Design Patterns

SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Table of Contents

Have a Question?

sli.do

#csharp-advanced

Definition, Solutions and Elements

Design Patterns

What Are Design Patterns?

- **General** and **reusable solutions** to common problems in software design
- A **template** for solving given problems
- Add additional layers of **abstraction** in order to reach flexibility

What Do Design Patterns Solve?

- Patterns solve **software structural problems** like:
  - Abstraction
  - Encapsulation
  - Separation of concerns
  - Coupling and cohesion
  - Separation of interface and implementation

Elements of a Design Pattern

- Pattern name - Increases **vocabulary** of designers
- Problem - **Intent**, context and when to apply
- Solution - **Abstract** code
- Consequences - **Results** and trade-offs

Benefits and Drawbacks

Why Design Patterns?

Benefits

- Names form a common vocabulary
- Enable large-scale **reuse** of software architectures
- Help improve developer **communication**
- Can **speed-up** the development

Drawbacks

- Do not lead to a direct code reuse
- Deceptively simple
- Developers may suffer from **pattern overload** and **overdesign**
- Validated by **experience** and discussion, not by automated testing
- Should be used only if **understood well**

Types of Design Patterns

Main Types

- Creational patterns
  - Deal with **initialization and configuration** of classes and objects
- Structural patterns
  - Describe ways to **assemble** objects to implement **new functionality**
  - **Composition** of classes and objects
- Behavioral patterns
  - Deal with dynamic **interactions** among societies of classes
  - Distribute **responsibility**

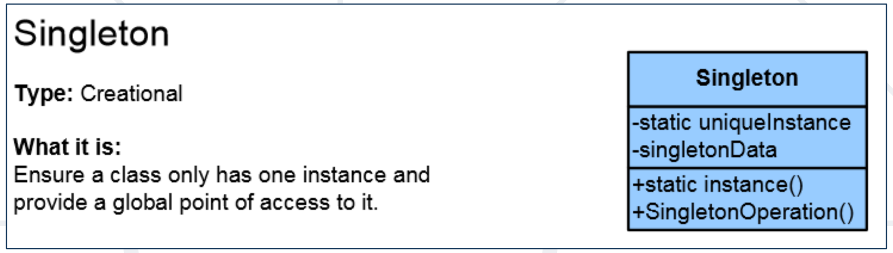
Creational Patterns

Purposes

- Deal with **object creation** mechanisms
- Trying to create objects in a **manner suitable** to the **situation**
- Two main ideas
  - **Encapsulating** knowledge about which classes the system uses
  - **Hiding** how instances of these classes are created

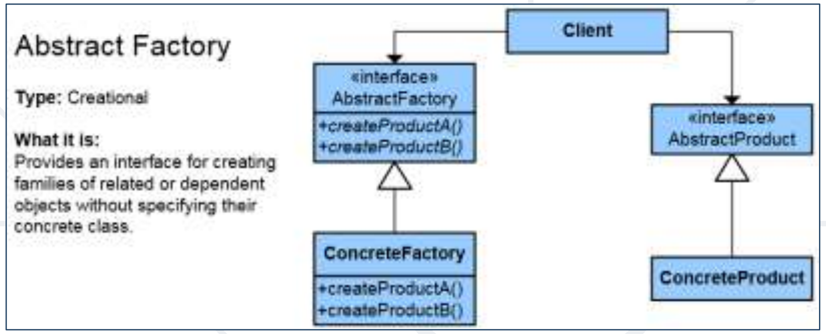
List of Creational Patterns

- Singleton
- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Fluent Interface
- Object Pool
- Lazy Initialization


Singleton

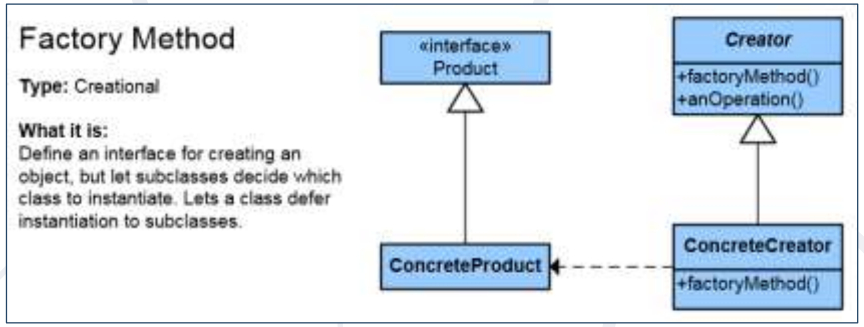
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.


