Faculty of Informatics and Computer Science

Software Construction & Testing

Topics 5 & 6

Design Patterns

Dr. Ahmed Maghawry

Contents

1. Introduction

"Design Patterns"
refer to **reusable solutions** to common problems that arise during software development.

Design patterns provide a structured approach to designing and implementing software systems, helping to **improve** code quality, maintainability, and scalability.

Design patterns categories, including: creational, structural, and behavioral patterns.

- **Creational patterns:** focus on object creation mechanisms, helping to create objects in a flexible and reusable way.
- **Structural patterns:** deal with the composition of classes and objects, defining relationships between them to form larger structures.
- **Behavioral patterns:** address how objects communicate and interact with each other.

Some commonly known design patterns include:

- **Singleton pattern.**
- **Factory pattern.**
- **Observer pattern.**
- **Strategy pattern.**

Each pattern has its own purpose and usage scenario, and they can be combined and adapted to suit specific software development needs.

2. Categories and Types of Design Patterns

- **Creational Patterns:**
  - **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
  - **Factory** Method: Defines an interface for creating objects but lets subclasses decide which class to instantiate.
  - **Builder**: Separates the construction of complex objects from their representation, allowing the same construction process to create different representations.
- **Structural Patterns:**
  - **Adapter**: Converts the interface of a class into another interface that clients expect.
  - **Decorator**: Dynamically adds responsibilities to objects by wrapping them with additional behavior.
  - **Proxy**: Provides a surrogate or placeholder for another object to control access to it.
- **Behavioral Patterns:**
  - **Observer**: Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
  - **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.
  - **Command**: Encapsulates a request as an object, thereby allowing users to parameterize clients with queues, requests, and operations.

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - **Singleton**
  - Factory
  - Builder
- Structural Patterns:
  - Adapter
  - Decorator
  - Proxy
- Behavioral Patterns:
  - Observer
  - Strategy
  - Command

```
public class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}

// Usage:
Singleton singleton = Singleton.Instance;
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - **Factory**
  - Builder
- Structural Patterns:
  - Adapter
  - Decorator
  - Proxy
- Behavioral Patterns:
  - Observer
  - Strategy
  - Command

```
public interface IProduct
{
    void Operation();
}

public class ConcreteProduct : IProduct
{
    public void Operation()
    {
        Console.WriteLine("ConcreteProduct.Operation");
    }
}

public abstract class Creator
{
    public abstract IProduct FactoryMethod();

    public void SomeOperation()
    {
        IProduct product = FactoryMethod();
        product.Operation();
    }
}
```

```
public class ConcreteCreator : Creator
{
    public override IProduct FactoryMethod()
    {
        return new ConcreteProduct();
    }
}

// Usage:
Creator creator = new ConcreteCreator();
creator.SomeOperation();
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - Factory
  - **Builder**
- Structural Patterns:
  - Adapter
  - Decorator
  - Proxy
- Behavioral Patterns:
  - Observer
  - Strategy
  - Command

```
public class Product
{
    private string partA;
    private string partB;

    public void SetPartA(string partA)
    {
        this.partA = partA;
    }

    public void SetPartB(string partB)
    {
        this.partB = partB;
    }

    public void Show()
    {
        Console.WriteLine($"Part A: {partA}\nPart B: {partB}")
    }
}
```

```
public abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract Product GetResult();
}

public class ConcreteBuilder : Builder
{
    private Product product = new Product();

    public override void BuildPartA()
    {
        product.SetPartA("Part A");
    }

    public override void BuildPartB()
    {
        product.SetPartB("Part B");
    }

    public override Product GetResult()
    {
        return product;
    }
}
```

```
public class Director
{
    public void Construct(Builder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}

// Usage:
Director director = new Director();
Builder builder = new ConcreteBuilder();

director.Construct(builder);

Product product = builder.GetResult();
product.Show();
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - Factory
  - Builder
- Structural Patterns:
  - **Adapter**
  - Decorator
  - Proxy
- Behavioral Patterns:
  - Observer
  - Strategy
  - Command

```
public class Target
{
    public virtual void Request()
    {
        Console.WriteLine("Target.Request");
    }
}

public class Adaptee
{
    public void SpecificRequest()
    {
        Console.WriteLine("Adaptee.SpecificRequest");
    }
}
```

```
public class Adapter : Target
{
    private Adaptee adaptee = new Adaptee();

    public override void Request()
    {
        adaptee.SpecificRequest();
    }
}

// Usage:
Target target = new Adapter();
target.Request();
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - Factory
  - Builder
- Structural Patterns:
  - Adapter
  - **Decorator**
  - Proxy
- Behavioral Patterns:
  - Observer
  - Strategy
  - Command

```
public interface IComponent
{
    void Operation();
}

public class ConcreteComponent : IComponent
{
    public void Operation()
    {
        Console.WriteLine("ConcreteComponent.Operation");
    }
}
```

```
public abstract class Decorator : IComponent
{
    protected IComponent component;

    public Decorator(IComponent component)
    {
        this.component = component;
    }

    public virtual void Operation()
    {
        component.Operation();
    }
}
```

```
public class ConcreteDecorator : Decorator
{
    public ConcreteDecorator(IComponent component) : base
(component)
    {
    }

    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("ConcreteDecorator.Operation");
    }
}

// Usage:
IComponent component = new ConcreteComponent();
IComponent decorator = new ConcreteDecorator(component);

decorator.Operation();
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - Factory
  - Builder
- Structural Patterns:
  - Adapter
  - Decorator
  - **Proxy**
- Behavioral Patterns:
  - Observer
  - Strategy
  - Command

