

CS 563: Software Maintenance And Evolution

Software Design Patterns & Software Evolution

Oregon State University, Spring 2024

Today's plan

- Learn about
 - Software design patterns (useful for HW1)
 - Software evolution
 - Basic Concepts
 - How to read research papers (example using a software evolution paper)

Design Patterns

- **What are design patterns?**
 - Common reusable *solutions* to *problems* given a certain *context*

Design Patterns

- **What are design patterns?**
 - Common reusable *solutions* to *problems* given a certain *context*
- **How many design patterns exist?**
 - Infinite!
 - Depends on the framework, programming paradigm and programming language

Design Patterns

- **What are design patterns?**
 - Common reusable *solutions* to *problems* given a certain *context*
- **How many design patterns exist?**
 - Infinite!
 - Depends on the framework, programming paradigm and programming language
- **Why should you know and use design patterns?**
 - Faster deliveries
 - Robust and maintainable software
 - Better communication

Design Patterns

- **What are design patterns?**
 - Common reusable *solutions* to *problems* given a certain *context*
- **How many design patterns exist?**
 - Infinite!
 - Depends on the framework, programming paradigm and programming language
- **Why should you know and use design patterns?**
 - Faster deliveries
 - Robust and maintainable software
 - Better communication

Design Patterns

Creational Patterns

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton

Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

Gang of Four (GoF) Design Patterns



Gang of Four Design Patterns is the collection of 23 design patterns from the book “[Design Patterns: Elements of Reusable Object-Oriented Software](#)”. This book was first published in 1994 and is still considered one of the most popular books to learn design patterns today. The gang of four authors, [Erich Gamma](#), Richard Helm, [Ralph Johnson](#) and [John Vlissides](#), initiated the concept of Design Pattern in Software development. These authors are collectively known as Gang of Four.

Design Patterns

Creational Patterns

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton

Focus on
class/object
creation

Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

Gang of Four (GoF) Design Patterns



Gang of Four Design Patterns is the collection of 23 design patterns from the book “[Design Patterns: Elements of Reusable Object-Oriented Software](#)”. This book was first published in 1994 and is still considered one of the most popular books to learn design patterns today. The gang of four authors, [Erich Gamma](#), Richard Helm, [Ralph Johnson](#) and [John Vlissides](#), initiated the concept of Design Pattern in Software development. These authors are collectively known as Gang of Four.

Design Patterns

Creational Patterns

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton

Focus on
class/object
creation

Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Focus on how to
store the
information

Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

Gang of Four (GoF) Design Patterns



Gang of Four Design Patterns is the collection of 23 design patterns from the book “[Design Patterns: Elements of Reusable Object-Oriented Software](#)”. This book was first published in 1994 and is still considered one of the most popular books to learn design patterns today. The gang of four authors, [Erich Gamma](#), Richard Helm, [Ralph Johnson](#) and [John Vlissides](#), initiated the concept of Design Pattern in Software development. These authors are collectively known as Gang of Four.

Design Patterns

Creational Patterns

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton

Focus on
class/object
creation

Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Focus on how to
store the
information

Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

Focus on
communication

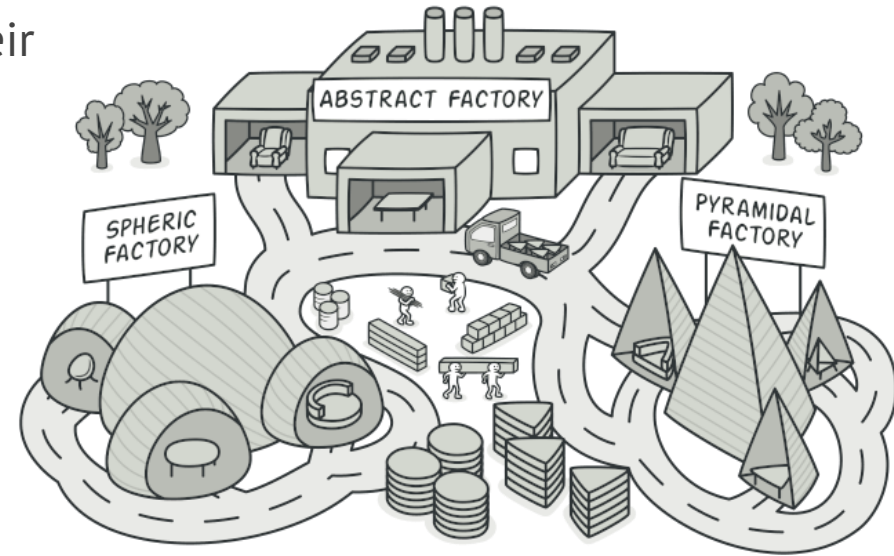
Gang of Four (GoF) Design Patterns



Gang of Four Design Patterns is the collection of 23 design patterns from the book “[Design Patterns: Elements of Reusable Object-Oriented Software](#)”. This book was first published in 1994 and is still considered one of the most popular books to learn design patterns today. The gang of four authors, [Erich Gamma](#), Richard Helm, [Ralph Johnson](#) and [John Vlissides](#), initiated the concept of Design Pattern in Software development. These authors are collectively known as Gang of Four.

Creational Design Pattern (Abstract Factory)

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

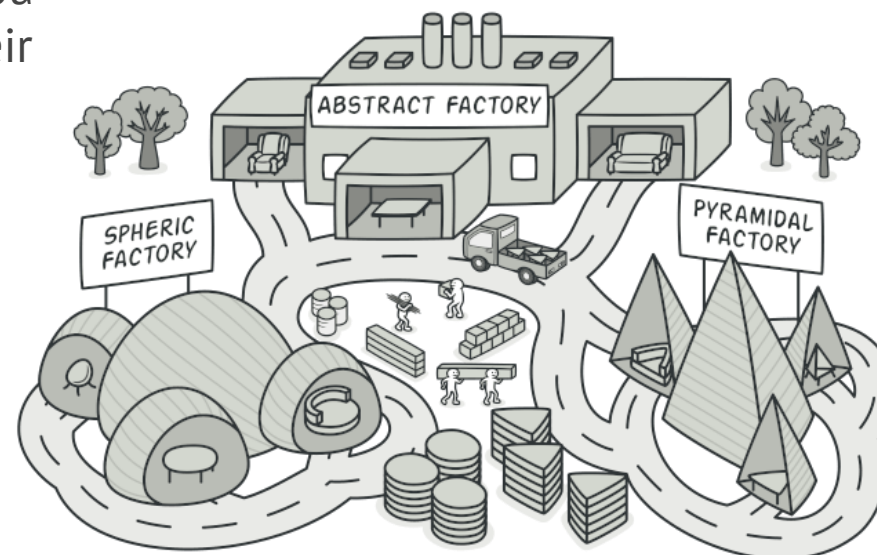











Creational Design Pattern (Abstract Factory)

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Problem: Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

1. A family of related products, say: Chair + Sofa + CoffeeTable.
2. Several variants of this family. For example, products: Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco.