Abstract Factory

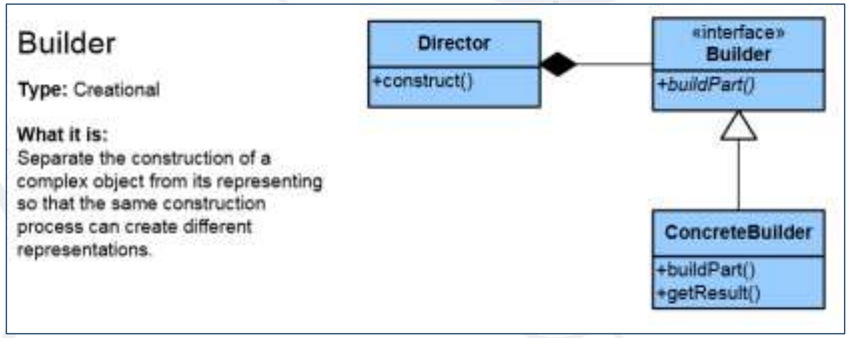
Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.


Factory Method

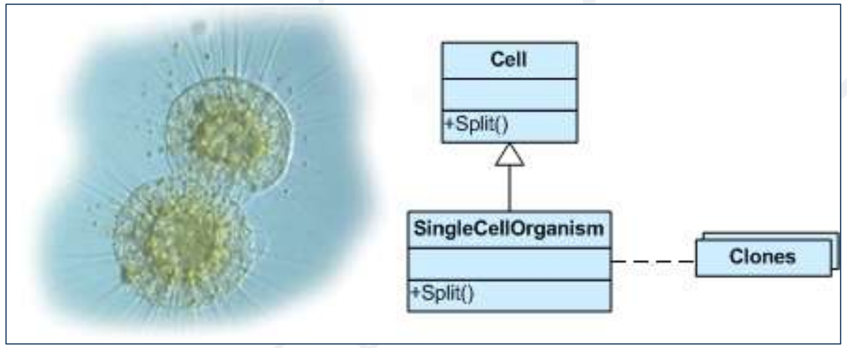
Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.


Builder

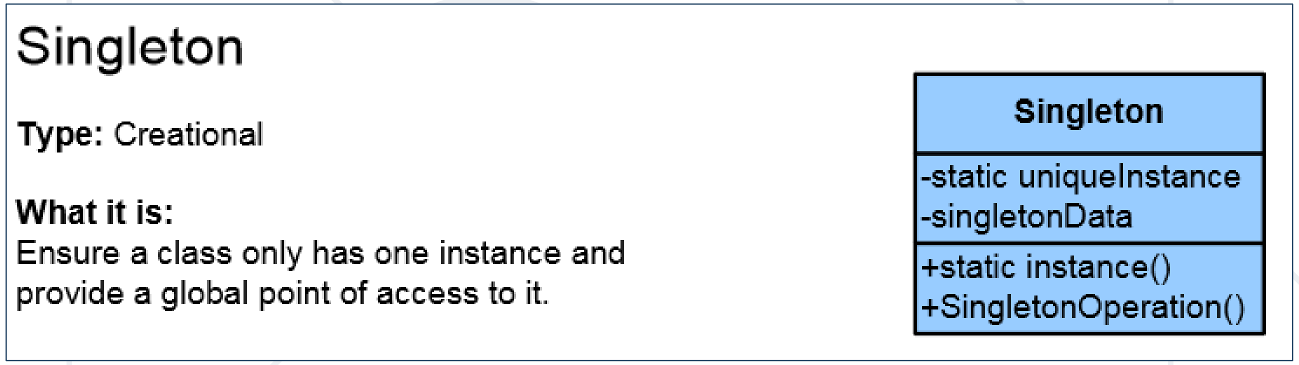
Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Singleton Pattern

- The most often used creational design pattern
- A Singleton class is supposed to have **only one instance**
- It is **not a global variable**
- Possible problems
  - Lazy loading
  - Thread-safe



