التكنولوجيا التطبيقية
APPLIED TECHNOLOGY
وزارة التربية والتعليم والتعليم الفني

مدارس we للتكنولوجيا التطبيقية

IT Department
Web Development and Programming

Unit 23
Software Design Patterns
الصف الثالث

Web Development and Programming
Workshops

2st

2024 - 2025

Unit 23
Software Design Patterns
الصف الثالث

Unit 23

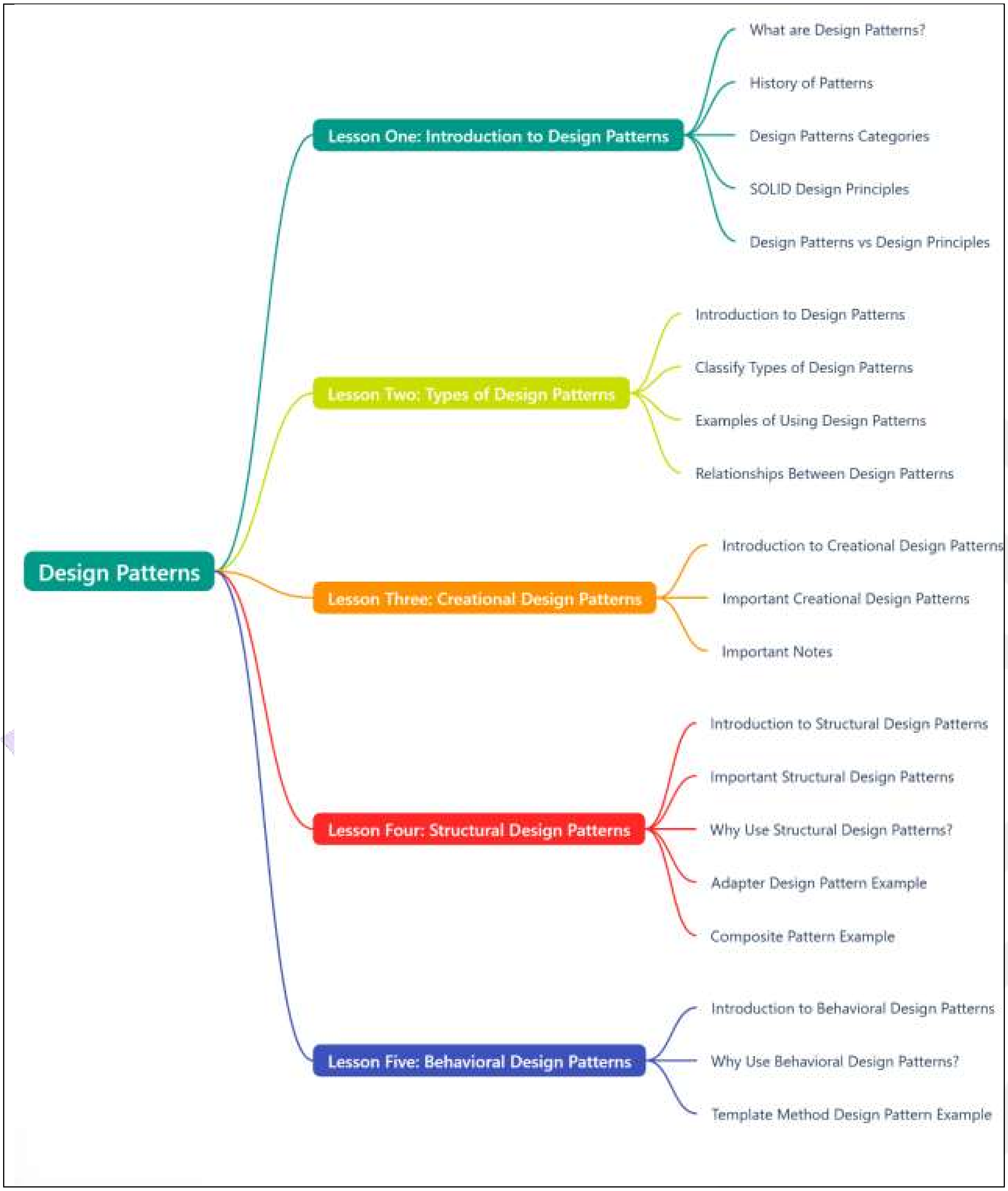
Software Design Patterns

الصف الثالث

W o r k s h o p s

Unit	**23**
Name	**Software Design Patterns**
Goals / Outcomes	➢ **Remembering** 1. Define what a design pattern is and describe its purpose in software development. 2. List the three main categories of design patterns (Creational, Structural, Behavioral). 3. Recall key characteristics of specific design patterns such as Singleton, Adapter, and Template Method. ➢ **Understanding** 1. Explain the importance of design patterns in creating scalable, maintainable, and reusable code. 2. Differentiate between the types of design patterns and their use cases. 3. Discuss the intent and mechanics of the Singleton, Adapter, and Template Method patterns. ➢ **Applying** 1. Implement the Singleton pattern to ensure a class has only one instance. 2. Use the Adapter pattern to allow two incompatible interfaces to work together. 3. Apply the Template Method pattern to define the skeleton of an algorithm while allowing flexibility in specific steps. ➢ **Analyzing** 1. Identify scenarios in which a specific design pattern is the best solution. 2. Compare and contrast Creational, Structural, and Behavioral design patterns in terms of their roles and benefits. 3. Break down the structure and implementation details of design patterns to understand their components and interactions. ➢ **Evaluating** 1. Assess the suitability of a design pattern for solving a given software problem. 2. Critique the effectiveness of using a particular pattern versus alternative solutions.

	3. Evaluate how design patterns can improve code quality, readability, and maintenance in complex systems. ➢ **<u>Creating</u>** 1. Design a small-scale software solution using multiple design patterns appropriately. 2. Extend an existing design pattern to address a unique or advanced requirement. 3. Collaborate on a team project to apply design patterns in a real-world application.	
Knowledge	**Code**	**Description**
	TPK18	Digital tools options and uses
Skill	**Code**	**Description**
	TPC3.5	Classify design patterns
	TPC3.6	Implement design patterns concepts into the projects
	TPC5.3	Develop and use simple Dashboards and simple data visualization forms
	TPC6.4	Analyze object-based design patterns
"تنبيه عام :جميع الحلول المقدمة في هذا الدليل هي استرشادية فقط. الهدف منها هو مساعدة المعلمين على شرح المفاهيم وتقديم أمثلة للطلاب. يُنصح المعلمون بتشجيع الطلاب على التفكير النقدي وحل المشكلات بطرق مختلفة".		
Unit Preface		


