# CSE1500: Web - Lecture 7 - Design Patterns & ReactJS

Lecture Summary

The WDT Team

Ujwal Gadiraju, Sagar Chethan Kumar, Ciprian Ionescu

## Contents



⚠️ These summaries are not extensive overviews of lecture content and **DO NOT** cover all possible exam material. Rather, these serve as a refresher and capture **main points** for a given lecture.

## Learning Goals

You should be able to do the following after following the lecture:

- **Explain the basic architecture of the Internet and the Web. (LO1)**
- **Develop server-side code for authorization, cookies, sessions, and database access. (LO5)**

# 1 Design Patterns & ReactJS

## 1.1 Design Patterns

### 1.1.1 Intro & Further Background (Optional)

**What are design patterns?** Design patterns are typical solutions for commonly occurring problems in software design. They are pre-made blueprints you can adapt to solve a recurring design problem in your code. One should not confuse an *algorithm* with a *design pattern*, although they might sound the same. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be completely different!

**Why should I learn design patterns?** Design patterns are tried and tested solutions to common problems in software design. Design patterns define a common language you and your teammates can use to communicate more efficiently. For example, you can say, *"Oh, just use a Singleton for that,"* and everyone will understand the idea behind your suggestion. There is no need to explain what a singleton is if you know the pattern and its name. As a programmer, who might be unaware of these design patterns, you might have unintentionally been implementing these for multiple years without knowing - we just now define a name for these patterns.


Figure 1: Classification of Design Patterns

**Creational Design Patterns** These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

**Singleton Pattern:** With this pattern, we ensure that a class has only one instance, which we can provide as a global access point to this instance. This is useful when there must be exactly **ONE** instance of a class for the lifetime of the application and it must be accessible to clients. Typical examples include thread handlers, processes, and loggers.

**Factory Method Pattern:** This is a pattern that allows the instantiation of an object to a subclass. The Factory Method pattern defines an interface (superclass) for creating an object (the product) but lets subclasses decide which class (concrete product) to instantiate.

**Abstract Factory Method Pattern:** This pattern is very similar to the Factory Method pattern, with the added bonus of another dimension of variability. We have an interface that allows us to create families of related or dependent objects! (Essentially another abstract interface from which factories can extend/implement).

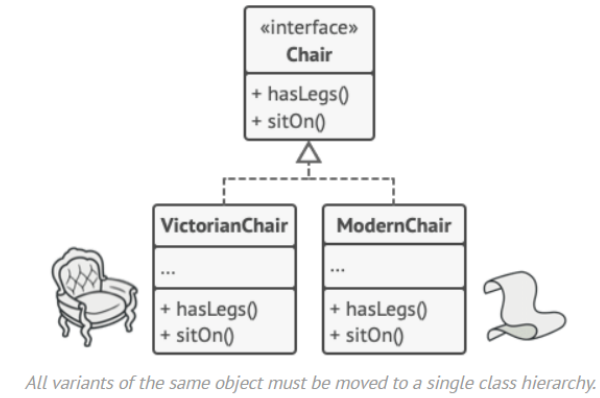


Figure 2: Factory Pattern Example

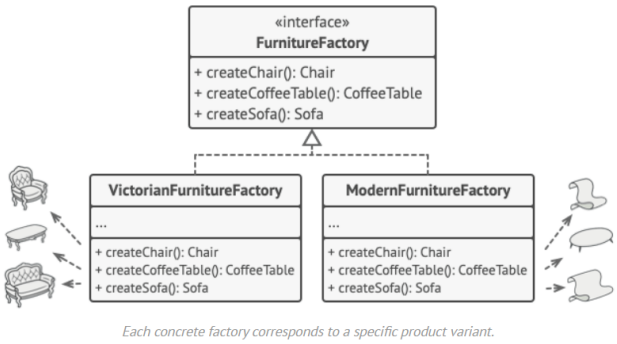


Figure 3: Abstract Factory Pattern Example

It should be noted, that the Factory Method abstracts the way *objects* are created, while the Abstract Factory also abstracts the way *factories* are created.

**Structural Design Patterns** These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

**Adapter Pattern:** This pattern allows incompatible interfaces to work together. It is a special type of class/object that converts the interface of one object into a format in which another object can understand it. Intuitively, they can be described as "socket" adapters we use when charging devices in different countries.

