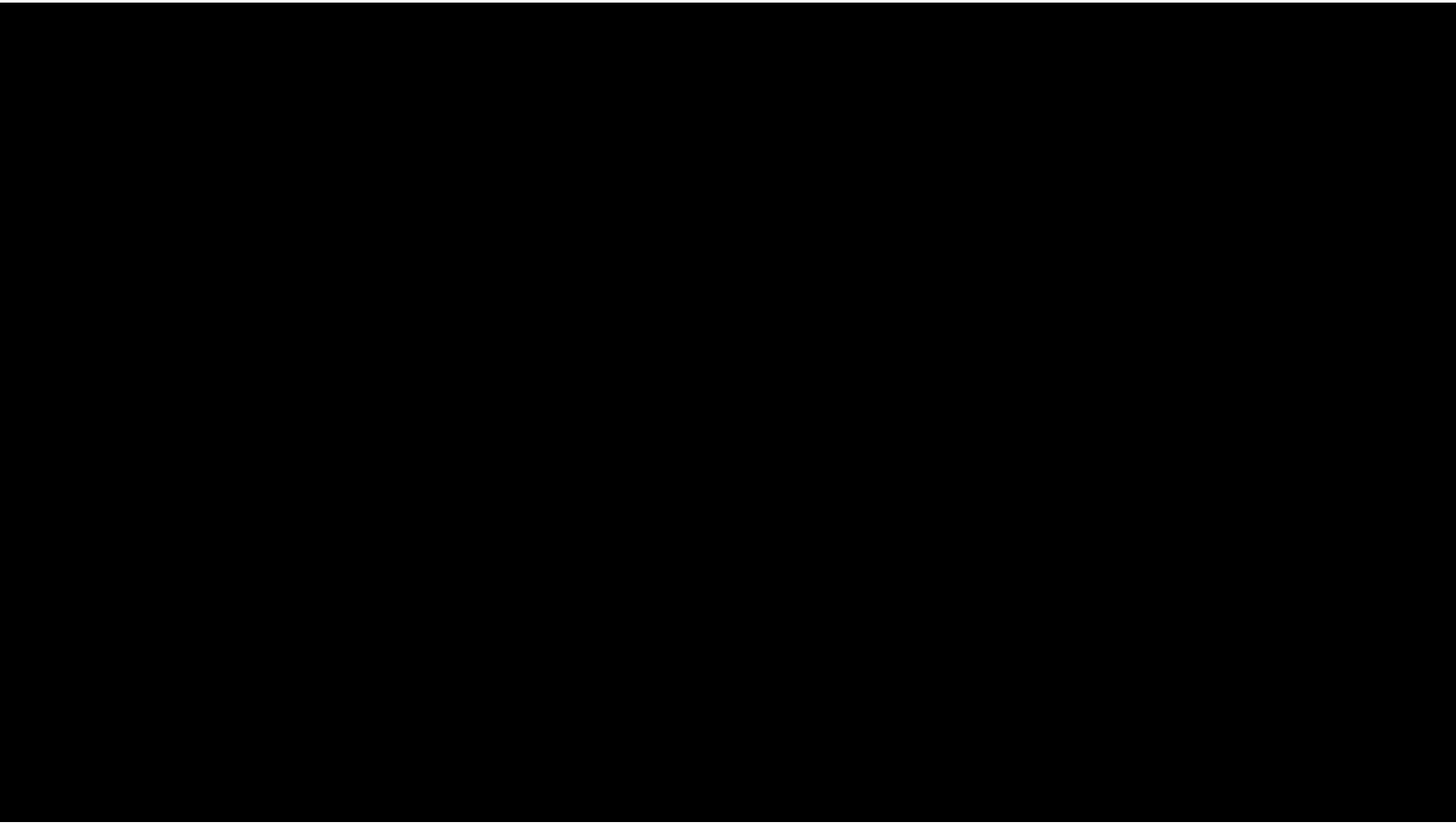
```
interface LoginService{
    void signIn();
    void signOut();
    void updateRememberMeCookie();
    User getUserDetails();
    void setSessionExpiration(int seconds);
    void validateToken(Jwt token);
    …
}
```

SOLID Principles

Interface Segregation Principle

- Violation of ISP

```
class GoogleLoginService implements LoginService{
    …
    public void updateRememberMeCookie(){
        throw new UnsupportedOperationException();
    }
    public void setSessionExpiration(int seconds){
        throw new UnsupportedOperationException();
    }
```



SOLID Principles

Dependency Inversion Principle

“High Level Modules should not depend on Low Level Modules, both should depend on Abstractions.”

