Lecture 10

Dr. Umair Rehman

Agenda

- Creational Design Patterns:
  - Prototype Pattern
  - Modifying Cloned Objects
  - Copy Constructors
  - Combining Creational Design Patterns
    - Factory and Prototype Pattern
    - Factory and Builder Pattern

Prototype Pattern

- Creating new objects by "cloning" existing ones
- Efficiently creating multiple instances of complex objects that need to be slightly modified rather than built from scratch each time

Prototype Pattern

- You have a complex object
- Creating a new one with same properties would take a lot of work
- Instead of building a new one each time
  - You "clone" an existing object to get a copy quickly

Prototype Pattern Analogy

- Think of a template document you reuse.

- Instead of rewriting the whole thing,
  - You make a copy and change only what you need, like the name or date.

Prototype Pattern Code Structure

- You start with a base class that has a `clone()` method.
- The `clone()` method is responsible for copying the object.
- Each subclass can implement `clone()` in its own way,
  - If it has specific cloning needs.

Prototype Pattern Code Example

- Create two shapes in C++ that can be cloned and drawn using the Prototype pattern.

Prototype Pattern Code Example

```cpp
#include <iostream>

// Step 1: Define the Prototype Interface
class Shape {
public:
    virtual ~Shape() {}
    virtual Shape* clone() const = 0; // The clone method returns a raw pointer
    virtual void draw() const = 0;
};
```

Prototype Pattern Code Example

```cpp
// Step 2: Implement Concrete Prototypes (Circle and Rectangle)
class Circle : public Shape {
public:
    Circle(float radius) : radius(radius) {}
    Shape* clone() const override {
        return new Circle(*this); // Clone by creating a new Circle object
    }
    void draw() const override {
        std::cout << "Drawing Circle with radius " << radius << "\n";
    }
private:
    float radius;
};
```

Prototype Pattern Code Example

```cpp
class Rectangle : public Shape {
public:
    Rectangle(float width, float height) : width(width), height(height) {}
    Shape* clone() const override {
        return new Rectangle(*this); // Clone by creating a new Rectangle object
    }
    void draw() const override {
        std::cout << "Drawing Rectangle with width " << width << " and height " << height << "\n";
    }
private:
    float width, height;
};

```

Prototype Pattern Code Example

```cpp
// Step 3: Use the Prototype Pattern to clone and create objects
int main() {
    // Original objects
    Shape* originalCircle = new Circle(5.0);
    Shape* originalRectangle = new Rectangle(3.0, 4.0);

    // Cloning
    Shape* clonedCircle = originalCircle->clone();
    Shape* clonedRectangle = originalRectangle->clone();

    // Test output
    originalCircle->draw();
    clonedCircle->draw(); // Should be the same as originalCircle
    originalRectangle->draw();
    clonedRectangle->draw(); // Should be the same as originalRectangle

    // Clean up memory
    delete originalCircle;
    delete originalRectangle;
    delete clonedCircle;
    delete clonedRectangle;

    return 0;
}

```

Modifying Cloned Objects

- Create two shapes in C++ that can be cloned and drawn using the Prototype pattern.

- Modify specific properties of each clone before drawing them.

We Will Use Setter Methods

- Setters are methods that update an object's private properties.

- Allow controlled access, maintaining encapsulation and data integrity.

Modifying Cloned Objects

```cpp
#include <iostream>

// Prototype Interface
class Shape {
public:
    virtual ~Shape() {}
    virtual Shape* clone() const = 0;
    virtual void draw() const = 0;
    virtual void setDimension(int newDimension) = 0;
};

```

Modifying Cloned Objects

```
// Concrete Prototype: Circle
class Circle : public Shape {
private:
    int radius;

public:
    Circle(int r) : radius(r) {}
    Shape* clone() const override {
        return new Circle(*this);
    }
    void draw() const override {
        std::cout << "Drawing a Circle with radius " << radius << std::endl;
    }
    void setDimension(int newRadius) override {
        radius = newRadius;
    }
};
```