	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

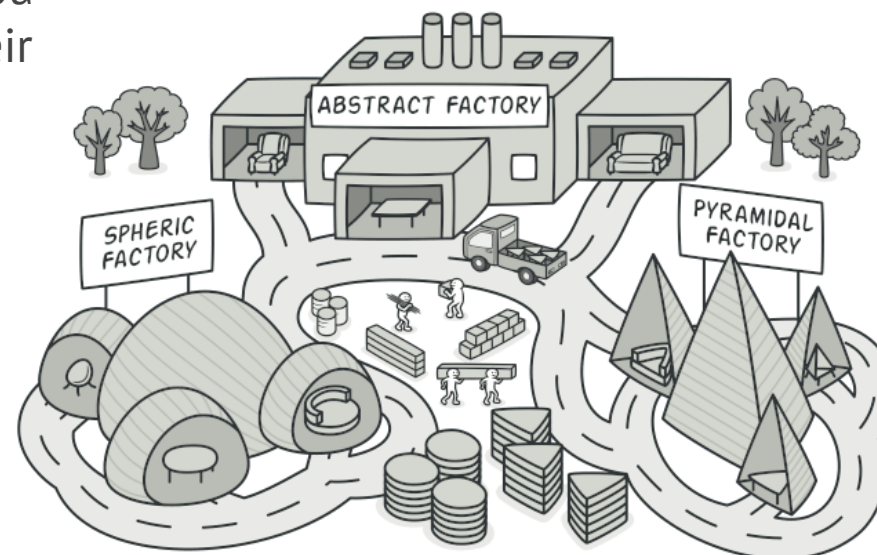
Creational Design Pattern (Abstract Factory)

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Problem: Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

1. A family of related products, say: Chair + Sofa + CoffeeTable.
2. Several variants of this family. For example, products: Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco.

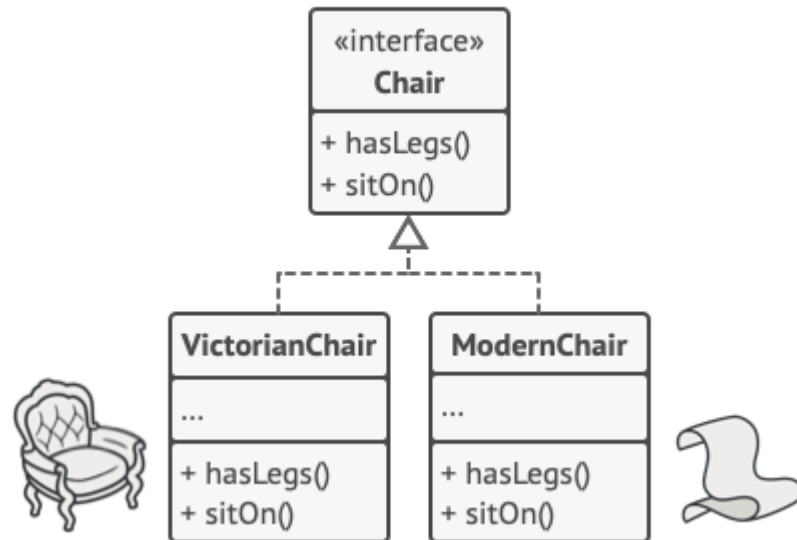
You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture. Also, you don't want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn't want to change the core code each time it happens.



	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

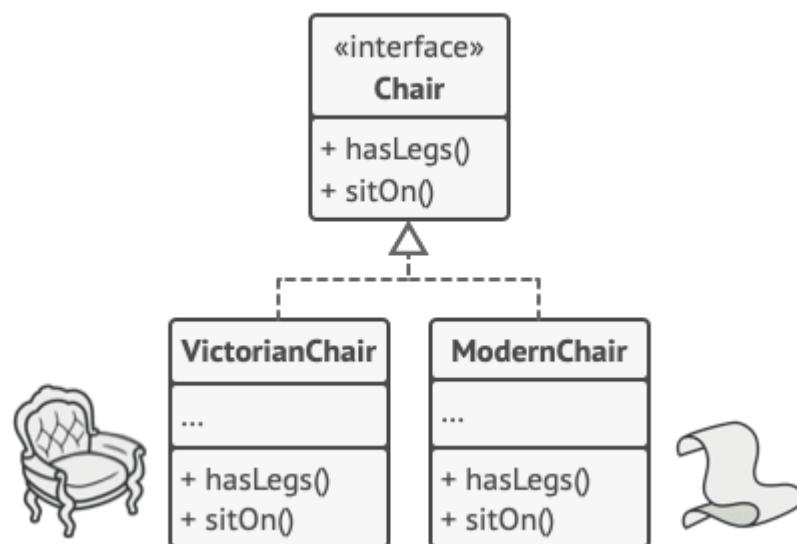
Creational Design Pattern (Abstract Factory)

Solution: The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products following those interfaces.

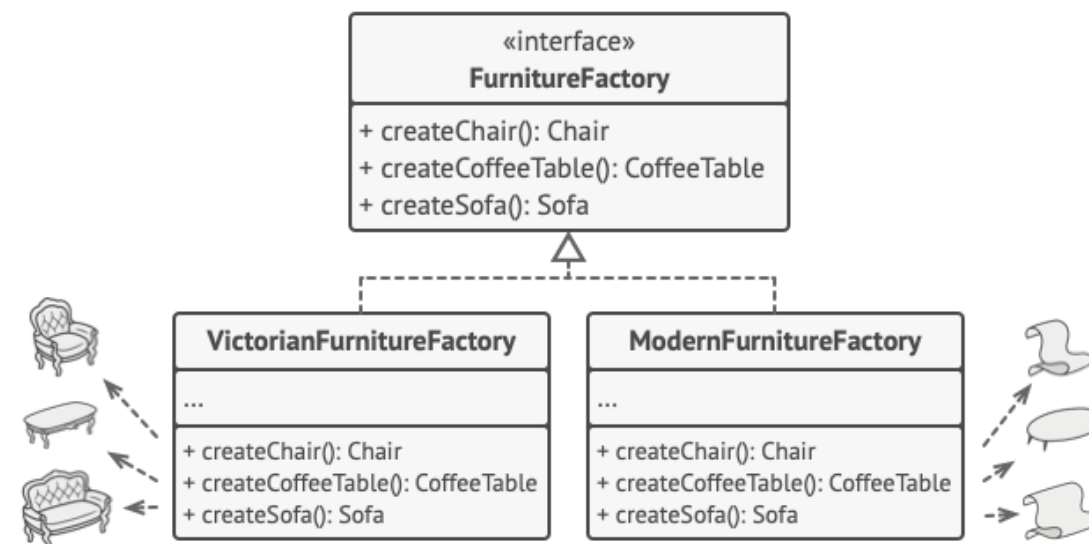


Creational Design Pattern (Abstract Factory)

Solution: The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products following those interfaces.