Singleton

Type: Creational

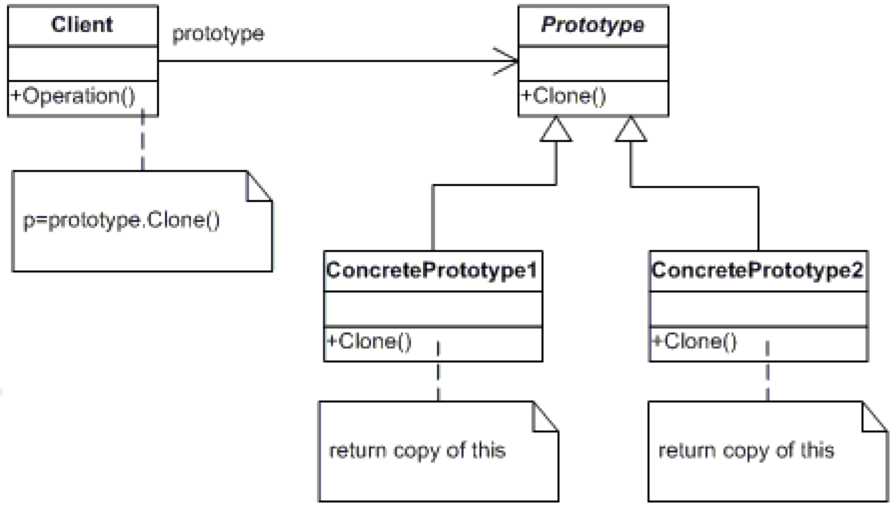
What it is:
Ensure a class only has one instance and provide a global point of access to it.

Double-Check Singleton Example

```
public sealed class Singleton {
  private static Singleton instance;
  private static readonly object padlock = new object();
  private Singleton() { }
  public static Singleton Instance {
    get {
      if (instance == null) {
        lock (padlock) {
          if (instance == null)
            instance = new Singleton(); } }
      return instance; } } }
```

Prototype Pattern

- Factory for **cloning** new instances from a prototype
  - Create new objects by copying this prototype
  - Instead if using the "new" keyword
- **ICloneable** interface acts as Prototype



The Prototype Abstract Class

```
abstract class Prototype {
  private string _id;

  public Prototype(string id) {
    this._id = id; }

  public string Id => this._id;

  public abstract Prototype Clone();
}
```

A Concrete Prototype Class

```
class ConcretePrototype : Prototype
{
  public ConcretePrototype(string id) : base(id) { }

  public override Prototype Clone()
    => return (Prototype)this.MemberwiseClone();
}
```

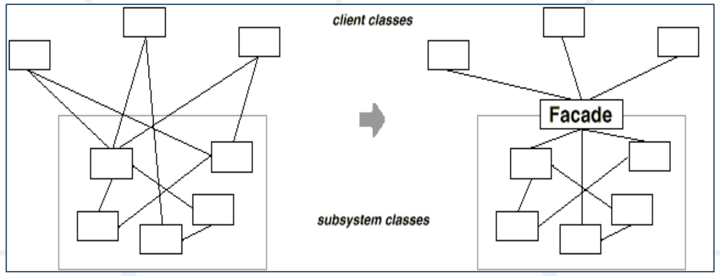
Structural Patterns

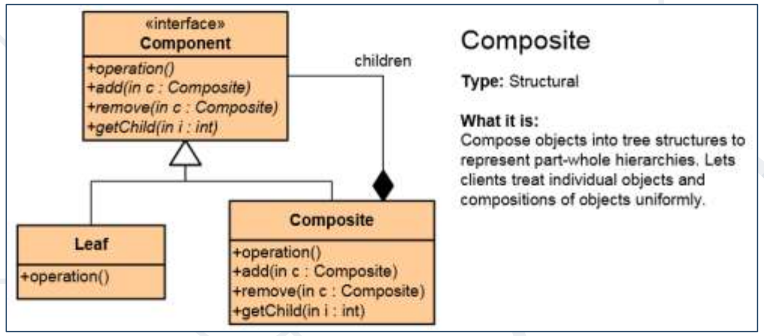
Purposes

- Describe ways to assemble **objects** to implement a **new functionality**
- Ease the design by identifying a simple way to realize **relationship** between entities
- All about Class and Object composition
  - **Inheritance** to compose interfaces
  - Ways to compose objects to obtain **new functionality**

