Design Patterns in Java: The Big Picture

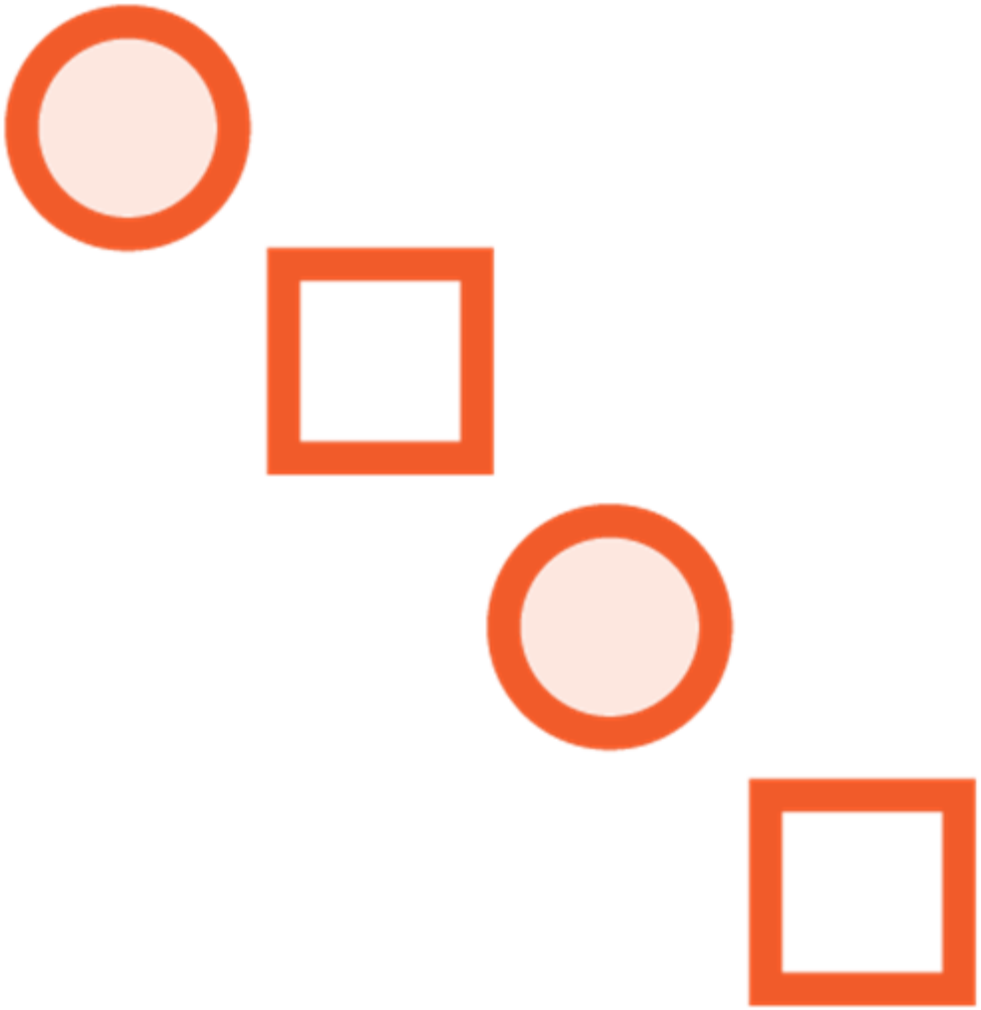
UNDERSTANDING DESIGN PATTERNS

Esteban Herrera
JAVA ARCHITECT

@eh3rrera www.eherrera.net

Problem / Solution

Patterns



Patterns

Design Patterns

Design Patterns
Gang of Four

Overview

Understanding design patterns

Why are design patterns important?