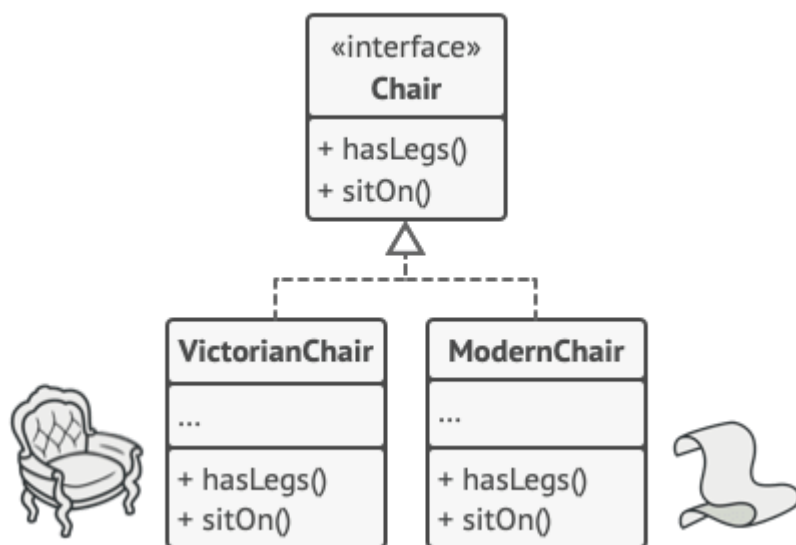


The next move is to declare the *Abstract Factory*—an interface with a list of creation methods for all products that are part of the product family (e.g., createChair, createSofa and createCoffeeTable). These methods must return **abstract** product types represented by the interfaces extracted previously: Chair, Sofa, CoffeeTable. For each variant of a product family, we create a separate factory class based on the AbstractFactory interface.



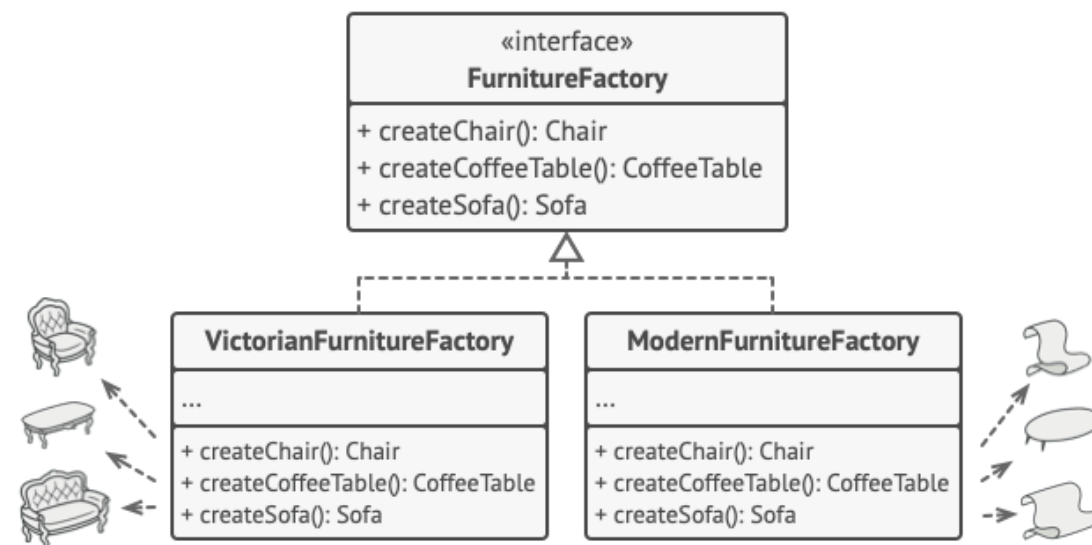
Creational Design Pattern (Abstract Factory)

Solution: The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products following those interfaces.



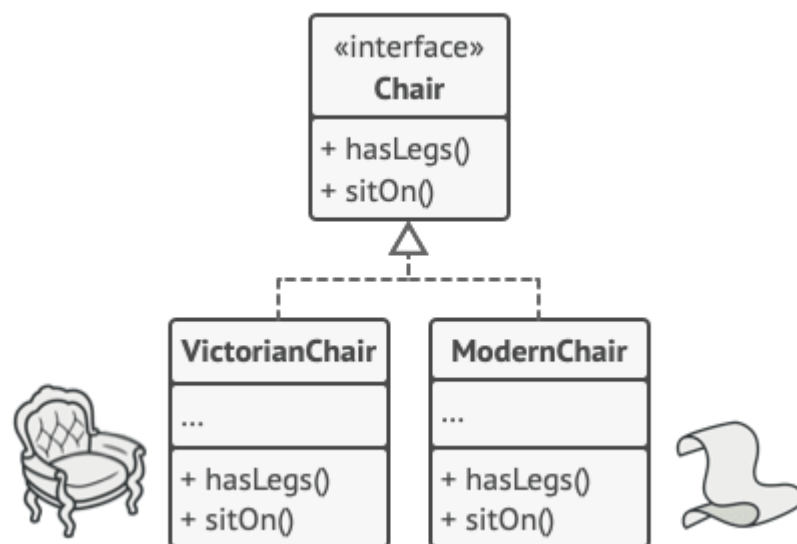
A factory is a class that returns products of a particular kind. E.g., **ModernFurnitureFactory** can only create **ModernChair**, **ModernSofa** and **ModernCoffeeTable** objects.

The next move is to declare the *Abstract Factory*—an interface with a list of creation methods for all products that are part of the product family (e.g., `createChair`, `createSofa` and `createCoffeeTable`). These methods must return **abstract** product types represented by the interfaces extracted previously: **Chair**, **Sofa**, **CoffeeTable**. For each variant of a product family, we create a separate factory class based on the **AbstractFactory** interface.



