KINGSLAND UNIVERSITY

JS Design Patterns

Following Best Practices

Table of Contents

Definition and Structure

Design Patterns

What is a Pattern?

- **Recurring solutions to design problems** you see over and over
- Constitute a **set of rules** describing how to accomplish certain tasks
- Design patterns focus more on **reuse of recurring architectural design themes**
- Frameworks focus on detailed design and implementation

Categories of Design Patterns

- Design Patterns can be broken down into a number of different categories:
  - **Creational**
  - **Structural**
  - **Behavioral**

Creational Design Patterns

- Focus on handling **object creation** mechanisms
- These patterns control the creation problems
- Some of the patterns that fall under this category are:
  - **Constructor**
  - **Factory**
  - **Prototype**
  - **Singleton**

Structural Design Patterns

- Focus on **object composition**
- Ensure that when one part of a system changes, the entire structure of the system **doesn't need** to do the same
- Some of the patterns that fall under this category are:
  - **Decorator**
  - **Facade**
  - **Adapter**
  - **Proxy**

Behavioral Design Patterns

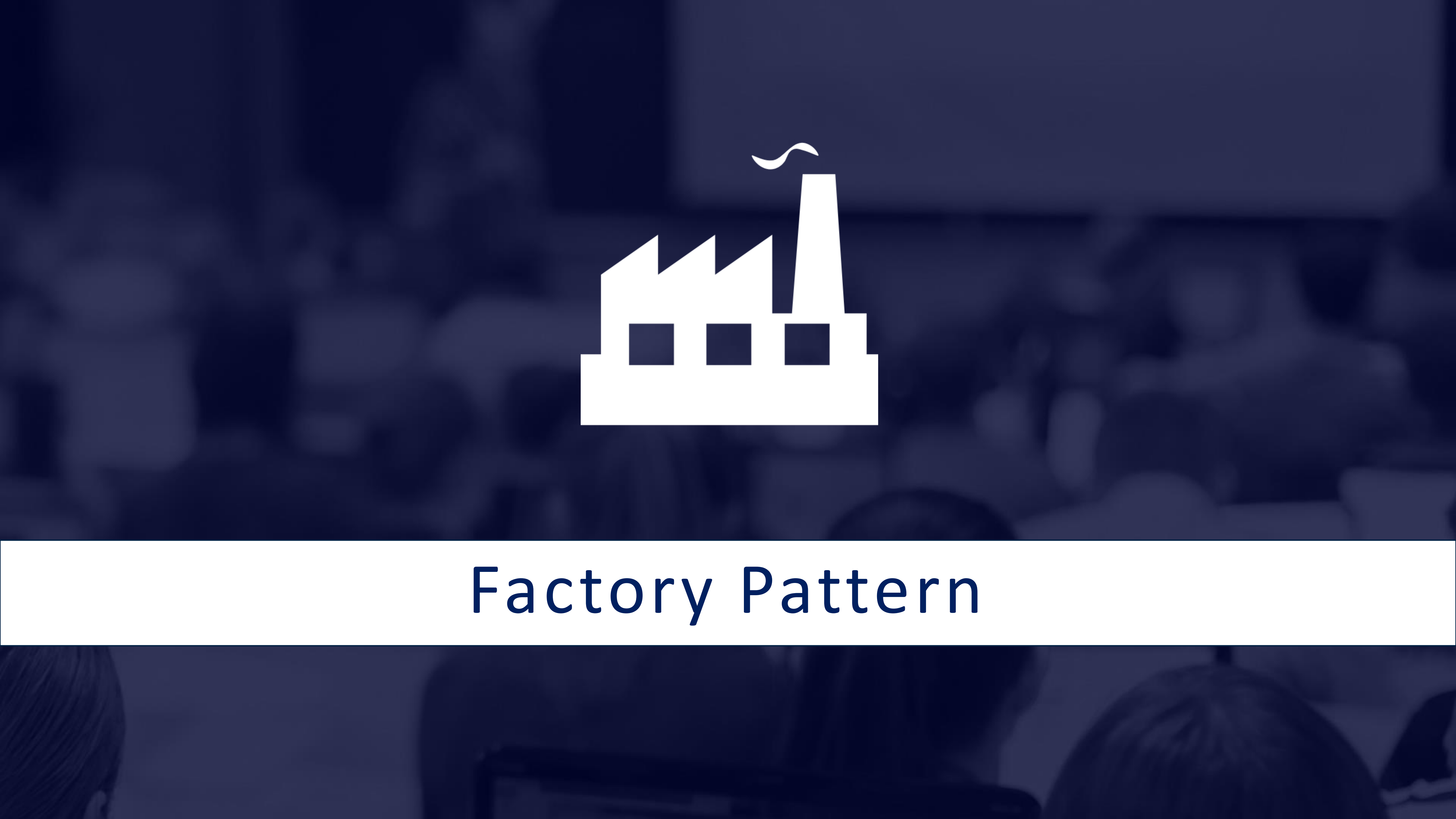
- Focus on improving or streamlining the **communication** between disparate objects in a system
- Some of the patterns that fall under this category are:
  - **Iterator**
  - **Mediator**
  - **Observer**
  - **Visitor**

Benefits of Design Patterns

- **Inspiration**
  - Patterns don't provide solutions, they **inspire solutions**
  - Patterns explicitly **capture expert knowledge** and design tradeoffs
- Patterns improve **communication**
  - Pattern names form a **vocabulary**
- Design patterns enable **large-scale reuse** of software architectures

Drawbacks of Design Patterns

- Patterns **do not lead to direct code reuse**
- Patterns are **deceptively simple**
- Teams may suffer from **patterns overload**
- **Integrating patterns** into a software development process is a **human-intensive** activity