List of Structural Patterns

- Facade
- Composite
- Flyweight
- Proxy
- Decorator
- Adapter
- Bridge

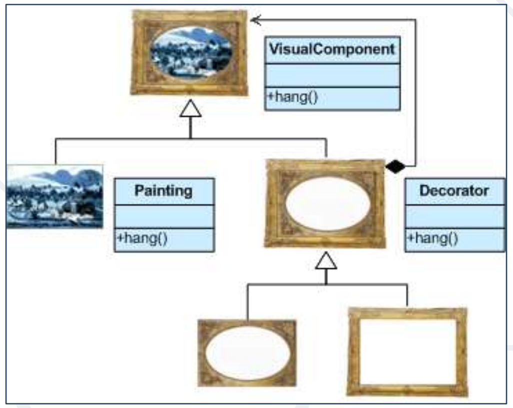


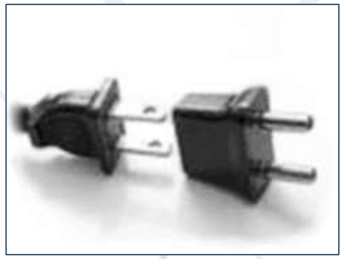


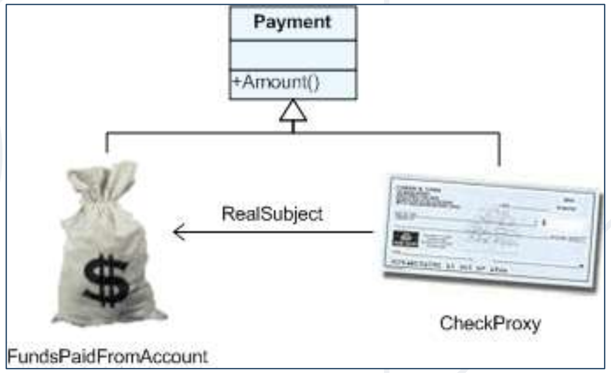
Composite

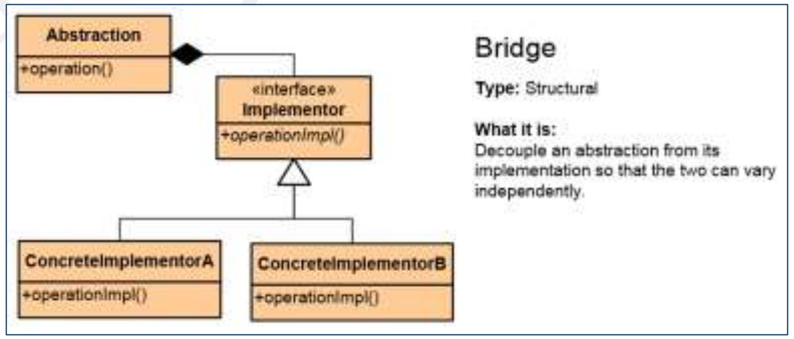
Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.