Creational Design Pattern (Abstract Factory)

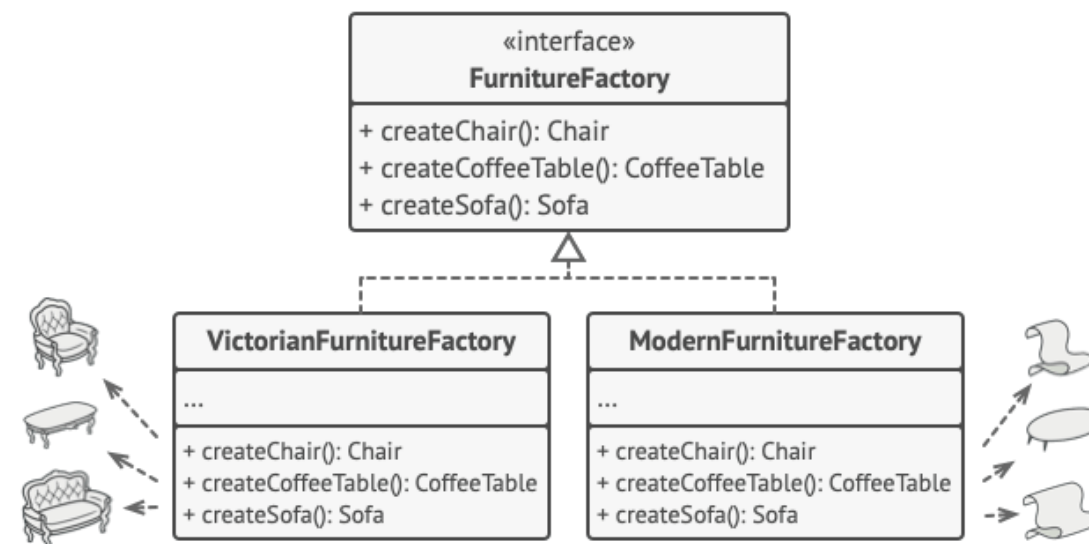
Solution: The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products following those interfaces.



The client code must work with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.

Image source: <https://refactoring.guru/design-patterns>

The next move is to declare the *Abstract Factory*—an interface with a list of creation methods for all products that are part of the product family (e.g., `createChair`, `createSofa` and `createCoffeeTable`). These methods must return **abstract** product types represented by the interfaces extracted previously: **Chair**, **Sofa**, **CoffeeTable**. For each variant of a product family, we create a separate factory class based on the **AbstractFactory** interface.



Object Oriented Design Principles (SOLID)

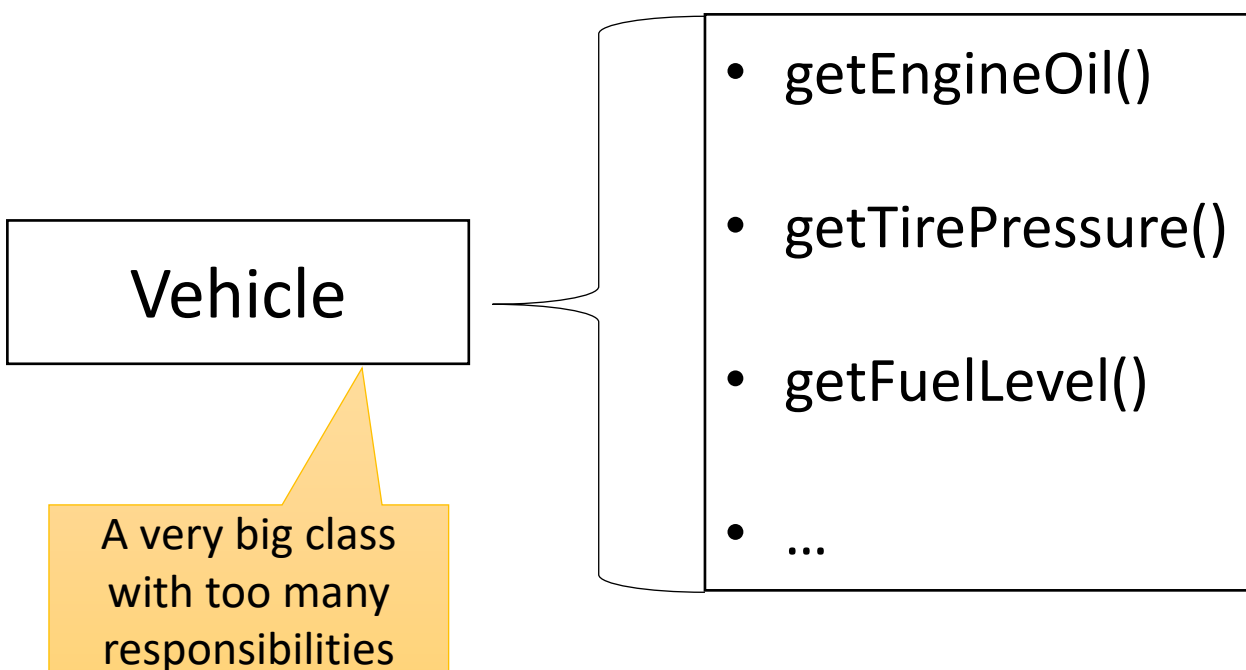
- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

Single Responsibility Principle

“A class should have only one reason to change”

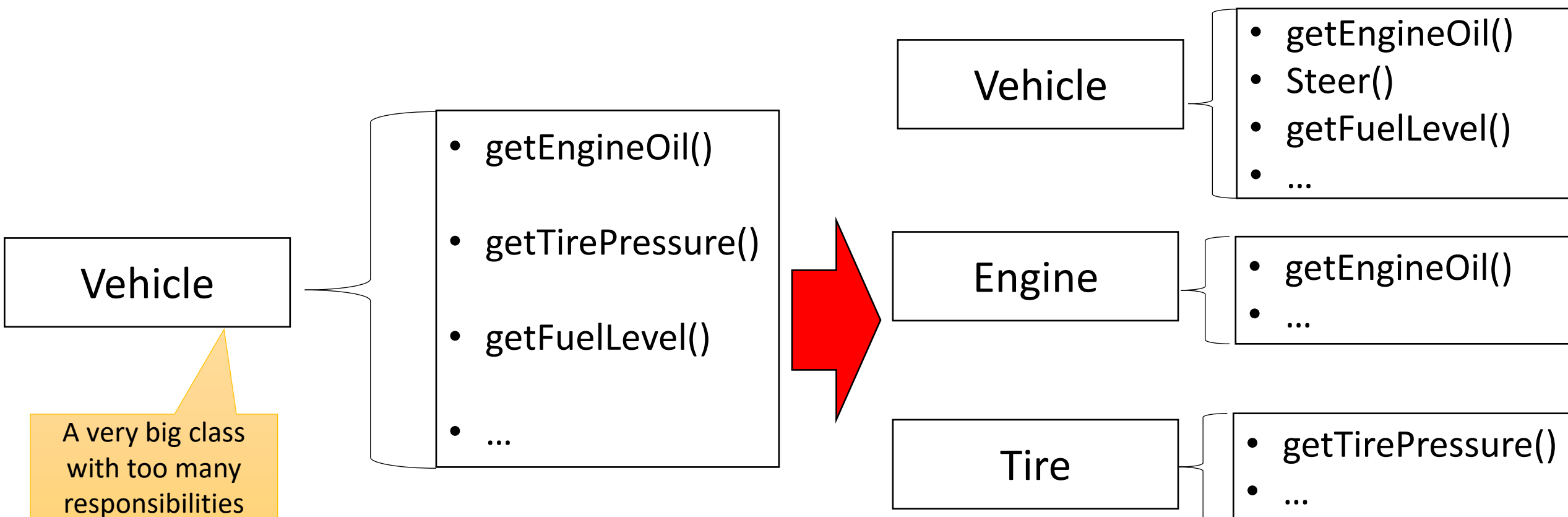
Single Responsibility Principle

“A class should have only one reason to change”