Getting to know:

- Behavioral design patterns
- Creational design patterns
- Structural design patterns

Exploring other design patterns in Java

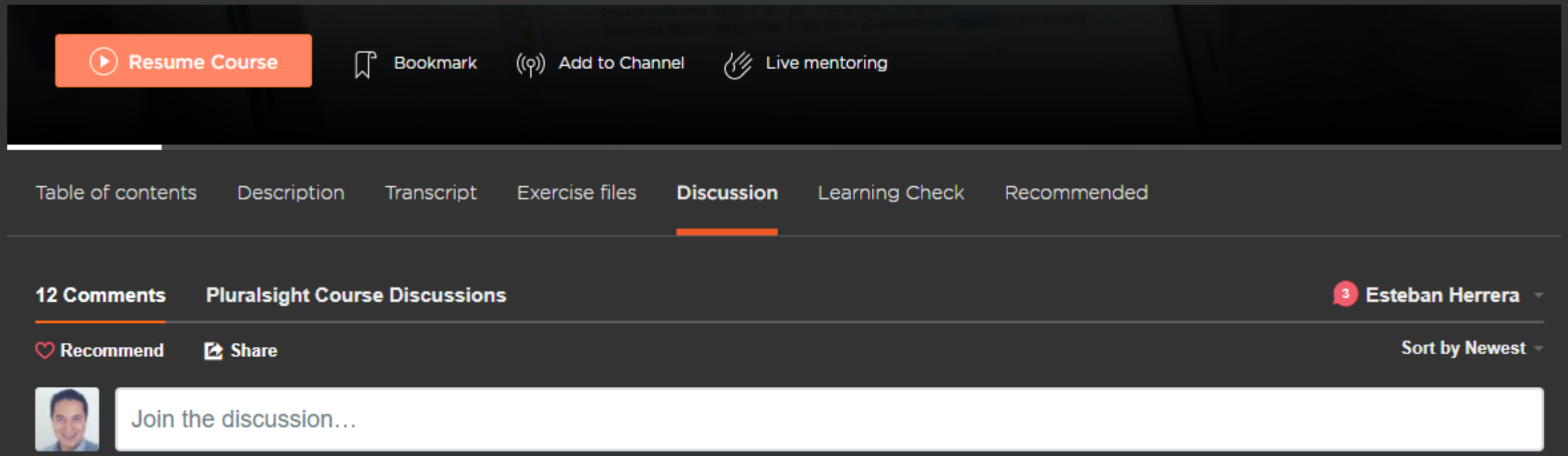
Where to go from here

Audience

Unfamiliar with design patterns

Beginner's perspective

Ask Questions



What Are Design Patterns?

Christopher Alexander

A Pattern Language: Towns, Buildings, Construction

(1977)

A Timeless Way of Building

(1979)

The Gang of Four

Erich Gamma

Richard Helm

Ralph Johnson

John Vlissides

The Definitive Design Patterns Book

Design Patterns: Elements of Reusable Object-Oriented Software

(1995)

Facade Visitor Strategy Proxy

Decorator Chain of Responsibility

Observer Abstract Factory Bridge

Builder Composite Factory Method

Command Interpreter State Mediator

Adapter Prototype Iterator Memento

Singleton Flyweight Template Method

Design Pattern

~~A solution to a problem in a context.~~

Decomposing the Definition

Context

Problem

Solution

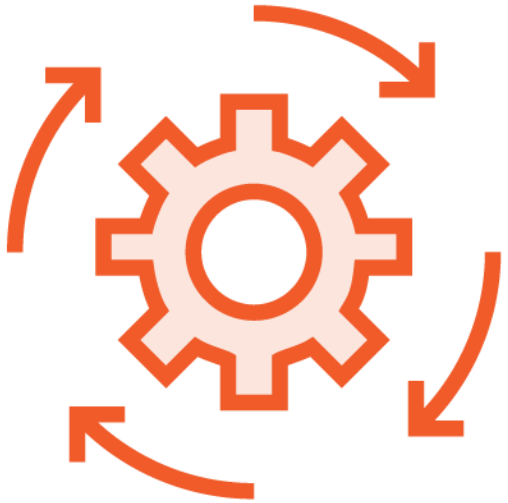
An Example of a Pattern?