Modifying Cloned Objects

```
// Concrete Prototype: Square
class Square : public Shape {
private:
    int side;

public:
    Square(int s) : side(s) {}
    Shape* clone() const override {
        return new Square(*this);
    }
    void draw() const override {
        std::cout << "Drawing a Square with side " << side << std::endl;
    }
    void setDimension(int newSide) override {
        side = newSide;
    }
};

```

Modifying Cloned Objects

```
int main() {
    // Create original shapes
    Shape* originalCircle = new Circle(5);
    Shape* originalSquare = new Square(4);

    // Draw original shapes
    originalCircle->draw();
    originalSquare->draw();

    // Clone shapes and modify the clones
    Shape* clonedCircle = originalCircle->clone();
    Shape* clonedSquare = originalSquare->clone();

    clonedCircle->setDimension(10);  // Modify cloned circle's radius
    clonedSquare->setDimension(8);   // Modify cloned square's side

    // Draw the modified clones
    clonedCircle->draw();
    clonedSquare->draw();

    // Clean up
    delete originalCircle;
    delete originalSquare;
    delete clonedCircle;
    delete clonedSquare;

    return 0;
```

Copy Constructors to Duplicate Objects

- Copy constructors to duplicate objects.
- This avoids the Prototype pattern but still allows duplication and modification.

Copy Constructors to Duplicate Objects

```cpp
#include <iostream>

class Shape {
public:
    virtual ~Shape() {}
    virtual void draw() const = 0;
};

class Circle : public Shape {
private:
    int radius;

public:
    Circle(int r) : radius(r) {}
    Circle(const Circle& other) : radius(other.radius) {} // Copy constructor
    void setRadius(int r) { radius = r; }
    void draw() const override {
        std::cout << "Drawing Circle with radius " << radius << std::endl;
    }
};
```

Copy Constructors to Duplicate Objects

```
int main() {
    Circle original(5);
    Circle copy = original; // Using copy constructor
    copy.setRadius(10);     // Modify copy

    original.draw();
    copy.draw();

    return 0;
}

```

Object Pooling

- Object Pool class manages a set of pre-initialized, reusable objects.
- When a client needs an object, it "borrows" one from the pool.
- When done, the client "returns" it to the pool, making it available for reuse.

Object Pooling

- This pattern is especially useful when
  - creating and destroying objects is expensive regarding time and resources (e.g., database connections, large data objects).

Object Pooling

- Imagine a database connection pool where you might need many connections over time, but creating a new connection each time is costly.

- By keeping a limited pool of active connections, you avoid the overhead of creating and destroying connections frequently.

Object Pooling

```cpp
#include <iostream>
#include <vector>

// Circle Class
class Circle {
public:
    Circle(float radius = 1.0) : radius(radius) {
        std::cout << "Creating Circle with radius: " << radius << "\n";
    }

    void setRadius(float newRadius) {
        radius = newRadius;
    }

    void draw() const {
        std::cout << "Drawing Circle with radius " << radius << "\n";
    }

private:
    float radius;
};
```

Object Pooling

```
// Object Pool Class for Circles
class CirclePool {
public:
    // Destructor to clean up any remaining circles in the pool
    ~CirclePool() {
        for (Circle* circle : pool) {
            delete circle;
        }
    }

    // Returns a Circle from the pool, creating a new one if none are available
    Circle* borrowCircle(float radius) {
        if (!pool.empty()) {
            Circle* circle = pool.back();
            pool.pop_back();           // Remove it from the pool
            circle->setRadius(radius);  // Set the desired radius
            return circle;
        } else {
            // If no Circle is available in the pool, create a new one
            return new Circle(radius);
        }
    }

    // Returns a Circle back to the pool for future reuse
    void returnCircle(Circle* circle) {
        pool.push_back(circle);
    }
```

Object Pooling

- When we need a circle, we call borrowCircle(radius).
- If there's a circle in storage, we take it out, adjust its size, and use it.
- If no circles are in storage, we make a brand-new one.
- After we're done using a circle,
  - We call `returnCircle(circle),` putting it back in the storage room for future use.