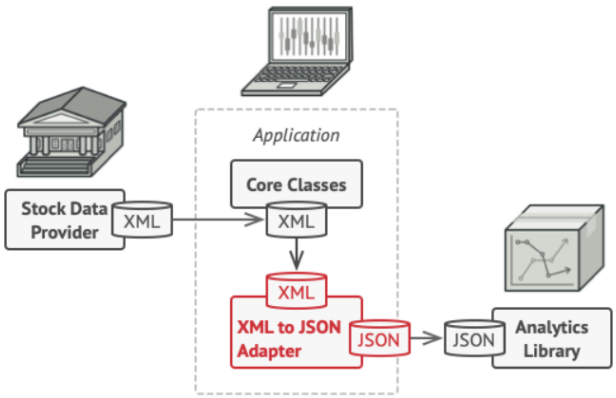


Figure 4: Adapter Pattern Example

**Decorator Pattern:** This pattern allows for new functionality to be added to an object dynamically! This typically is used when extending a class becomes no longer maintainable or within scope. This oftentimes provides a better solution to permutation issues that cannot be covered by inheritance (e.g. all combinations of weapons and armor as subclasses). **Compos-**

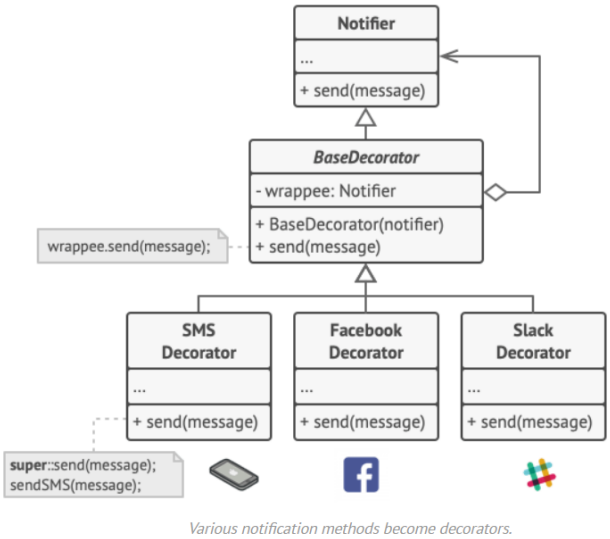


Figure 5: Decorator Pattern Example

**ite Pattern:** This is a design pattern that allows us to compose objects into tree structures and work with the structures as if they were individual objects themselves! This is often used when we have inherent recursive structures in our application (like menus, which are submenus of a menu, which is also a submenu of a menu...)