- I'm in a restaurant, ready to ask for the check
- I forgot my wallet at my house
- Run out of the restaurant while the waiters are not watching

A Bad Example of a Pattern

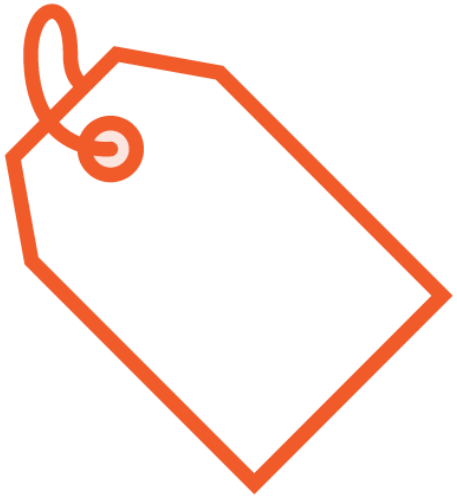
- I'm in a restaurant, ready to ask for the check
- I forgot my wallet at my house
- Run out of the restaurant while the waiters are not watching

Important Aspects of Patterns



Recurring problem

Reusable solution



Name

Design Pattern

A reusable and named solution to a recurring problem in a context.

Patterns are discovered,
not created.

System Architecture

Concurrency

Functional Programming

Object-Oriented Programming

Enterprise

Security

Business Process

User Interface Design

Object-oriented Programming Building Blocks

OOP Building Blocks

Abstraction

Encapsulation

Inheritance

Polymorphism

Abstraction

Essential details

Hiding complexity

In Java:

- Interfaces
- Abstract classes

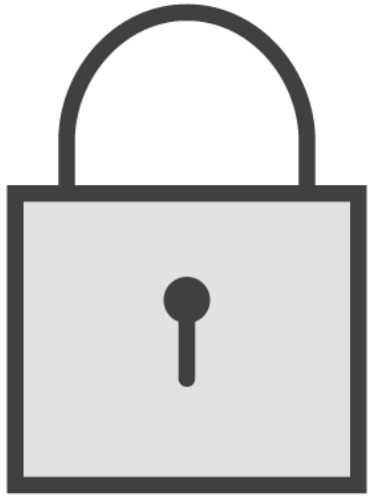
Interface Abstraction

```
public interface Engine {
    void start();
    void stop();
}
```

Higher-level Abstraction

```
public abstract class Car {
    private Engine engine;
    public void drive() {
        engine.start();
        // ....
    }
    public abstract void accelerate();
}
```

Encapsulation



Information hiding

Separate behavior that changes

Data Encapsulation

```
public abstract class Car {
    private Engine engine;
    // ....
    public setEngine(Engine engine) {
        // ....
    }
}
```

Behavior Encapsulation

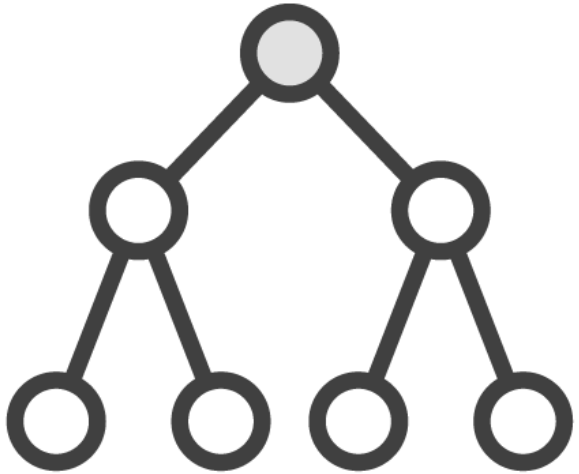
```
public class Car {
  // ....
  public void checkEngine() {
    // ....
  }
}
```

```
class EngineChecker {
  public void checkEngine() {
    // ....
  }
}
```