Object Pooling

```cpp
private:
    std::vector<Circle*> pool;  // Pool of reusable Circle objects (raw pointers)
};

// Main Function to Test Object Pooling
int main() {
    CirclePool circlePool;

    // Borrow a Circle with a radius of 5.0
    Circle* circle1 = circlePool.borrowCircle(5.0);
    circle1->draw();

    // Borrow another Circle with a radius of 10.0
    Circle* circle2 = circlePool.borrowCircle(10.0);
    circle2->draw();

```

Object Pooling

```

    // Return circles to the pool for reuse
    circlePool.returnCircle(circle1);
    circlePool.returnCircle(circle2);

    // Borrow another Circle (reuse the one from the pool)
    Circle* circle3 = circlePool.borrowCircle(15.0); // This should reuse circle1 or circle2
    circle3->draw();

    // Return the final Circle to the pool and clean up any resources
    circlePool.returnCircle(circle3);

    // No need for explicit deletes here as the CirclePool destructor handles cleanup
    return 0;
}

```

Pros and Cons of Copy Constructor, Object Pooling, and Prototype Pattern

- Copy Constructor
  - Pros: Efficient cloning; preserves existing object state.
  - Cons: Limited flexibility; can be costly if objects have complex dependencies.

- Object Pooling
  - Pros: Resource-efficient; reduces object creation overhead.
  - Cons: Increased code complexity; potential memory leaks if not managed well.

Pros and Cons of Copy Constructor, Object Pooling, and Prototype Pattern

- Prototype Pattern
  - Pros: Flexible cloning of complex objects; efficient with fewer constructor calls.
  - Cons: Cloning can be complex with deep copies; requires careful handling of object dependencies.

Combining Creational Design Patterns

- Can be helpful when creating complex systems that require multiple strategies for object creation.

Combining Creational Design Patterns

- Combine Builder and Factory to handle complex assembly and manage object instantiation.
  - Example: Factory creates parts, Builder assembles them.

- Use Abstract Factory and Prototype to define object families and clone base instances for variations.
  - Ideal for related products with subtle differences.

Combining Creational Design Patterns

- Pair Singleton with Factory Method or Abstract Factory to ensure a single instance while allowing instance type flexibility.

- Combine Factory Method and Builder for dynamic product customization based on runtime context.
  - Factory selects Builder based on context for varied configurations.

Combining Creational Design Patterns

- Use Builder with Prototype to modularize complex, multi-step product creation.
  - Prototype provides base; Builder customizes in steps.

- Combine Factory Method and Builder for dynamic product customization based on runtime context.
  - Factory selects Builder based on context for varied configurations.

Abstract Factory and Prototype Pattern

- Beneficial when we need to create families of related objects
  - That might require slight variations or copies of each object.

Abstract Factory and Prototype Pattern

- Imagine we're developing a UI library that supports different themes, such as Dark Theme and Light Theme.

- Each theme provides its own unique family of components, like Buttons, Textboxes, and Dropdowns.

- Each component family has subtle differences between themes (e.g., colors, sizes, styles), but their functionality remains the same.

Abstract Factory and Prototype Pattern

- Abstract Factory Pattern will define the family of UI components for each theme.

- Prototype Pattern will allow us to clone base instances of each component in a theme
  - Enabling slight variations without creating new instances from scratch.

Abstract Factory and Prototype Pattern

```
/// Abstract Prototype Interfaces for UI Components

class Button {
public:
    virtual Button* clone() const = 0;
    virtual void render() const = 0;
    virtual void setLabel(const std::string& newLabel) = 0;
    virtual ~Button() = default;
};

class Textbox {
public:
    virtual Textbox* clone() const = 0;
    virtual void render() const = 0;
    virtual void setPlaceholder(const std::string& newPlaceholder) = 0;
    virtual ~Textbox() = default;
};
```

Abstract Factory and Prototype Pattern

```cpp
/// Concrete Dark Theme Components

class DarkButton : public Button {
public:
    DarkButton() : label("Default Dark Button") {}

    Button* clone() const override {
        return new DarkButton(*this);
    }

    void render() const override {
        std::cout << "Rendering Dark Button with label: " << label << "\n";
    }

    void setLabel(const std::string& newLabel) override {
        label = newLabel;
    }

private:
    std::string label;
};
```