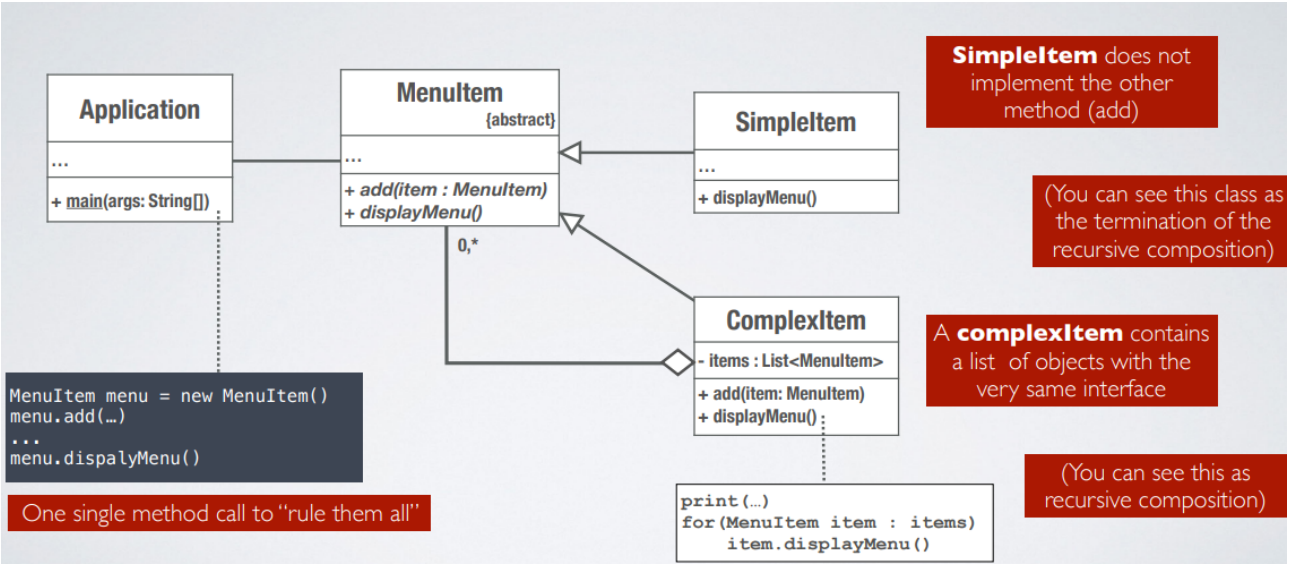


Figure 6: Composite Pattern Example

**Behavioral Design Patterns** These patterns are concerned with algorithms and the assignment of responsibilities between objects.

**Command Pattern:** This pattern is typically used when encapsulating a request as an object, thereby allowing for us to pass requests as a method argument, delay or queue its execution, and support irreversible operations.

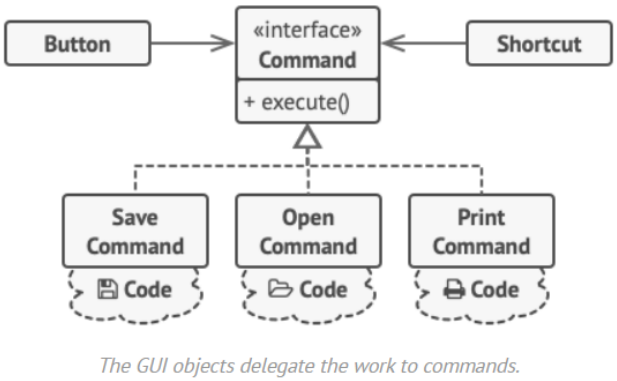


Figure 7: Command Pattern Example

**Observer Pattern:** Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing (one-to-many dependency between objects).

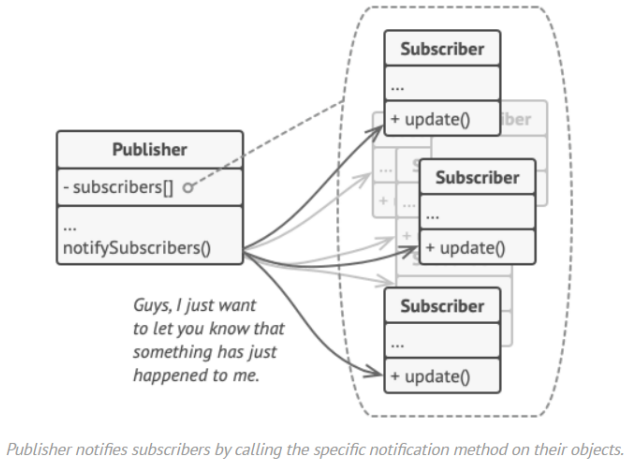


Figure 8: Observer Pattern Example

**Strategy Pattern:** This is a design pattern that allows us to define a family of related algorithms, encapsulating each one, and making them interchangeable through a shared hierarchical setup.

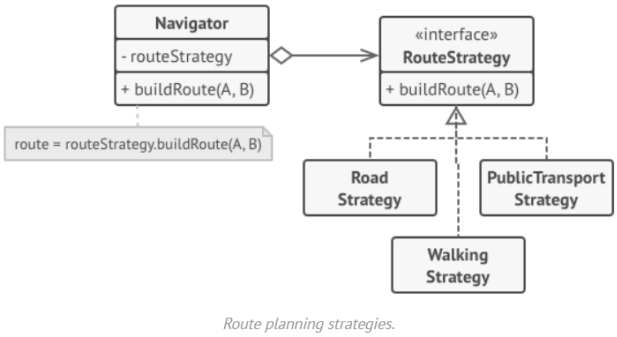


Figure 9: Strategy Pattern Example

### 1.1.2 Model-View-Controller (MVC) Pattern

The Model-View-Controller Pattern, also known as MVC, is one of the most widely used architectural patterns that divide the application into three main layers: Model, View, and Controller. As expected, it is often used in **web development**!