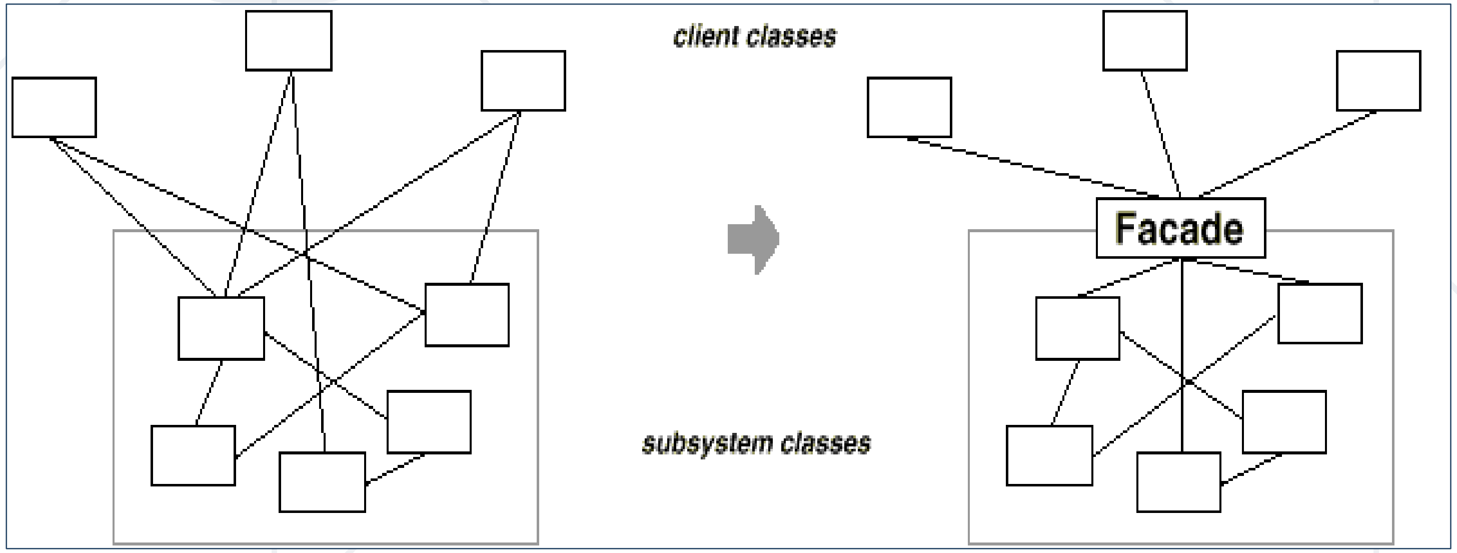
Bridge

Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Facade Pattern

- Provides a **unified interface** to a set of interfaces in a subsystem
- Defines a **higher-level interface** that makes the subsystem easier to use



The Facade Class (1)

```
class Facade
{
  private SubSystemOne _one;
  private SubSystemTwo _two;

  public Facade()
  {
    _one = new SubSystemOne();
    _two = new SubSystemTwo();
  }
```

The Facade Class (2)

```
  public void MethodA() {
    Console.WriteLine("\nMethodA() ---- ");
    _one.MethodOne();
    _two.MethodTwo(); }

  public void MethodB() {
    Console.WriteLine("\nMethodB() ---- ");
    _two.MethodTwo(); }
}
```

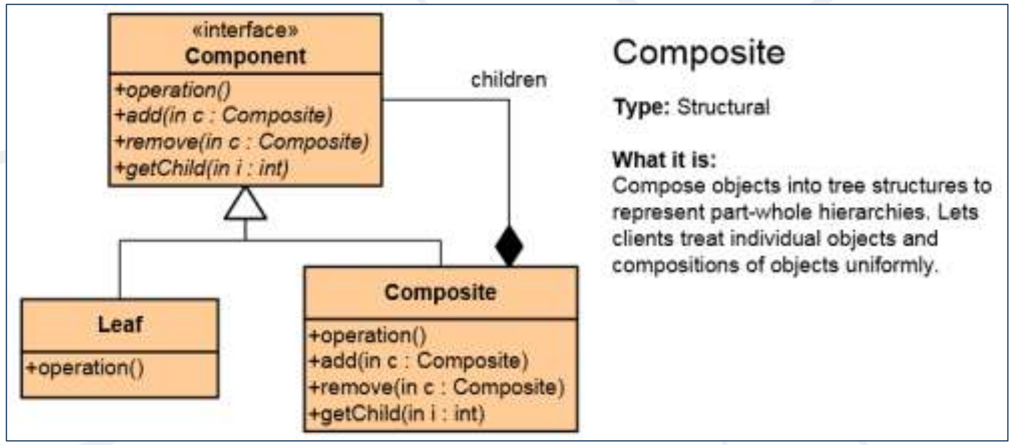
Subsystem Classes

```
class SubSystemOne
{
  public void MethodOne()
    => Console.WriteLine(" SubSystemOne Method");
}
```

```
class SubSystemTwo
{
  public void MethodTwo()
    => Console.WriteLine(" SubSystemTwo Method");
}
```

Composite Pattern

- Allows to **combine** different types of objects in tree structures
- Gives the possibility to treat the **same object(s)**
- Used when
  - You have different objects that you want to **treat the same way**
  - You want to present **hierarchy** of objects



Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

The Component Abstract Class

```
abstract class Component {
  protected string name;

  public Component(string name) {
      this.name = name; }

  public abstract void Add(Component c);
  public abstract void Remove(Component c);
  public abstract void Display(int depth);
}
```

The Composite Class (1)

```
class Composite : Component {
  private List<Component> _children = new List<Component>();

  public Composite(string name) : base(name) { }

  public override void Add(Component component)
      => _children.Add(component);

  public override void Remove(Component component)
      => _children.Remove(component);
```

The Composite Class (2)

```
public override void Display(int depth)
  {
    Console.WriteLine(new String('-', depth) + name);

    foreach (Component component in _children)
    {
      component.Display(depth + 2);
    }
  }
}
```

The Leaf Class

```
class Leaf : Component {
  public Leaf(string name) : base(name) { }

  public override void Add(Component c)
    => Console.WriteLine("Cannot add to a leaf");
  public override void Remove(Component c)
    => Console.WriteLine("Cannot remove from a leaf");
  public override void Display(int depth)
    => Console.WriteLine(new String('-', depth) + name);
}
```