Factory Pattern

The Factory Pattern

- Main purpose - **creation of objects**
- Use when
  - a class **can't anticipate** the class of objects it must create
  - a class wants its **subclasses to specify** the objects it creates
  - classes **delegate responsibility** to one of several helper subclasses

Pros and Cons

Pros

- ✓ Compatible products
- ✓ You avoid tight coupling
- ✓ ***Single Responsibility Principle***
- ✓ ***Open/Closed Principle***

Cons

- ✓ The code may become more **complicated** than it should be

Example

```
function Employee(name) {
  this.name = name;
  this.say = function () {
    console.log(`I am ${name}`)
  };
}
function EmployeeFactory() {
  this.create = function (name) {
    return new Employee(name);
  };
}
let employeeFactory = new EmployeeFactory();
```

Example (2)

```
let people = [];
let employeeFactory = new EmployeeFactory();

people.push(employeeFactory.create("Joan Peterson"));
people.push(employeeFactory.create("Tim O'Neill"));

people.forEach((person) => {
  person.say();
})
```

Decorator Pattern

The Decorator Pattern

- Lets you attach **new behaviors** to objects
- Uses
  - For **adding responsibilities** to individual objects dynamically and transparently
  - For **responsibilities** that can be **withdrawn**
  - When **extension** by subclassing is **impractical**

Pros and Cons

Pros

- **Alternative to subclassing** for extending functionality
- Supports the principle that **classes should be open for extension but closed for modification**

Cons

- **Many small objects** in our design
- Can cause **issues** if the client relies heavily on the **components concrete type**
- Can complicate the process of **instantiating the component**

Example

```
let User = function (name) {
  this.name = name;
  this.say = function () { console.log("User: " + this.name); };
}
let DecoratedUser = function (user, city) {

  this.name = user.name;  // ensures interface stays the same

  this.say = function () {
    console.log(`Decorated User: ${this.name}, ${this.city}`)
  };
}
//Continues on the next slide
```

Example

```
let user = new User("Kelly");
user.say();
let decorated = new DecoratedUser(user, "New York");
decorated.say();
```

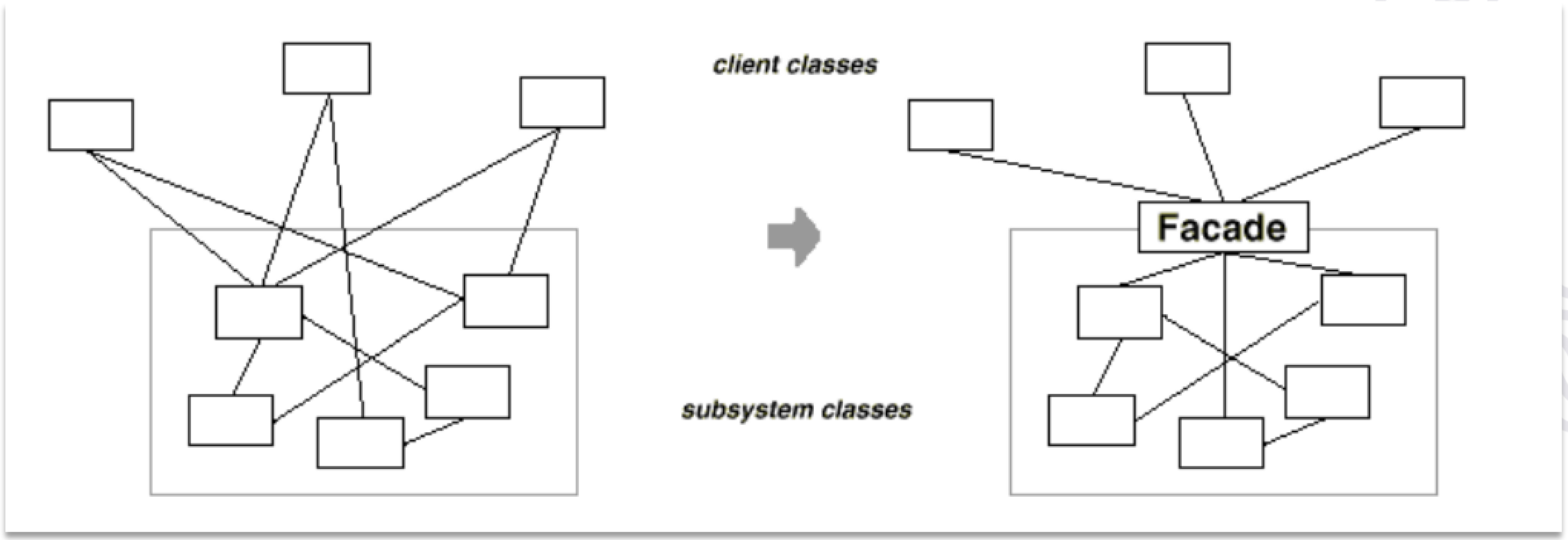
```
User: Kelly

Decorated User: Kelly, New York
```

Facade Pattern

The Facade Patters

- Widely used in the JavaScript libraries
- Provides an interface which **shields clients** from **complex functionality** in one or more subsystems



Pros and Cons

Pros

- You can **isolate** your code from the **complexity of** a **subsystem**

Cons

- facade can become **a god object** coupled to all classes of an app

Example

```
class ComplaintRegistry {
  registerComplaint(customer, type, details) {
    let registry;
    if (type === 'service') {
      registry = new ServiceComplaints();
    } else {
      registry = new ProductComplaints();
    }
    return registry.addComplaint({ id, customer, details });
  }
}
```

Summary

- Design Pattern - Reusable solution
- There are 3 different categories:
  - Creational
  - Behavioral
  - Structural

Questions?

License

KINGSLAND UNIVERSITY

THANK YOU