Behavior Encapsulation

```
public class Car {
  // ....
  public void setCheckEngine(EngineChecker ec) {
    // ....
  }
}
```

Inheritance



Inherit from another class:

- Methods
- Properties

Avoid duplicated code

In Java:

- Extending a class
- Implementing an interface

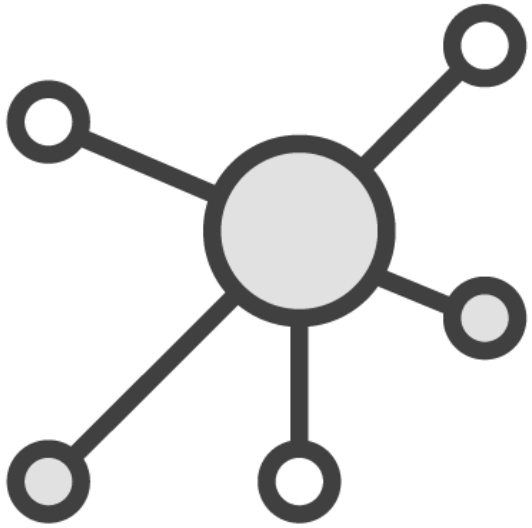
Inheritance

```
class EngineChecker {
    // ....
    public checkEngine() {
        // ....
    }
}
```

```
class FourCylinderEngineChecker
                extends EngineChecker {
   public checkEngine() {
       // Redefine method
   }
}
```

```
class V6EngineChecker
                extends EngineChecker {
   public checkEngine() {
       // Redefine method
   }
}
```

Polymorphism



Subclass stands in for the superclass

Actual object type is decided at runtime

Polymorphism

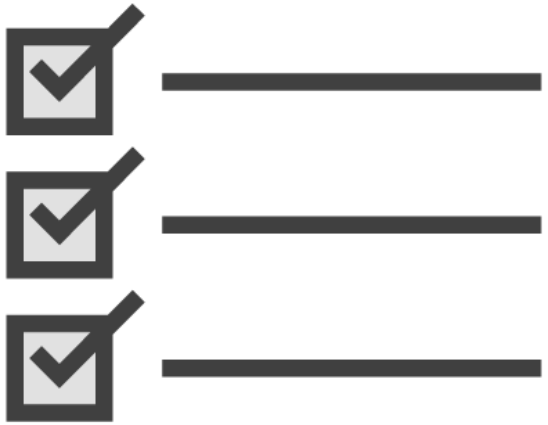
```
Engine engineChecker1 = new EngineChecker();
Engine engineChecker2 = new V6EngineChecker();

// Call V6EngineChecker's method
engineCheker2.checkEngine();
```