- Abstrations should not depend on Details, Details should depend on Abstractions.

SOLID Principles

Dependency Inversion Principle

- **High-Level Modules**
  - Modules that contain the Business Logic
  - Specify what need to be done
- **Low-Level Modules**
  - Contain the implementation details that are required to execute the business policies.
  - Specify the How?
- **Abstrations**
  - Something that is not concrete. Eg: interfaces in Java

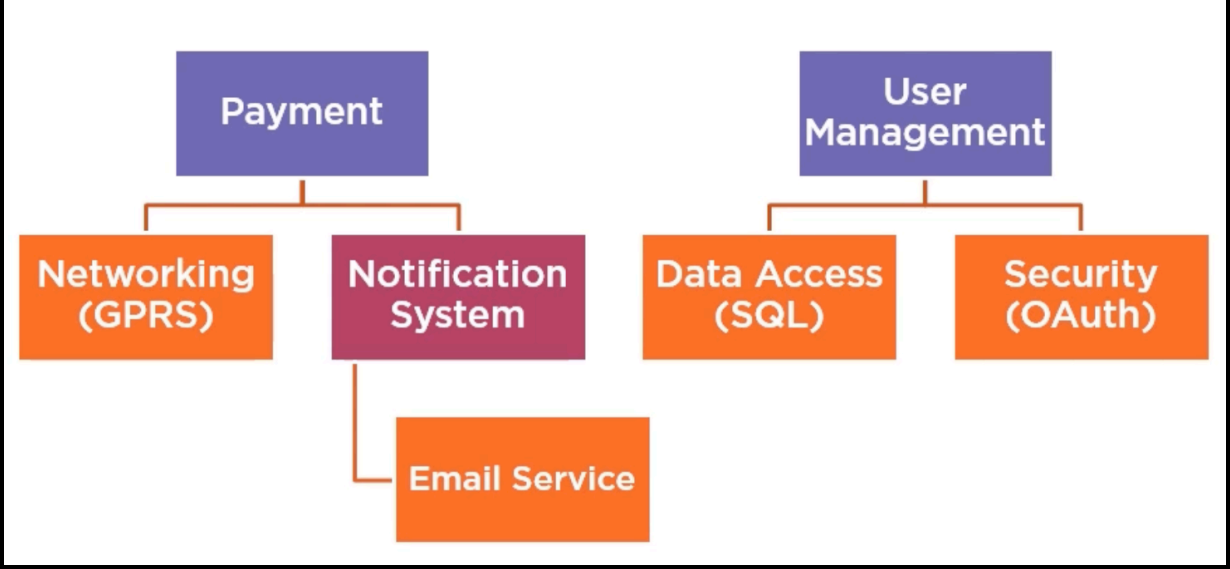
SOLID Principles
Dependency Inversion Principle

- Examples of Low Level Modules
  - Logging
  - Network IO
  - Data Access
  - Etc.

SOLID Principles

Dependency Inversion Principle

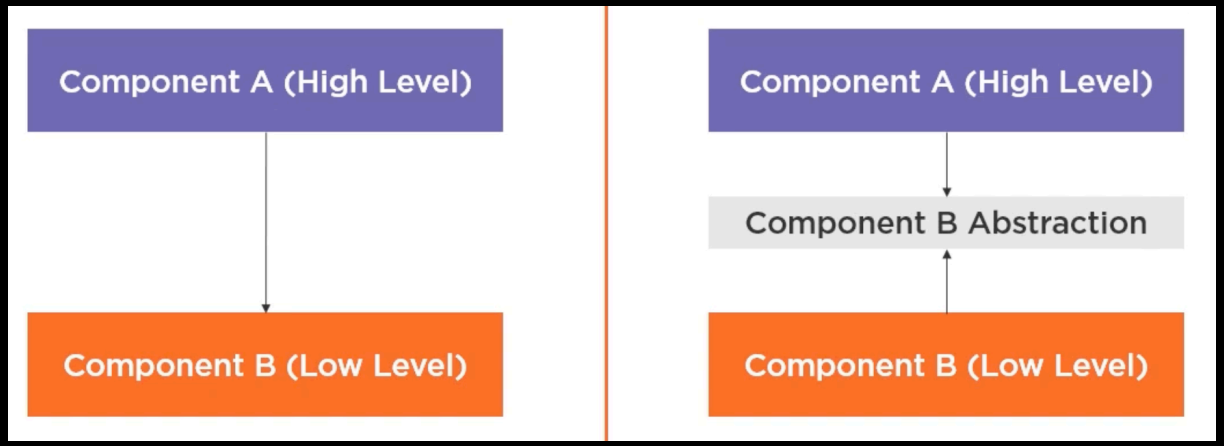
- High Level Modules work with Low Level Modules to provide functionality



SOLID Principles

Dependency Inversion Principle

- High Level Modules should not depend on Low Level Modules



Design Patterns with Java

Design Patterns

What are Design Patterns?