Abstract Factory and Prototype Pattern

```cpp
class DarkTextbox : public Textbox {
public:
    DarkTextbox() : placeholder("Dark Theme Placeholder") {}

    Textbox* clone() const override {
        return new DarkTextbox(*this);
    }

    void render() const override {
        std::cout << "Rendering Dark Textbox with placeholder: " << placeholder << "\n";
    }

    void setPlaceholder(const std::string& newPlaceholder) override {
        placeholder = newPlaceholder;
    }

private:
    std::string placeholder;
};

```

Abstract Factory and Prototype Pattern

```cpp
/// Concrete Light Theme Components

class LightButton : public Button {
public:
    LightButton() : label("Default Light Button") {}

    Button* clone() const override {
        return new LightButton(*this);
    }

    void render() const override {
        std::cout << "Rendering Light Button with label: " << label << "\n";
    }

    void setLabel(const std::string& newLabel) override {
        label = newLabel;
    }

private:
    std::string label;
};
```

Abstract Factory and Prototype Pattern

```
class LightTextbox : public Textbox {
public:
    LightTextbox() : placeholder("Light Theme Placeholder") {}

    Textbox* clone() const override {
        return new LightTextbox(*this);
    }

    void render() const override {
        std::cout << "Rendering Light Textbox with placeholder: " << placeholder << "\n";
    }

    void setPlaceholder(const std::string& newPlaceholder) override {
        placeholder = newPlaceholder;
    }

private:
    std::string placeholder;
};
```

Abstract Factory and Prototype Pattern

```
/// Abstract Factory Interface

class ThemeFactory {
public:
    virtual Button* createButton() const = 0;
    virtual Textbox* createTextbox() const = 0;
    virtual ~ThemeFactory() = default;
};

/// Concrete Theme Factories

class DarkThemeFactory : public ThemeFactory {
public:
    Button* createButton() const override {
        return new DarkButton();
    }

    Textbox* createTextbox() const override {
        return new DarkTextbox();
    }
};
```

Abstract Factory and Prototype Pattern

```cpp
class LightThemeFactory : public ThemeFactory {
public:
    Button* createButton() const override {
        return new LightButton();
    }

    Textbox* createTextbox() const override {
        return new LightTextbox();
    }
};

/// Factory Selector Function to Decide Theme

ThemeFactory* getThemeFactory(const std::string& themeType) {
    if (themeType == "dark") {
        return new DarkThemeFactory();
    } else if (themeType == "light") {
        return new LightThemeFactory();
    } else {
        throw std::invalid_argument("Unknown theme type");
    }
}
```

Abstract Factory and Prototype Pattern

```
int main() {
    // Determine theme type (this could come from user input or configuration)
    std::string themeType = "dark"; // Can be "light" as well

    // Obtain the appropriate ThemeFactory based on themeType
    ThemeFactory* themeFactory = getThemeFactory(themeType);

    // Create base components
    Button* baseButton = themeFactory->createButton();
    Textbox* baseTextbox = themeFactory->createTextbox();
```

Abstract Factory and Prototype Pattern

```
    // Clone and customize components for variations
    Button* saveButton = baseButton->clone();
    saveButton->setLabel("Save");

    Button* cancelButton = baseButton->clone();
    cancelButton->setLabel("Cancel");

    Textbox* usernameTextbox = baseTextbox->clone();
    usernameTextbox->setPlaceholder("Enter Username");

    Textbox* passwordTextbox = baseTextbox->clone();
    passwordTextbox->setPlaceholder("Enter Password");

    // Render all components to check their customizations
    saveButton->render();        // Expected: Rendering Dark Button with label: Save
    cancelButton->render();      // Expected: Rendering Dark Button with label: Cancel
    usernameTextbox->render();   // Expected: Rendering Dark Textbox with placeholder: Enter Username
    passwordTextbox->render();   // Expected: Rendering Dark Textbox with placeholder: Enter Password
```

Factory and Prototype Pattern

- ThemeFactory can create base prototypes for DarkButton and LightButton.