Design Patterns
Lesson One: Introduction to Design Patterns
What are Design Patterns?
History of Patterns
Design Patterns Categories
SOLID Design Principles
Design Patterns vs Design Principles
Lesson Two: Types of Design Patterns
Introduction to Design Patterns
Classify Types of Design Patterns
Examples of Using Design Patterns
Relationships Between Design Patterns
Lesson Three: Creational Design Patterns
Introduction to Creational Design Patterns
Important Creational Design Patterns
Important Notes
Lesson Four: Structural Design Patterns
Introduction to Structural Design Patterns
Important Structural Design Patterns
Why Use Structural Design Patterns?
Adapter Design Pattern Example
Composite Pattern Example
Lesson Five: Behavioral Design Patterns
Introduction to Behavioral Design Patterns
Why Use Behavioral Design Patterns?
Template Method Design Pattern Example

Workshop1

Question 1: Single Responsibility Principle (SRP)

Q: Why should a React component follow the Single Responsibility Principle (SRP)?

Write an example of a component that violates SRP and then refactor it to follow SRP.

A: A component should have only one reason to change. If a component handles multiple responsibilities, it becomes harder to maintain.

Bad Example (Violating SRP):

```
function UserProfile({ user }) {
  return (
    <div>
      <h2>{user.name}</h2>
      <p>Email: {user.email}</p>
      <button onClick={() => alert("Logged out")}>Logout</button>
    </div>
  );
}
```

Problem: This component displays user details **and** handles logout functionality.

Refactored Example (Following SRP):

```
function UserInfo({ user }) {
  return (
    <div>
      <h2>{user.name}</h2>
      <p>Email: {user.email}</p>
    </div>
  );
}

function LogoutButton() {
  return <button onClick={() => alert("Logged out")}>Logout</button>;
}

function UserProfile({ user }) {
  return (
    <div>
      <UserInfo user={user} />
      <LogoutButton />
    </div>
  );
}
```

Why is this better?

- UserInfo handles displaying user data.
- LogoutButton is responsible only for the logout action.
- UserProfile combines them without mixing responsibilities.

Question 2: Open/Closed Principle (OCP)

Q: How does the Open/Closed Principle (OCP) apply to React components? Show an example where a component is not following OCP and then refactor it.

A: OCP states that components should be **open for extension but closed for modification**.

Bad Example (Violating OCP):

```
function Alert({ type, message }) {
  if (type === "success") {
    return <div style={{ color: "green" }}>{message}</div>;
  } else if (type === "error") {
    return <div style={{ color: "red" }}>{message}</div>;
  } else {
    return <div>{message}</div>;
  }
}
```

Problem: Every time a new alert type is needed, we must modify the Alert component.