Object-oriented Programming Principles

Change

SOLID Principles



Single responsibility

Open-closed

Liskov substitution

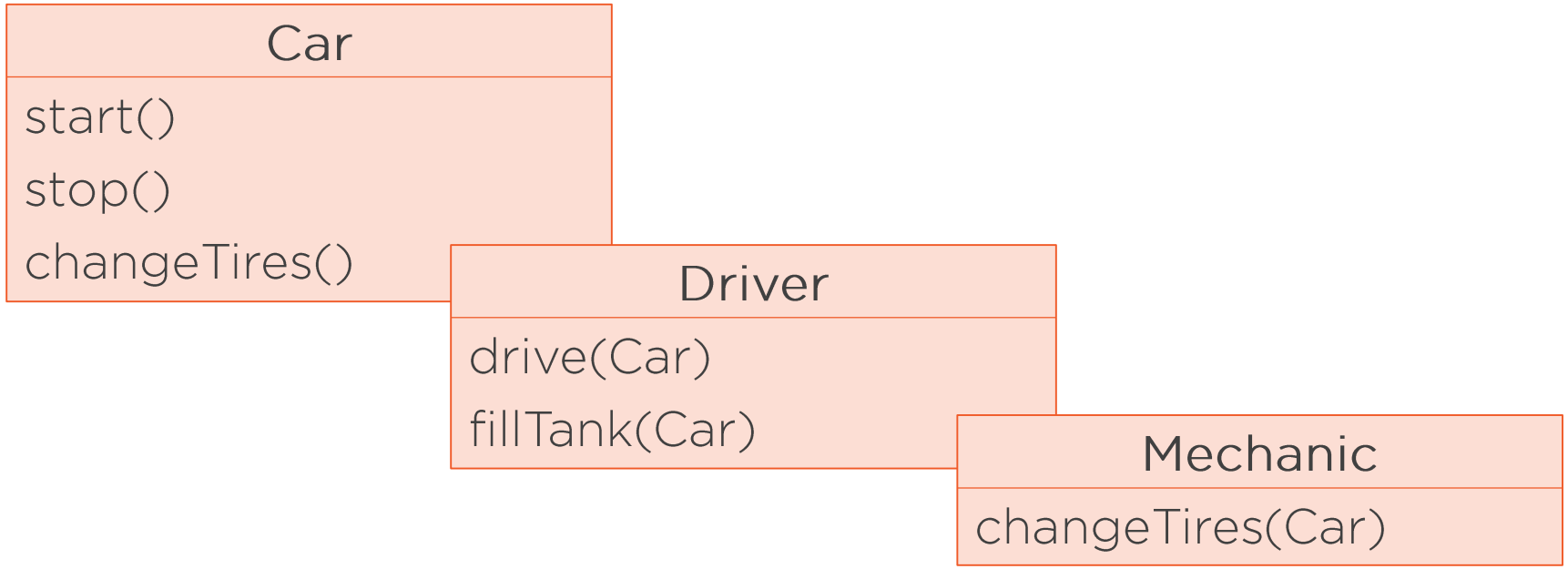
Interface segregation

Dependency injection

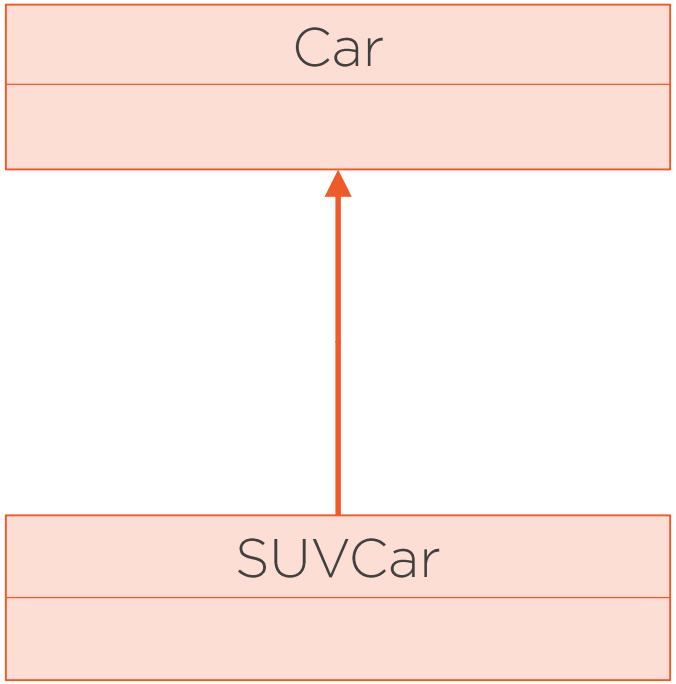
Single Responsibility Principle

Car
drive() fillTank() changeTires()