Single Responsibility Principle

“A class should have only one reason to change”

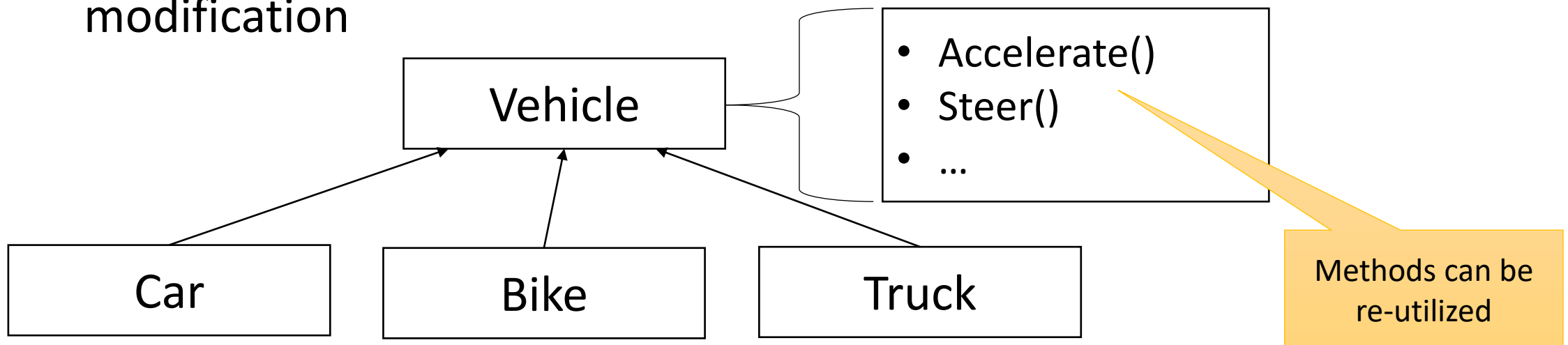


Open/Closed Principle

- “A class should be **open** for extension, but **closed** for modification”
- Promotes **re-utilization**
- When you split responsibility of a class, you should do so in a way that behavior can be extended/replaced instead of requiring its modification

Open/Closed Principle

- “A class should be **open** for extension, but **closed** for modification”
- Promotes **re-utilization**
- When you split responsibility of a class, you should do so in a way that behavior can be extended/replaced instead of requiring its modification

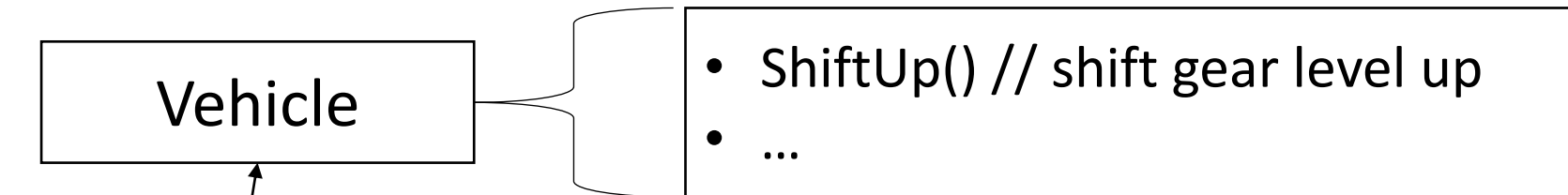


Liskov Substitution Principle

You should be able to change an instance using a sub-type and your code should still work.

Liskov Substitution Principle

You should be able to change an instance using a sub-type and your code should still work.



```

1. public class ElectricCar implements Vehicle {
2.
3.     @override
4.     public void shiftUp() {
5.         throw new UnsupportedOperationException("Method is not supported");
6.     }
7.
8.     public void accelerate() {
9.         ...
10.    }
11.    ...
12. }
    
```

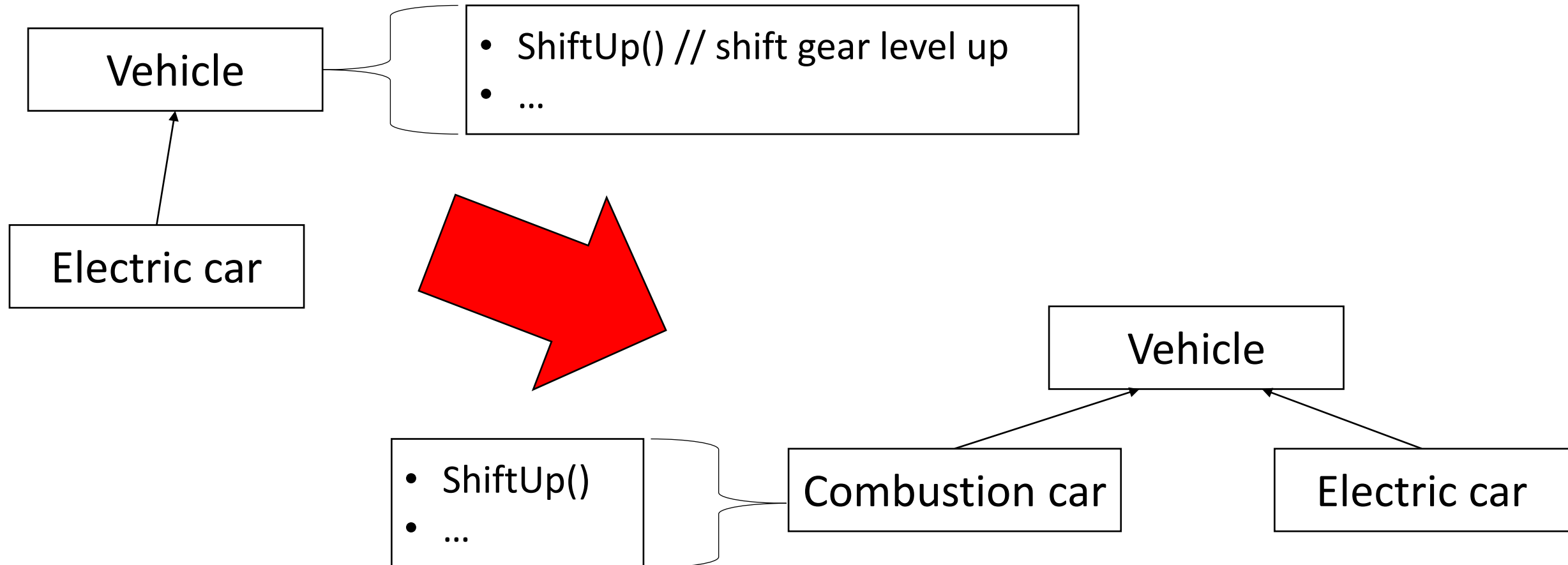
Violates Liskov Substitution principle as you are forced to implement the interface method