- Factory initializes and provides themed buttons as prototypes
  - Prototype clones the buttons to create variations (like changing text or icon) without reinitializing.

Factory and Builder Pattern

- Complex Objects Need to be Assembled in a Step-by-Step Process
- Initial Setup Requires Choosing Among Variants.
- Customization or Optional Features Are Needed After Selecting a Base Type

Factory and Builder Pattern: Scenario

- We have a car factory that produces different types of cars:
  - Sedans, SUVs, and Electric Cars.
- Each car type has customizable features such as color and GPS .
- We want the system to:
  - Choose the correct type of car to produce.
- Allow customization of the car's features.
- Maintain a clean and modular design by separating concerns.

Factory and Builder Pattern

- Factory Pattern helps us select and create the car type based on user needs.

- Once a car type is selected, we use the Builder Pattern to customize the car with specific features

Factory and Builder Pattern

```cpp
// Core component - Engine
class Engine {
public:
    Engine(const std::string &type) : type(type) {}
    std::string getType() const { return type; }
private:
    std::string type;
};

// Base Car class with Engine as a core component
class Car {
public:
    virtual void specifications() const = 0;
    virtual ~Car() {
        delete engine;
    }
protected:
    Engine* engine = nullptr;
};

```

Factory and Builder Pattern

```
// Sedan, SUV, ElectricCar classes, each with a default engine
class Sedan : public Car {
public:
    Sedan() {
        engine = new Engine("I4");
    }
    void specifications() const override {
        std::cout << "Sedan with engine type: " << engine->getType() << "\n";
    }
};

class SUV : public Car {
public:
    SUV() {
        engine = new Engine("V6");
    }
    void specifications() const override {
        std::cout << "SUV with engine type: " << engine->getType() << "\n";
    }
};
```

Factory and Builder Pattern

```
class ElectricCar : public Car {
public:
    ElectricCar() {
        engine = new Engine("Electric");
    }
    void specifications() const override {
        std::cout << "Electric car with engine type: " << engine->getType() << "\n";
    }
};
```

Factory and Builder Pattern

```cpp
// Car Factory - responsible for creating the correct car type
class CarFactory {
public:
    static Car* createCar(const std::string &type) {
        if (type == "Sedan") return new Sedan();
        else if (type == "SUV") return new SUV();
        else if (type == "Electric") return new ElectricCar();
        return nullptr;
    }
};
```

Factory and Builder Pattern

```cpp
// CarBuilder - responsible only for optional features
class CarBuilder {
public:
    CarBuilder(Car* car) : car(car) {}

    CarBuilder& addGPS() {
        gpsInstalled = true;
        return *this;
    }

    CarBuilder& addSunroof() {
        sunroofInstalled = true;
        return *this;
    }

    void showSpecifications() const {
        car->specifications();
        if (gpsInstalled) std::cout << "Feature: GPS\n";
        if (sunroofInstalled) std::cout << "Feature: Sunroof\n";
    }
```

Factory and Builder Pattern

```
    ~CarBuilder() {
        delete car;
    }

private:
    Car* car;
    bool gpsInstalled = false;
    bool sunroofInstalled = false;
};

```

Factory and Builder Pattern

```cpp
// Client code
int main() {
    // Step 1: Car selection and creation through the factory
    Car* chosenCar = CarFactory::createCar("SUV");

    // Step 2: Car customization with CarBuilder
    CarBuilder builder(chosenCar);
    builder.addGPS().addSunroof();

    // Display the fully customized car specifications
    builder.showSpecifications();

    // Cleanup of chosenCar is handled in the CarBuilder destructor
    return 0;
}
```

Factory and Builder Pattern

- CarFactory selects the type of car to create (SUV, Sedan).

- Factory produces the basic car model,
    - While Builder adds optional features, like GPS or a sunroof, based on customer specifications.

Conclusion

- Creational Design Patterns
  - Prototype Pattern
  - Modifying Cloned Objects
  - Copy Constructors
  - Combining Creational Design Patterns
    - Factory and Prototype Pattern
    - Factory and Builder Pattern