Refactored Example (Following OCP):

```
function Alert({ children, style }) {
  return <div style={style}>{children}</div>;
}

function SuccessAlert({ message }) {
  return <Alert style={{ color: "green" }}>{message}</Alert>;
}

function ErrorAlert({ message }) {
  return <Alert style={{ color: "red" }}>{message}</Alert>;
}
```

Why is this better?

- The Alert component is now **closed for modification** but **open for extension** (we can add new alert types without changing Alert).

Question 3: Liskov Substitution Principle (LSP)

Q: What is the Liskov Substitution Principle (LSP) in React? Give an example where a component violates LSP and then refactor it to follow LSP.

A: LSP states that a subclass (or child component) should be able to replace its parent without breaking the application.

Bad Example (Violating LSP):

```
function Button({ label, onClick }) {
  return <button onClick={onClick}>{label}</button>;
}


function DisabledButton() {
  return <button disabled>Cannot Click</button>; // Missing onClick
}
```

Problem: DisabledButton does not behave like Button. It **removes functionality**, which can break expectations.

Refactored Example (Following LSP):

```
function Button({ label, onClick, disabled = false }) {
  return <button onClick={onClick}
disabled={disabled}>{label}</button>;
}
```

Why is this better?

Button can now be used as a **normal** or **disabled** button without breaking the expected behavior.

Workshop2

Question 4: Interface Segregation Principle (ISP)

Q: How does Interface Segregation Principle (ISP) apply to React props? Show an example where a component violates ISP and refactor it.

A: ISP suggests that components should not be forced to depend on unnecessary props.

Bad Example (Violating ISP):

```
function UserCard({ name, email, phone, address, onDelete }) {
  return (
    <div>
      <h3>{name}</h3>
      <p>Email: {email}</p>
      <p>Phone: {phone}</p>
      <p>Address: {address}</p>
      <button onClick={onDelete}>Delete</button>
    </div>
  );
}
```

Problem: Every UserCard must handle onDelete, even if deletion is not needed.

Refactored Example (Following ISP):

```
function UserInfo({ name, email, phone, address }) {
  return (
    <div>
      <h3>{name}</h3>
      <p>Email: {email}</p>
      <p>Phone: {phone}</p>
      <p>Address: {address}</p>
    </div>
  );
}

function UserCardWithDelete({ user, onDelete }) {
  return (
    <div>
      <UserInfo {...user} />
      <button onClick={onDelete}>Delete</button>
    </div>
  );
}
```

Why is this better?

- UserInfo does not force delete functionality.
- UserCardWithDelete extends it **only when needed**.

Question 5: Dependency Inversion Principle (DIP)

Q: How does the Dependency Inversion Principle (DIP) improve React components? Show an example where a component violates DIP and refactor it.

A: DIP states that high-level components should not depend on low-level details. Instead, both should depend on abstractions (like props or context).

Bad Example (Violating DIP):

```
function DataFetcher() {
  const [data, setData] = React.useState(null);

  React.useEffect(() => {
    fetch("https://api.example.com/data")
      .then((response) => response.json())
      .then((data) => setData(data));
  }, []);

  return <div>{data ? JSON.stringify(data) : "Loading..."}</div>;
}
```

Problem: DataFetcher directly depends on the API call. If we change the data source, we must edit this component.

Refactored Example (Following DIP):

```
function DataDisplay({ fetchData }) {
  const [data, setData] = React.useState(null);

  React.useEffect(() => {
    fetchData().then(setData);
  }, [fetchData]);

  return <div>{data ? JSON.stringify(data) : "Loading..."}</div>;
}

// Usage with API
function App() {
  const fetchData = () => fetch("https://api.example.com/data").then(res =>
res.json());
  return <DataDisplay fetchData={fetchData} />;
}
```

Why is this better?

- DataDisplay does not depend on where data comes from.
- We can now **swap data sources** easily.

Workshop3

Question 6: Factory Pattern.

Q: Let's say we want to create different button components based on the state (like primary button, secondary button, danger button).

// Factory function

```
function createButton(type) {
   switch (type) {
      case "primary":
         return <button className="btn-primary">Primary</button>;
      case "secondary":
         return <button className="btn-
secondary">Secondary</button>;
      case "danger":
         return <button className="btn-danger">Danger</button>;
      default:
         return <button className="btn-default">Default</button>;
   }
}
```

// Use the factory

```
export default function App() {
   return (
      <div>
         {createButton("primary")}
         {createButton("danger")}
      </div>
   );
}
```

Question 7: Singleton Pattern.