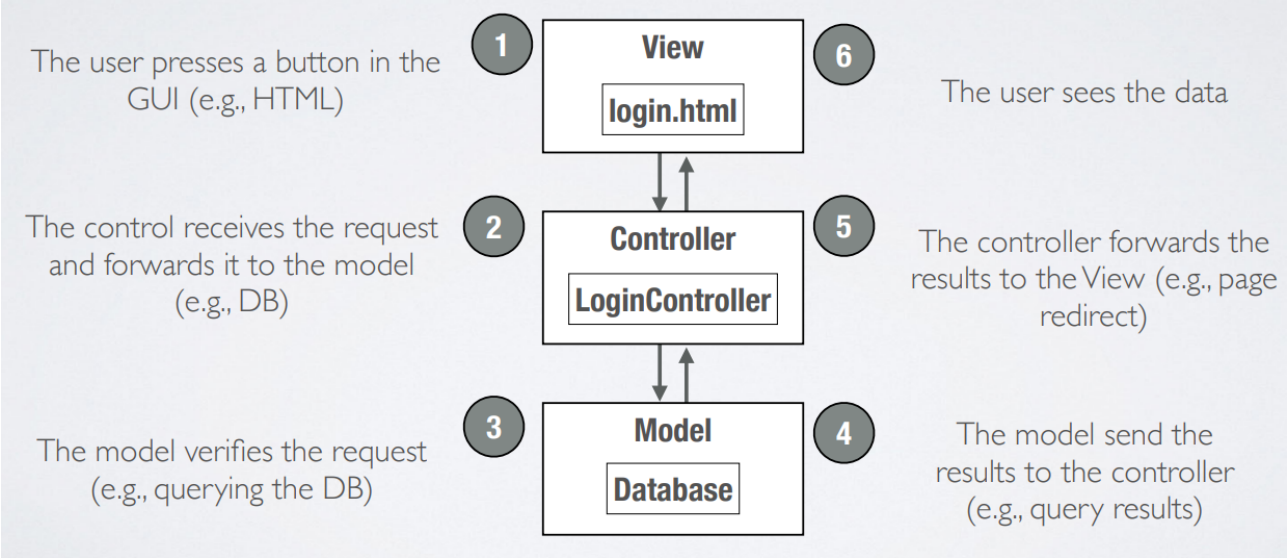


Figure 10: MVC Illustrated

In short, the Model represents the data, the View handles the UI, and the Controller manages the flow of data between the Model and the View.

## 1.2 ReactJS

ReactJS, also known as React, is a library for web and native user interfaces. ReactJS, unlike vanilla JS, allows you to build interactive and dynamic interfaces through their structuring of **components**.

ReactJS lets you build user interfaces out of individual pieces called **components**. Create your own React **components** like `Thumbnail`, `LikeButton`, and `Video`. Then combine them into entire screens, pages, and apps.