```
public interface ISubject
{
    void Request();
}

public class RealSubject : ISubject
{
    public void Request()
    {
        Console.WriteLine("RealSubject.Request");
    }
}
```

```
public class Proxy : ISubject
{
    private RealSubject realSubject = new RealSubject();

    public void Request()
    {
        // Perform additional actions before delegating to
the real subject
        Console.WriteLine("Proxy.Request");

        realSubject.Request();

        // Perform additional actions after the real subje
ct has performed its operation
    }
}

// Usage:
ISubject subject =new Proxy();
subject.Request();
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - Factory
  - Builder
- Structural Patterns:
  - Adapter
  - Decorator
  - Proxy
- Behavioral Patterns:
  - **Observer**
  - Strategy
  - Command

```
public interface IObserver
{
    void Update();
}

public class ConcreteObserver : IObserver
{
    public void Update()
    {
        Console.WriteLine("ConcreteObserver.Update");
    }
}

public interface ISubject
{
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify();
}
```

```
public class ConcreteSubject : ISubject
{
    private List<IObserver> observers = new List<IObserver
>();

    public void Attach(IObserver observer)
    {
        observers.Add(observer);
    }

    public void Detach(IObserver observer)
    {
        observers.Remove(observer);
    }

    public void Notify()
    {
        foreach (IObserver observer in observers)
        {
            observer.Update();
        }
    }
}

// Usage:
ISubject subject = new ConcreteSubject();
IObserver observer = new ConcreteObserver();

subject.Attach(observer);
subject.Notify();
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - Factory
  - Builder
- Structural Patterns:
  - Adapter
  - Decorator
  - Proxy
- Behavioral Patterns:
  - Observer
  - **Strategy**
  - Command

```
public interface IStrategy
{
    void Execute();
}

public class ConcreteStrategyA : IStrategy
{
    public void Execute()
    {
        Console.WriteLine("ConcreteStrategyA.Execute");
    }
}

public class ConcreteStrategyB : IStrategy
{
    public void Execute()
    {
        Console.WriteLine("ConcreteStrategyB.Execute");
    }
}
```

```
public class Context
{
    private IStrategy strategy;

    public Context(IStrategy strategy)
    {
        this.strategy = strategy;
    }

    public void SetStrategy(IStrategy strategy)
    {
        this.strategy = strategy;
    }

    public void ExecuteStrategy()
    {
        strategy.Execute();
    }
}

// Usage:
Context context = new Context(new ConcreteStrategyA());
context.ExecuteStrategy();

context.SetStrategy(new ConcreteStrategyB());
context.ExecuteStrategy();
```

3. Applying Design Patterns in Software Construction

- Creational Patterns:
  - Singleton
  - Factory
  - Builder
- Structural Patterns:
  - Adapter
  - Decorator
  - Proxy
- Behavioral Patterns:
  - Observer
  - Strategy
  - **Command**

```
public interface ICommand
{
    void Execute();
}

public class Receiver
{
    public void Action()
    {
        Console.WriteLine("Receiver.Action");
    }
}

public class ConcreteCommand : ICommand
{
    private Receiver receiver;

    public ConcreteCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }

    public void Execute()
    {
        receiver.Action();
    }
}
```

```
public class Invoker
{
    private ICommand command;

    public void SetCommand(ICommand command)
    {
        this.command = command;
    }

    public void ExecuteCommand()
    {
        command.Execute();
    }
}

// Usage:
Receiver receiver = new Receiver();
ICommand command = new ConcreteCommand(receiver);
Invoker invoker = new Invoker();

invoker.SetCommand(command);
invoker.ExecuteCommand();
```

4. Conclusion

- **Creational Patterns:**
  - **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
  - **Factory** Method: Defines an interface for creating objects but lets subclasses decide which class to instantiate.
  - **Builder**: Separates the construction of complex objects from their representation, allowing the same construction process to create different representations.

- **Structural Patterns:**
  - **Adapter**: Converts the interface of a class into another interface that clients expect.
  - **Decorator**: Dynamically adds responsibilities to objects by wrapping them with additional behavior.
  - **Proxy**: Provides a surrogate or placeholder for another object to control access to it.

- **Behavioral Patterns:**
  - **Observer**: Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
  - **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.
  - **Command**: Encapsulates a request as an object, thereby allowing users to parameterize clients with queues, requests, and operations.

Faculty of Informatics and Computer Science

Questions?