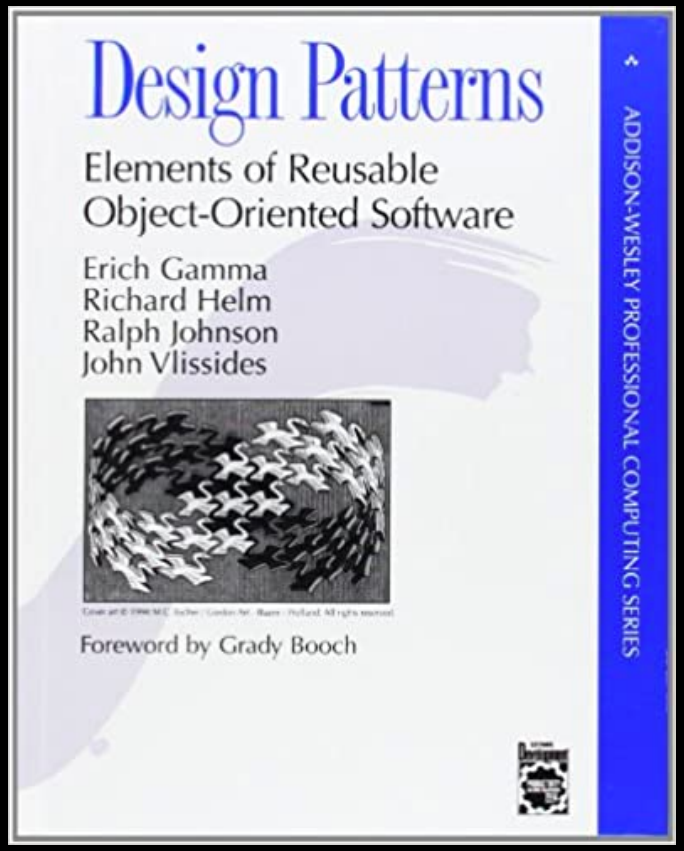
- Strategy to deal with a common (recurring) software development problem.
  - How to break down a solution into ideal parts?
  - How to structure the various parts?
  - Common Idioms
- In Modern times, focus has been Object Oriented Design

Design Patterns

- Model View Controller (MVC)
  - Late 1970s
  - Xerox, Palo Alto
- Model View Presenter (MVP)
  - Early 1990s

Basics

GOF Book



Design Patterns

- Solution to common problems related to
  - Architecture (Organising a solution into the different parts)
  - Creating Objects
  - Communication between Objects
  - Adding behaviour to Classes
- Language Agnostic

Classicification

- Creational Design Patterns
  - Singleton
  - Prototype
  - Factory Method
  - Abstract Factory
  - Builder

Classicification

- Structural Design Patterns

  “Define and manage relationship between objects”

  - Adapter
  - Decorator
  - Facade
  - Flyweight
  - Proxy

Classicification

- Behavioral Design Patterns

“Define and manage relationship between objects”

- Observer
- Strategy
- State

Singleton

Creational Design Patterns

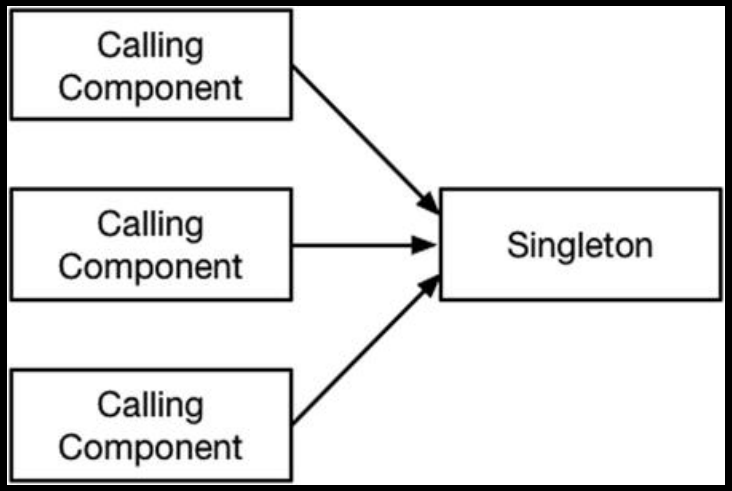
Singleton

- **Problem**
  - Access and Manage a Single Resource
- **Solution**
  - Disallow multiple instances of a class
  - Provide controlled access to the single instance