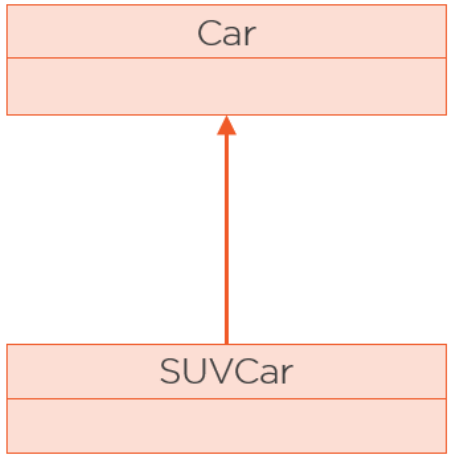
Single Responsibility Principle



Open-closed Principle



Liskov Substitution Principle

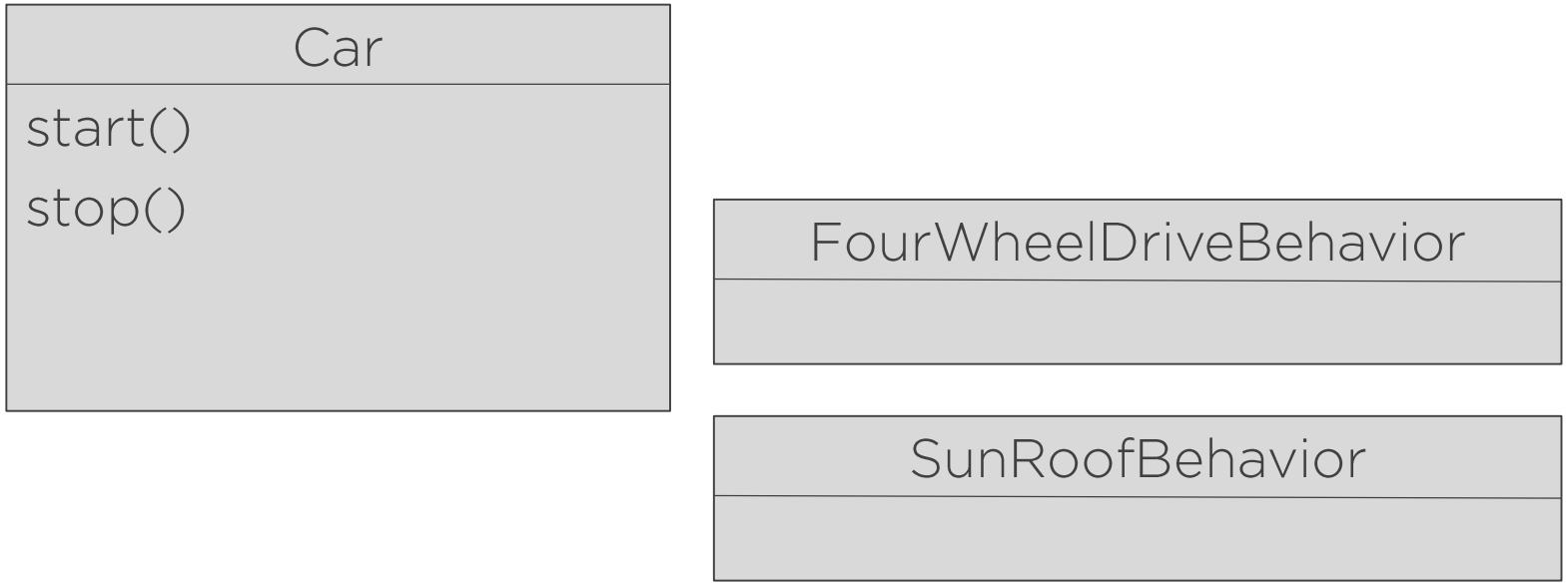


Can you actually use a SUVCar class instead of the more generic Car class?