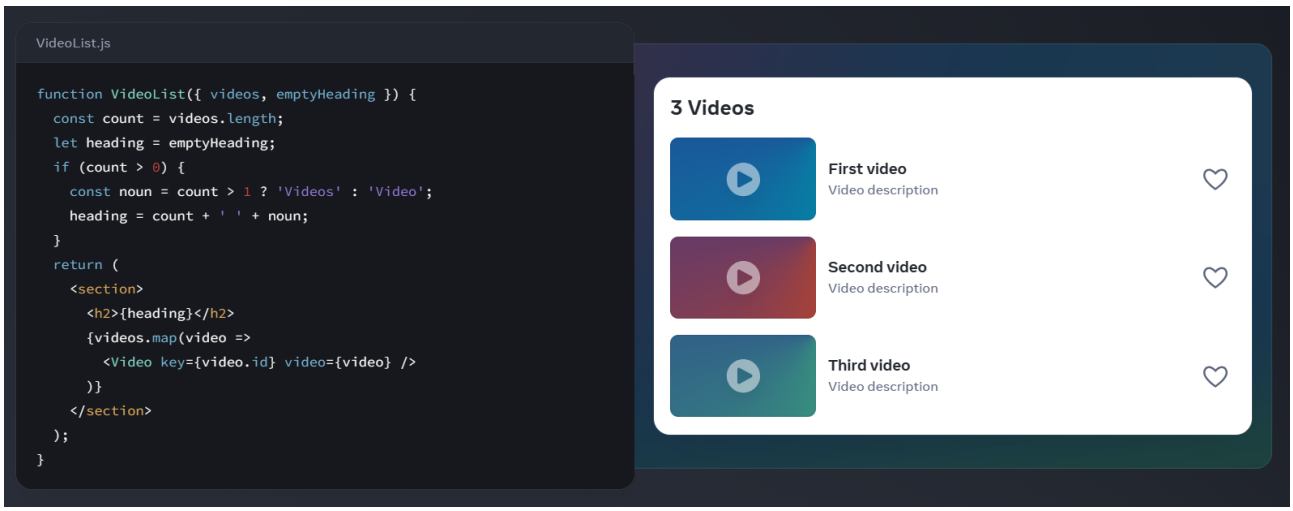


Figure 11: Components Example ReactJS

ReactJS components are JavaScript functions. Want to show some content conditionally? Use an if statement. Displaying a list? Try array `map()`. This makes DOM manipulation **MUCH** easier! ReactJS makes use of a markup syntax called JSX. It is a JavaScript syntax extension popularized by React. Putting JSX markup close to related rendering logic makes React components easy to create, maintain, and delete.

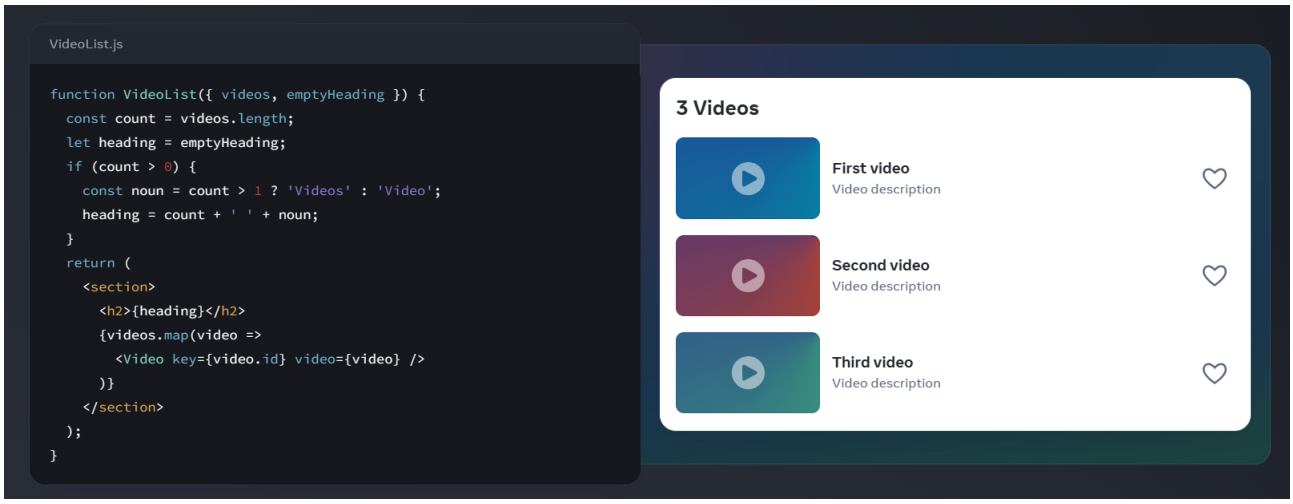


Figure 12: Conditional ReactJS

Styling with CSS in ReactJS is essentially the same as done in HTML (typically it is best practice to have a dedicated stylesheet). Based on a component props or state this can be different too!

Styling and CSS

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
// Or....
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

HTML Differences

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
<!-- Becomes.... -->
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Lastly, React Lists are similar to JS arrays (with function definitions) but have some semantic differences. See the example linked defined here. For more information on ReactJS/Reacat please refer to the React dev documentation.

## 2 Sample Question Types

> 🙃 Refer to the Weblab environment and the `.zip` file from the lecture.

## 3 References and Further Reading

Here are some useful resources for learning more about **Lecture 7**:

- Refactoring Guru
- CSE2115 Course References
- MVC Intro
- React Dev