Q: Setting up application state using React Context as a unified source of shared information in the application.

```
import React, { createContext, useContext, useState } from "react";
// Create a single context to store state
const AppContext = createContext();
export function AppProvider({ children }) {
   const [theme, setTheme] = useState("light");
   return (
      <AppContext.Provider value={{ theme, setTheme }}>
         {children}
      </AppContext.Provider>
   );
}
```

// Custom hook to access application context

```
export function useAppContext() {
   return useContext(AppContext);
}

// Usage
export default function App() {
   const { theme, setTheme } = useAppContext();
   return (
      <div>
         <p>Current Theme: {theme}</p>
         <button onClick={() => setTheme(theme === "light" ? "dark" :
"light")}>
            Toggle Theme
         </button>
      </div>
   );
}
```

Workshop4

Question 8: Adaptor Design Pattern.

Q: Adapting an API that provides data in a different format

// Data from API comes in different format

```
const apiData = {
   user_name: "JohnDoe",
   user_email: "john@example.com"
};
```

// Convert data to fit the component.

```
class UserAdapter {
   constructor(apiData) {
      this.name = apiData.user_name;
      this.email = apiData.user_email;
   }
}
```

// React component uses data in a new format

```
function UserInfo({ user }) {
   return (
      <div>
         <h1>User Information</h1>
         <p>Name: {user.name}</p>
         <p>Email: {user.email}</p>
      </div>
   );
}
```

```
export default function App() {
   const adaptedUser = new UserAdapter(apiData);
   return <UserInfo user={adaptedUser} />;
}
```

Question 9: Composite Pattern.

Q: Displaying nested comments that contain responses.

```
import React from "react";
function Comment({ text, replies }) {
  return (
    <div style={{ marginLeft: "20px", marginTop: "10px" }}>
      <p>{text}</p>
      {replies && replies.map((reply, index) => (
<Comment key={index} text={reply.text} replies={reply.replies} />
      ))}
    </div>
  );
}
```

```
const commentData = {
  text: "This is a main comment",
  replies: [
    {
      text: "This is a reply",
      replies: [{ text: "This is a nested reply" }]
    }
  ]
};

export default function App() {
  return (
    <div>
      <Comment text={commentData.text}
replies={commentData.replies} />
    </div>
  );
}
```

Workshop5

Question 10: template method pattern.

Q: Making a Hot Beverage Using Specific Steps

We have a base class **HotBeverage** that contains the sequence of steps to make a hot beverage, where water is boiled, ingredients are added, and the beverage is poured. Some of the steps in subclasses will be dedicated to making coffee or tea.

Step 1: Create the HotBeverage base class

```
class HotBeverage {
  prepare() {
    this.boilWater();
    this.brew();
    this.pourInCup();
    this.addCondiments();
  }

  boilWater() {
    console.log("Boiling water...");
  }

  pourInCup() {
    console.log("Pouring into cup...");
  }

  // This function will be assigned to subclasses.
  brew() {
    throw new Error("This method should be overridden in subclass.");
  }

  // This function will be assigned to subclasses.
  addCondiments() {
    throw new Error("This method should be overridden in subclass.");
  }
}
```

Step 2: Create the Tea and Coffee subclasses

We will now create the Tea and Coffee subclasses, which will customize the brew and addCondiments steps depending on the type of drink.

```
class Tea extends HotBeverage {
   brew() {
      console.log("Steeping the tea...");
   }

   addCondiments() {
      console.log("Adding lemon...");
   }
}

class Coffee extends HotBeverage {
   brew() {
      console.log("Dripping coffee through filter...");
   }

   addCondiments() {
      console.log("Adding sugar and milk...");
   }
}
```

Step 3: Using Classes in React

Now we will create a React component that uses these classes to print the steps for making coffee and tea.

```
import React, { useEffect } from "react";
export default function App() {
   useEffect(() => {
      const tea = new Tea();
      const coffee = new Coffee();

      console.log("Preparing Tea:");
      tea.prepare();

      console.log("\nPreparing Coffee:");
      coffee.prepare();
   }, []);

   return (
      <div>
         <h1>Hot Beverage Preparation</h1>
         <p>Check the console to see the preparation steps for Tea and
Coffee.</p>
      </div>
   );
}
```

Results when running
When you run the app, it will display in the console the sequence of steps for preparing tea and coffee, using the Template Method Pattern to handle each drink according to the common steps.

Expected result in Console:

```
Preparing Tea:
Boiling water...
Steeping the tea...
Pouring into cup...
Adding lemon...

Preparing Coffee:
Boiling water...
Dripping coffee through filter...
Pouring into cup...
Adding sugar and milk...
```

Description

- The HotBeverage base class defines the basic brewing steps: prepare, boilWater, pourInCup.
- The Tea and Coffee subclasses customize the brew and addCondiments steps to match the type of beverage.
- The App component creates objects of Tea and Coffee and uses a function