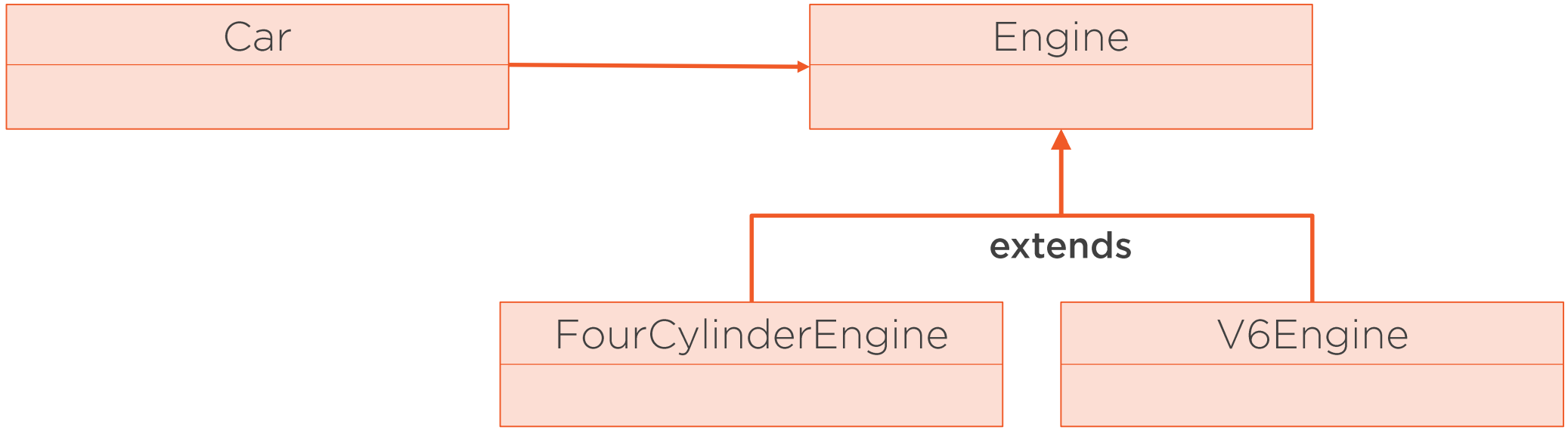
Interface Segregation Principle

Car
start() stop() activate4WheelDrive() openSunRoof()

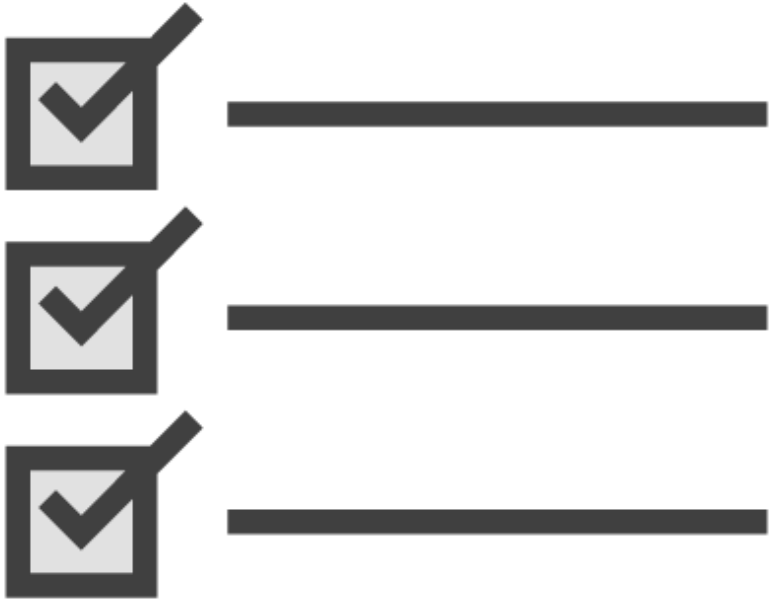
Interface Segregation Principle



Dependency Injection Principle



Other Principles



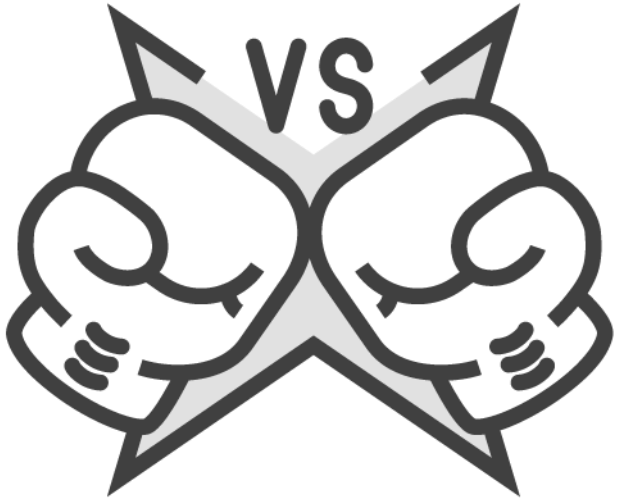
Don't repeat yourself (DRY)