Interface Segregation Principle

- *More interfaces are better than too little*
- *Allows to split the responsibility of a class without LSP violation*

Interface Segregation Principle

- *More interfaces are better than too little*
- *Allows to split the responsibility of a class without LSP violation*

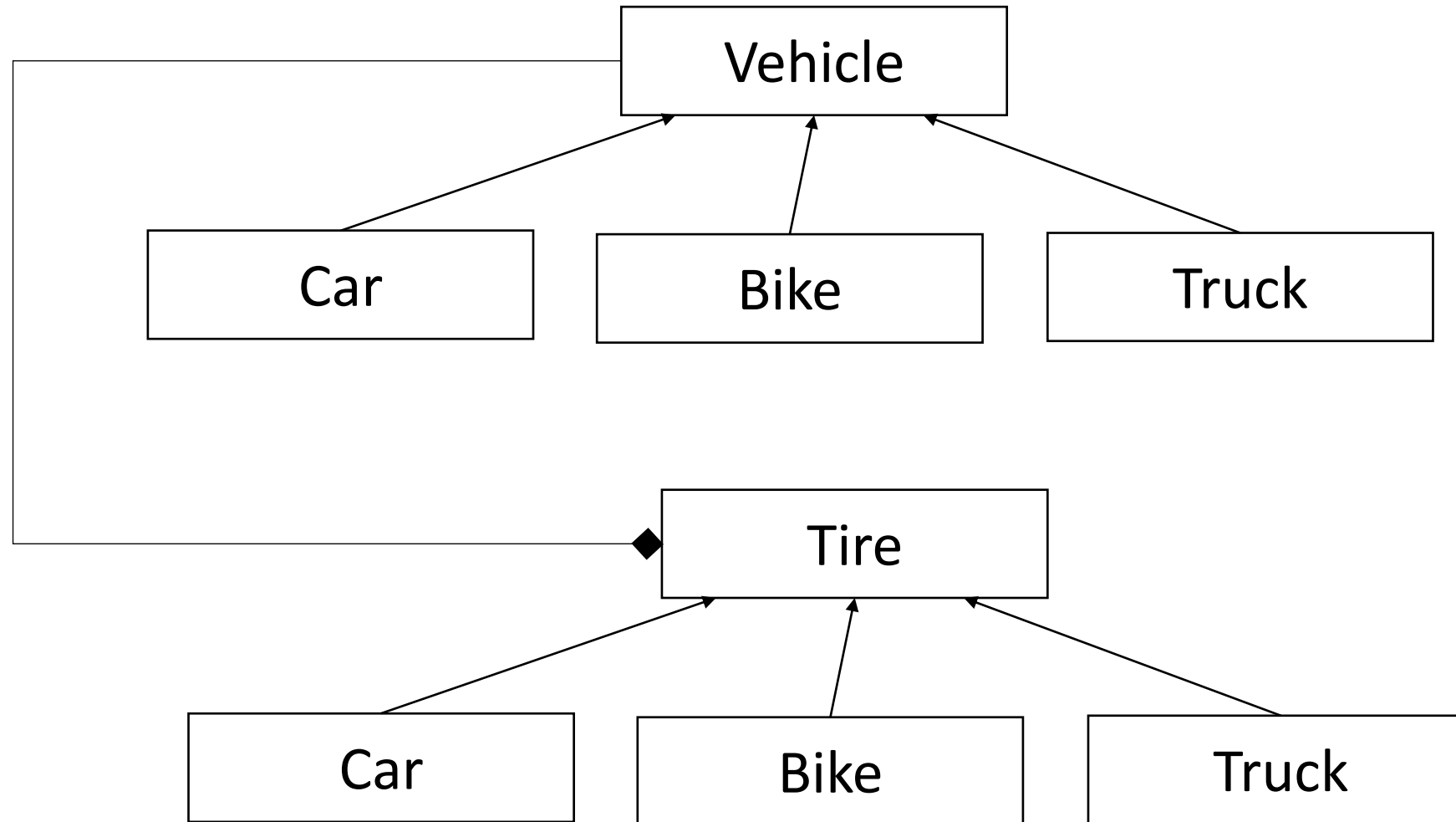


Dependency Inversion Principle

One should depend on abstraction and not concrete instances.

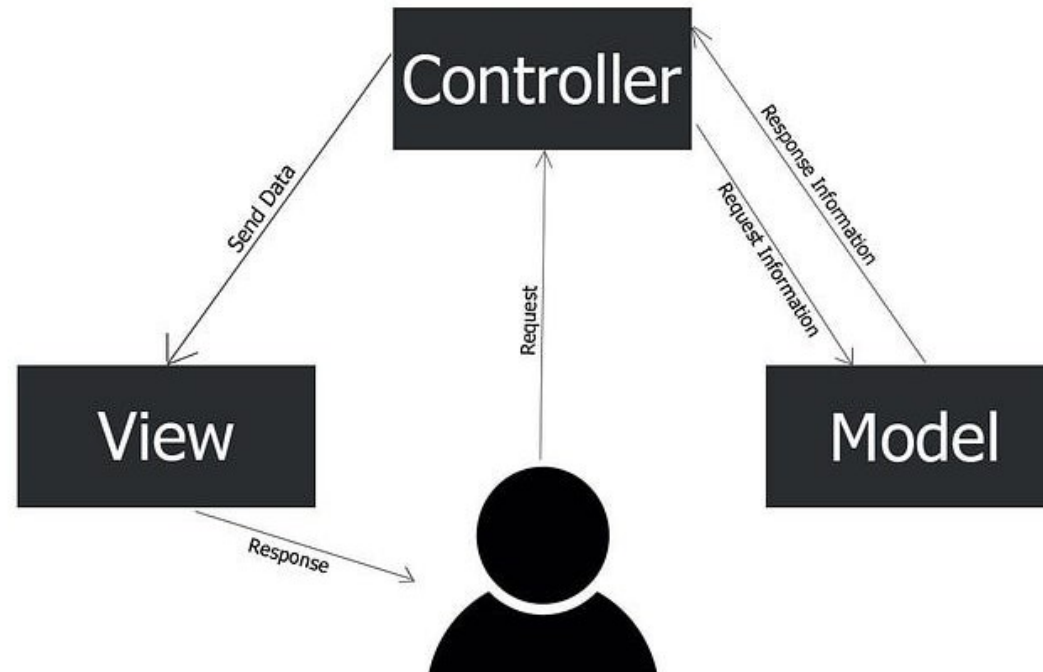
Dependency Inversion Principle

One should depend on abstraction and not concrete instances.