Creational Design Patterns

Singleton Pattern

- **Solution**: Singleton Pattern
- Ensures that only one object of a given type exists in the application
- Provide a global point of access to the single object.



Creational Design Patterns
Singleton Pattern

- **Implementation**:
- Make the initialisers of the class private
- Create a single instance of the object and assign it as a static property of class. (shared, default)

Builder

Creational Design Patterns

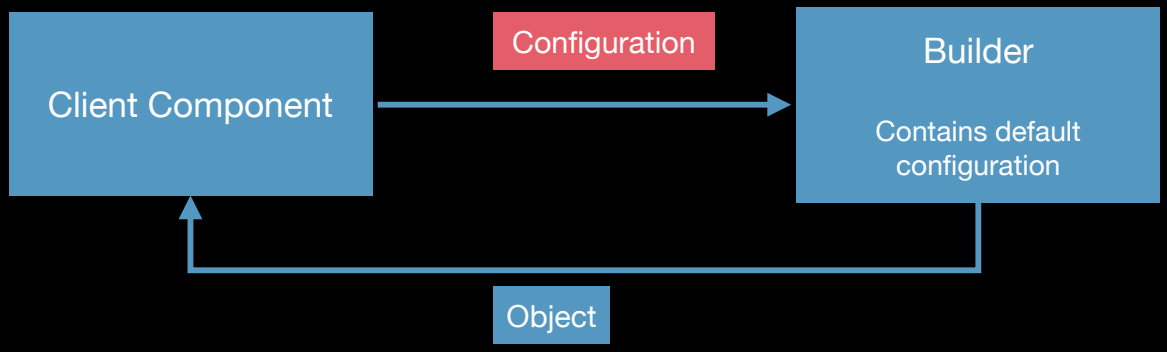
Builder Pattern

- **Problem**: Complex configuration process is required to create an object, because of the large number of values to be provided.

Creational Design Patterns

Builder Pattern

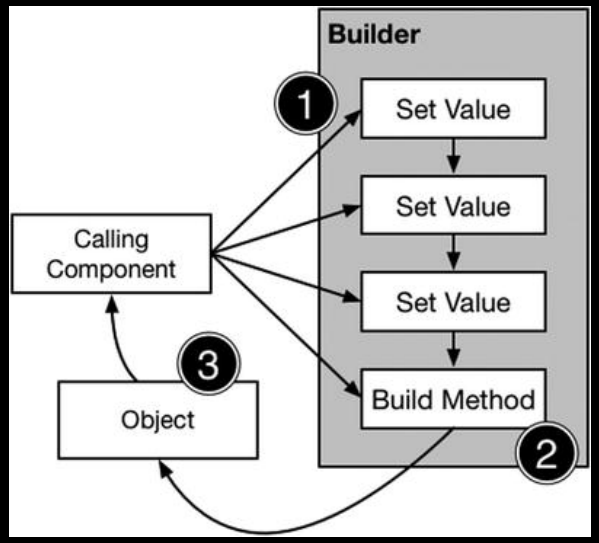
- “Separate the configuration of an object from its creation.”
- Calling component passes the configuration data to an intermediary—the builder—that is responsible for creating an object.
- Reduce the amount of knowledge that the calling component has about the objects it uses



Creational Design Patterns

Builder Pattern

- **Solution**: Builder Pattern
- Introduces an intermediary—called the Builder—between a component and the object it needs to work with the Builder to create the object.



Creational Design Patterns

Builder Pattern

- **BENEFITS**
- Change a default value in the Builder without having to make changes to the calling component or to the class.
- Change the way an object is instantiated without needing to make changes in the Builder or to the class.
- Change in the class can be absorbed in the Builder class so that it doesn’t propagate to the calling components.