Encapsulate what changes

Favor composition over inheritance

Programming to an interface, not an implementation

Principles vs. Patterns



You don't have to start with principles

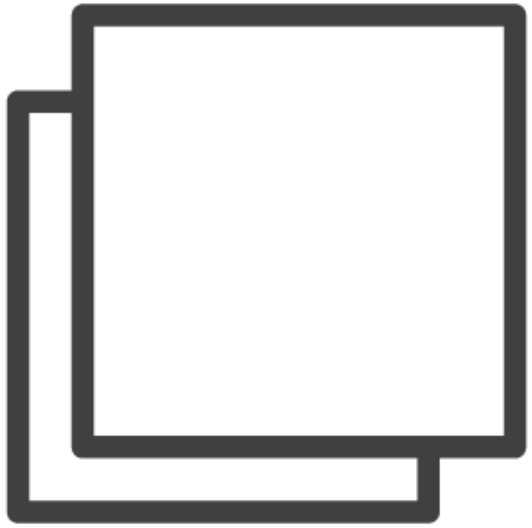
Principles – Low level knowledge

Patterns – High level knowledge

- Proven solutions

Pattern Classification

Pattern Classification



Purpose

- Creational
- Behavioral
- Structural

Creational

Abstract Factory

Builder

Factory Method

Singleton

Prototype

Behavioral

Visitor

Strategy

Observer

Chain of Responsibility

Command

State

Mediator

Interpreter

Iterator

Memento

Template Method

Structural

Decorator

Bridge

Facade

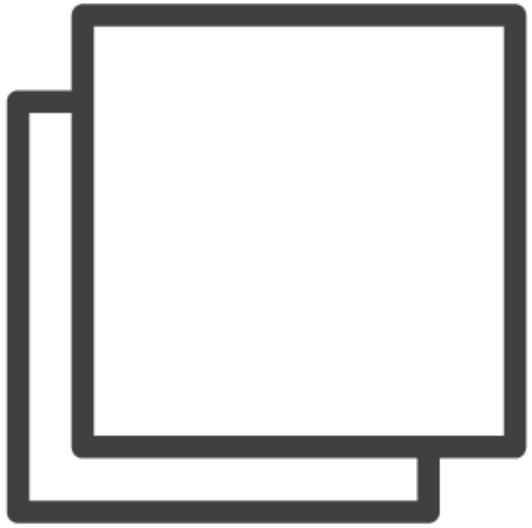
Proxy

Composite

Adapter

Flyweight

Pattern Classification



Scope

- Class
- Object

Class

Interpreter

Factory Method

Adapter

Template Method

Object

Facade Visitor Strategy Proxy

Decorator Chain of Responsibility

Observer Abstract Factory Bridge

Builder Composite State Mediator

Command Prototype Iterator Memento

Singleton Flyweight

Things to Remember

Design patterns

- Christopher Alexander
- Gang of Four
- Patterns are reusable and named solutions to recurring problems in a context

Things to Remember

OOP building blocks

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Things to Remember

OOP principles

- SOLID
  - Single responsibility
  - Open-closed
  - Liskov substitution
  - Interface segregation
  - Dependency injection
- Other
  - Don't repeat yourself
  - Encapsulate what changes
  - Favor composition over inheritance
  - Programming to interfaces

Principles vs. patterns

Things to Remember

Pattern classification

- Purpose
  - Creational
  - Behavioral
  - Structural
- Scope
  - Class
  - Object