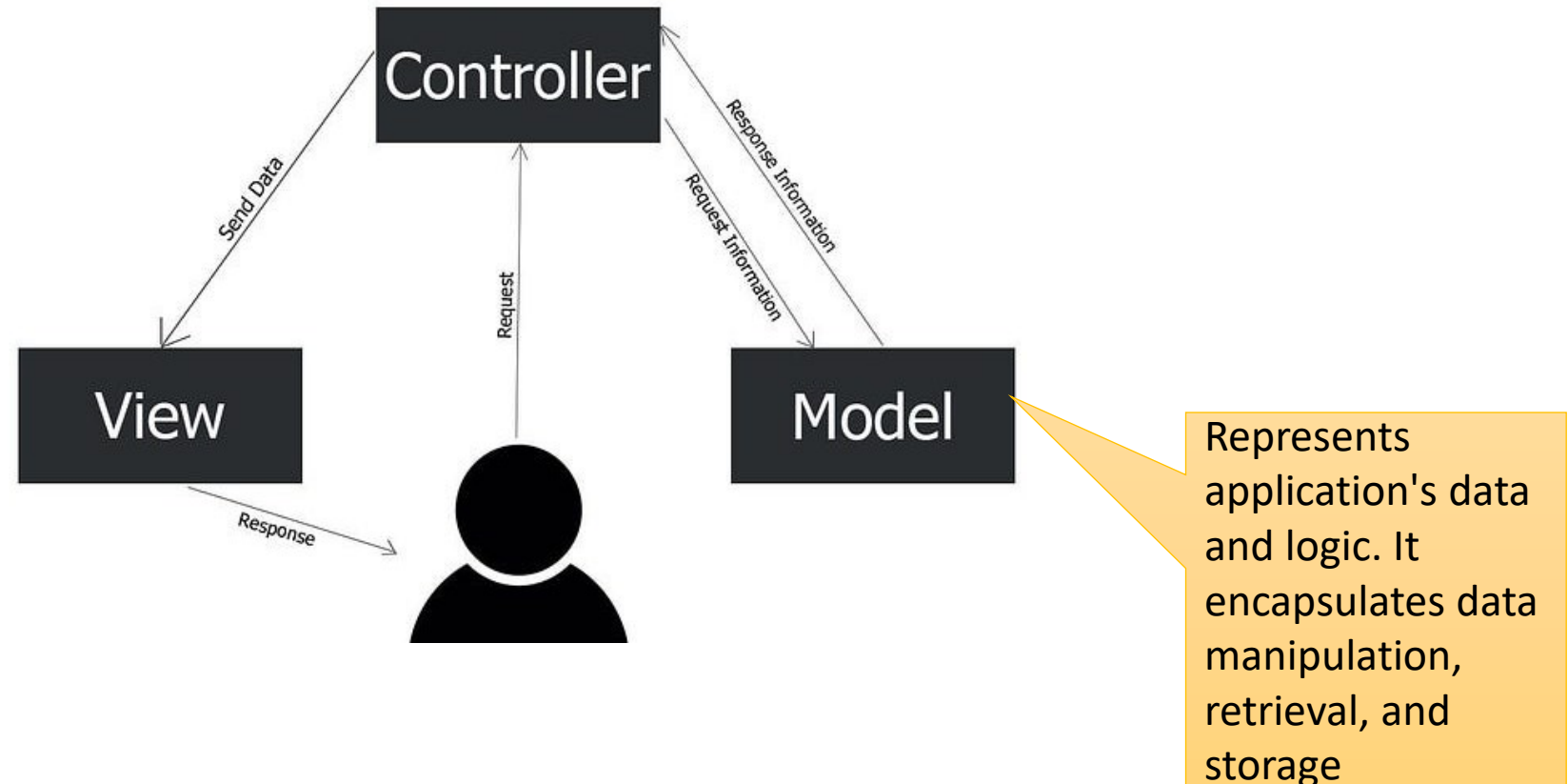


Model-View-Controller (MVC) design pattern

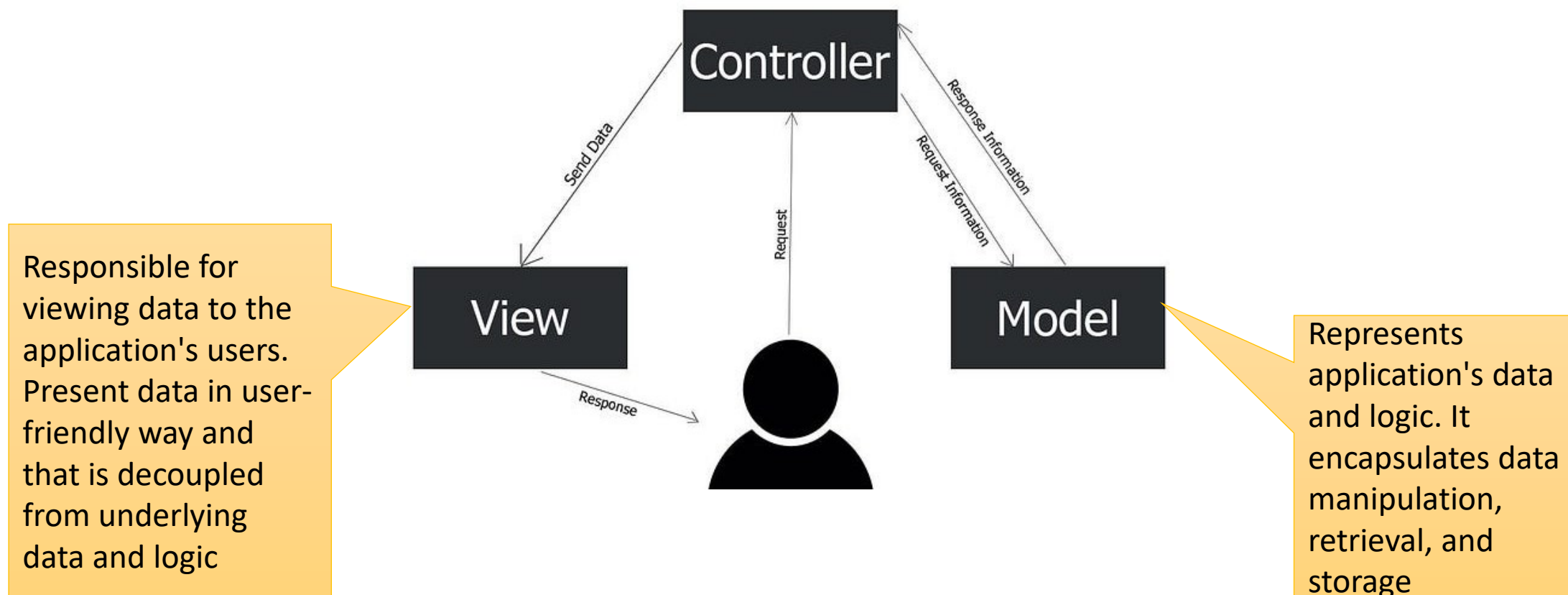
Useful for Desktop GUI applications



Model-View-Controller (MVC) design pattern



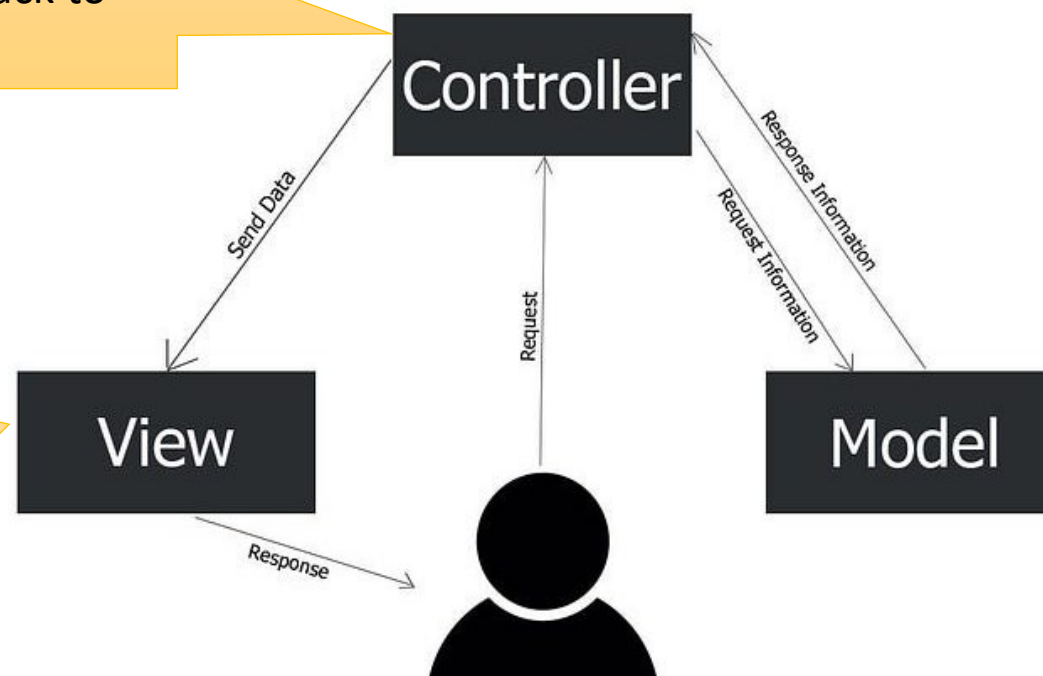
Model-View-Controller (MVC) design pattern



Model-View-Controller (MVC) design pattern

Acts as an intermediary between Model and View. Receives input from users via View, processes it via Model, and responds back to the user via View.

Responsible for viewing data to the application's users. Present data in user-friendly way and that is decoupled from underlying data and logic



Represents application's data and logic. It encapsulates data manipulation, retrieval, and storage

Benefits of using MVC design pattern

- **Separation of Concerns:** MVC promotes modularity by separating concerns into distinct components. This separation makes code easier to manage, test, and maintain.
- **Code Reusability:** With clear boundaries between components, developers can reuse Models, Views, and Controllers across different parts of the application or in entirely different projects.
- **Collaborative Development:** The MVC pattern facilitates collaborative development by allowing teams to work on different components simultaneously without stepping on each other's toes.
- **Scalability:** As applications grow in complexity, the MVC pattern makes it easier to scale by adding or modifying components without affecting the entire application.
- Frameworks like **Django (Python)**, **Ruby on Rails (Ruby)**, and **Spring (Java)** provide built-in support for MVC, making implementation even more straightforward.

Software Evolution vs. Software Maintenance

- The terms “**Software Evolution**” and “**Software Maintenance**” are often used interchangeably
- However, there is a semantic difference

Software Evolution vs. Software Maintenance

- The terms “**Software Evolution**” and “**Software Maintenance**” are often used interchangeably
- However, there is a semantic difference
- Lowell Jay Arthur distinguish the two terms as follows:
 - “**Software maintenance** means to preserve from failure or decline.”
 - “**Software evolution** means a continuous change from lesser, simpler, or worse state to a higher or better state.”
- Keith H. Bennett and Lie Xu use the term:
 - “**maintenance** for all post-delivery support and **evolution** to those driven by changes in requirements.”
- **Maintenance** is set of planned activities whereas **evolution** concerns whatever happens to a system over time.

Software Evolution

- In 1965, Mark Halpern used the term *evolution* to define the dynamic growth of software.
- The term evolution in relation to application systems took gradually in the 1970s.
- **Lehman and his collaborators from IBM are generally credited with pioneering the research field of software evolution.**
- Lehman formulated a set of observations that he called **laws of evolution**.
- These laws are the results of studies of the evolution of large-scale proprietary or closed source system (CSS).