Prototype

Creational Design Patterns

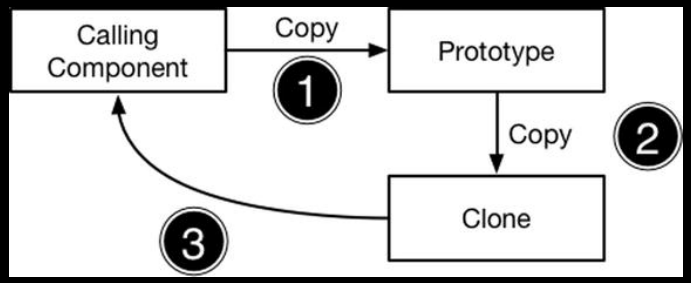
Prototype

- **Problem**: Creating a new object is expensive

Creational Design Patterns

Prototype

- **Solution**: Use an existing object to create new objects. This is often referred to as cloning an object.
- The existing object is referred to as the prototype



Prototype
- clone(): Prototype

Factory Pattern

Creational Design Patterns

Factory Pattern

- **Problem**: Object creation should not be coupled with business logic

```
interface RentalCar {
    String getName();
    int getNumberOfPassengers();
    float getPricePerDay();
}
```

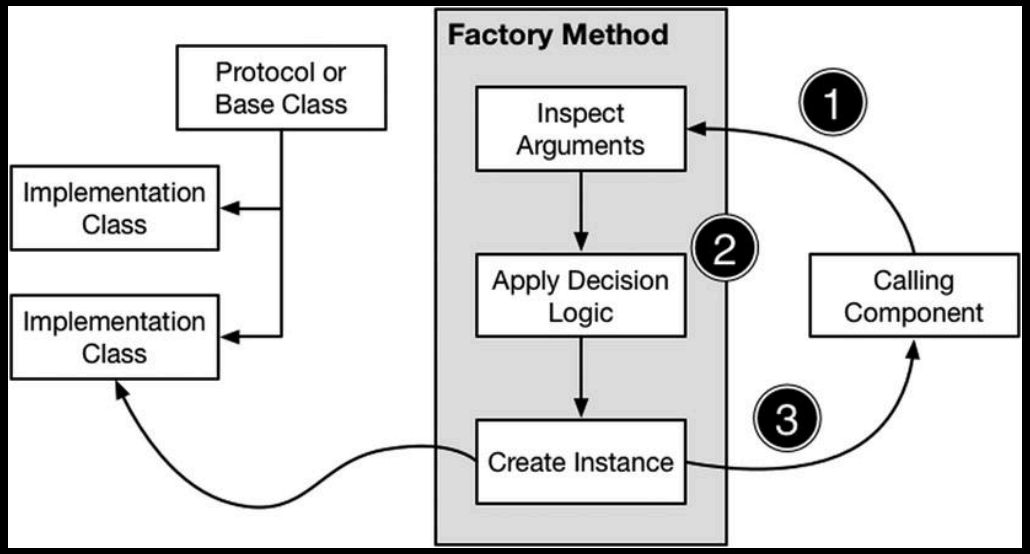
```
class Compact implements RentalCar

class Sports implements RentalCar

class SUV implements RentalCar
```

```
for (int count : passengersCount) {
    RentalCar car = null;
    if (count == 1) {
        car = new Sports();
        System.out.println("For " + count + " passengers we recommend " + car.getName());
    } else if (count >= 2 && count <= 3) {
        car = new Compact();
       System.out.println("For " + count + " passengers we recommend " + car.getName());
        break;
    } else if (count >= 4 && count <= 7) {
        car = new SUV();
        System.out.println("For " + count + " passengers we recommend " + car.getName());
    } else {
            System.out.println("For " + count + " passengers we don't have a car");
    }
}
```

Creational Design Patterns

Factory Pattern

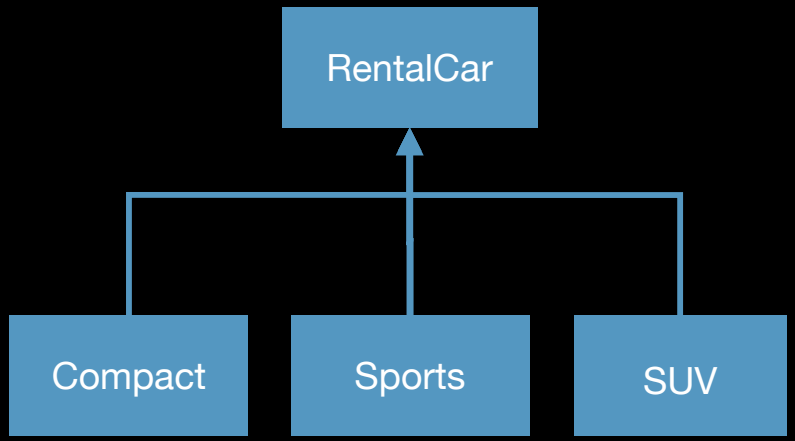
- **Solution**: Factory Pattern
- A method encapsulates the logic required to select an implementation class that is accessible to calling components.