In-class exercise: How to read research papers

- Download the reading materials from Canvas associated with today's lecture.
- *Three-pass method – S. Kesav*
 - *First pass – general idea of the paper*
 - *Second pass – grasp the paper's content, but not its details*
 - *Third pass – understand the paper in depth*
- *How to read research paper – Michael Mitzenmacher*
 - Reading *critically*
 - Reading *creatively*
 - Make notes
 - Summarize paper in one or two sentences
 - If possible, compare to other works

Software Evolution – Over last 50 Years

Goal: Apply "three-pass approach" to understand and discuss about the state-of-the-art in software evolution field

1

The evolution of the laws of software evolution. A discussion based on a systematic literature review.

ISRAEL HERRAIZ, Technical University of Madrid, Spain

DANIEL RODRIGUEZ, University of Alcala, Madrid, Spain

GREGORIO ROBLES and JESUS M. GONZALEZ-BARAHONA, GSyC/Libresoft, University Rey Juan Carlos, Madrid, Spain

After more than 40 years of life, software evolution should be considered as a mature field. However, despite such a long history, many research questions still remain open, and controversial studies about the validity of the laws of software evolution are common. During the first part of these 40 years the laws themselves evolved to adapt to changes in both the research and the software industry environments. This process of adaption to new paradigms, standards, and practices stopped about 15 years ago, when the laws were revised for the last time. However, most controversial studies have been raised during this latter period. Based on a systematic and comprehensive literature review, in this paper we describe how and when the laws, and the software evolution field, evolved. We also address the current state of affairs about the validity of the laws, how they are perceived by the research community, and the developments and challenges that are likely to occur in the coming years.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement

General Terms: Management

Additional Key Words and Phrases: Laws of Software Evolution, Software Evolution

Announcements

- Homework-1 assignment released today (Improve application based on the software design patterns and principles)
- Paper selection assignment released today
 - A list of 10 papers associated with their presentation date.
 - A group of at most 2 students will present each paper (selection is on the FCFS basis). Read the assignment for more details.
 - Everyone will write and submit a review of all the papers individually (due before that paper's presentation date)
 - Form a group (optional) and use the technique to read papers we discussed to go through all papers. Discuss papers with your partner and select the one that interests you.
 - NOTE: Upcoming lectures will cover the concepts described in the research papers, so don't reject a paper because you feel you don't know the specific concepts used in that paper. Ask me, if you have any doubts about whether a specific topic will be covered or not.