Creational Design Patterns

Factory Pattern

- **Pre-Requisites**: Types that share a common base class or conform to the same interface.



Creational Design Patterns

Factory Pattern

- **Implementation**: (Factory as Type)

```
interface RentalCarFactory {
    RentalCar createRentalCar(int passengerCount);
}
```

```
class MyRentalCarFactory implements RentalCarFactory {
    public RentalCar createRentalCar(int passengerCount) {
        if (passengerCount == 1) {
            return new Sports();
        } else if (passengerCount >= 2 && passengerCount <= 3) {
            return new Compact();
        } else if (passengerCount >= 4) {
            return new SUV();
        }
        return null;
    }
}
```

Abstract Factory

Creational Design Patterns

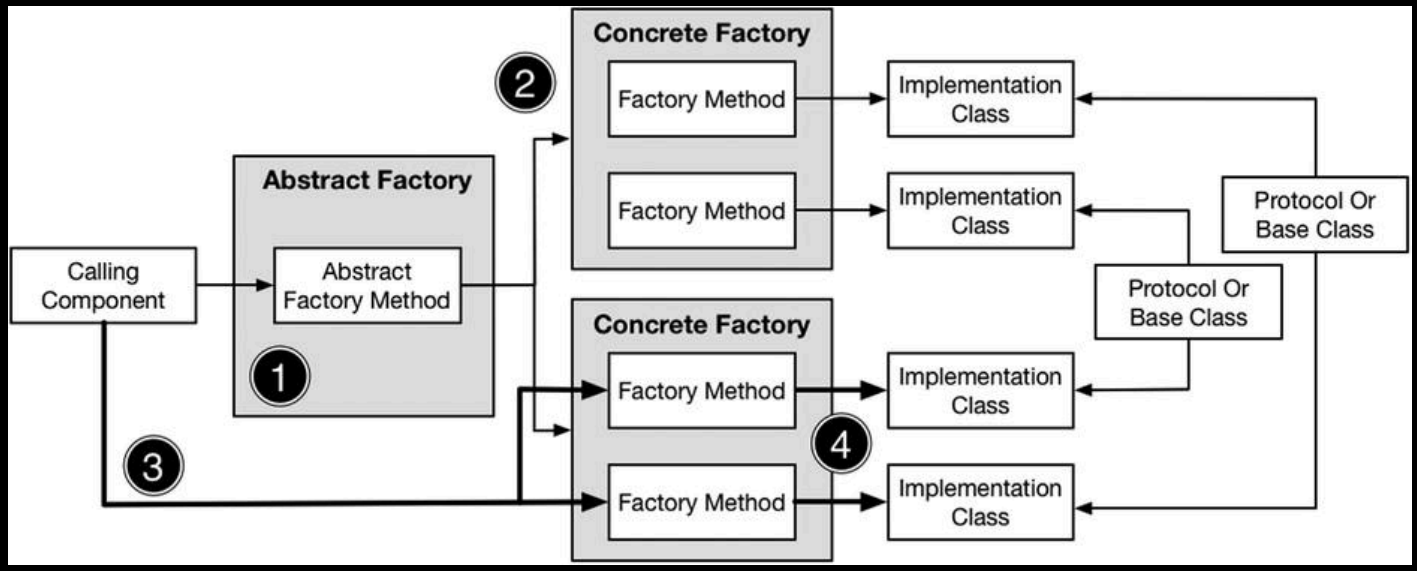
Abstract Factory

- **Problem**: Object creation requires multiple other related types, and it should not be coupled with business logic

Creational Design Patterns

Abstract Factory Pattern

- **Solution**: Abstract Factory Pattern
- Removes the dependency on concrete types, and provides protocol based access to concrete objects



Object Pool

Creational Design Patterns
Object Pool Pattern

- **Problem**: Restrict the number of instances of a type to be of a limited number

Creational Design Patterns

Object Pool Pattern

- **Solution**: Object Pool Pattern
- Create and manages a finite collection of objects, known as the object pool.