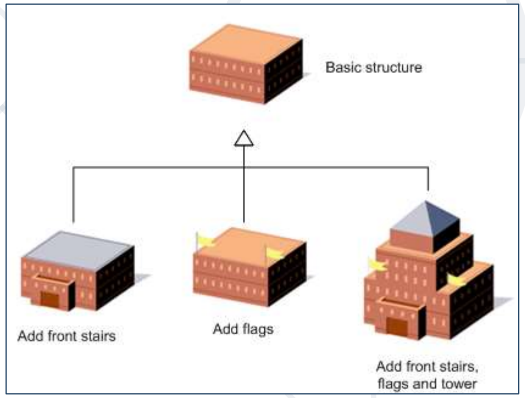
Behavioral Patterns

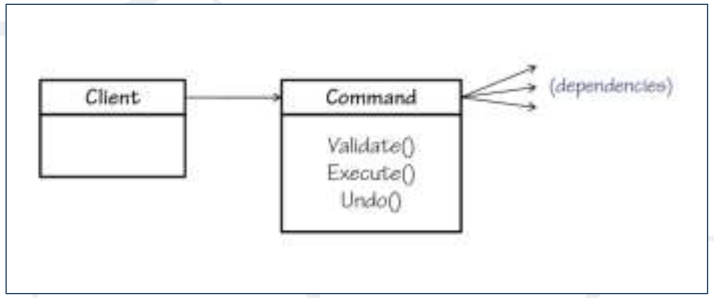
Purposes

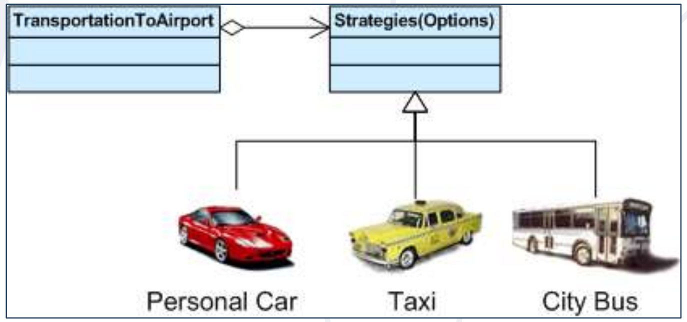
- Concerned with **interaction** between objects
  - Either with the **assignment of responsibilities** between objects
  - Or **encapsulating behavior** in an object and delegating requests to it
- Increases **flexibility** in carrying out cross-classes communication

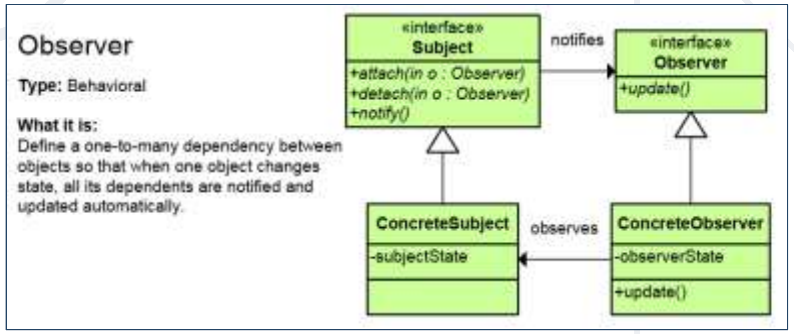
List of Behavioral Patterns (1)

- Chain of Responsibility
- Iterator
- Command
- Template Method
- Strategy
- Observer







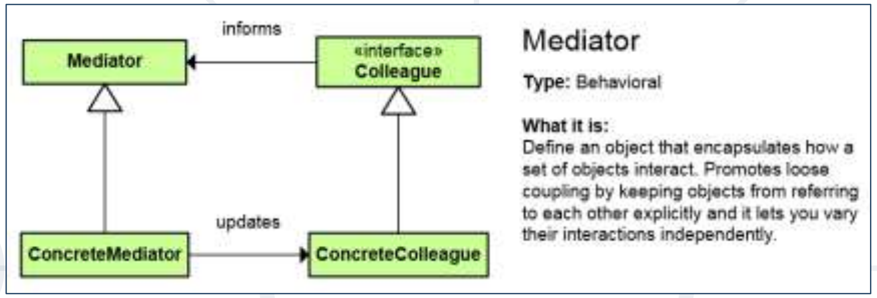

Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

List of Behavioral Patterns (2)

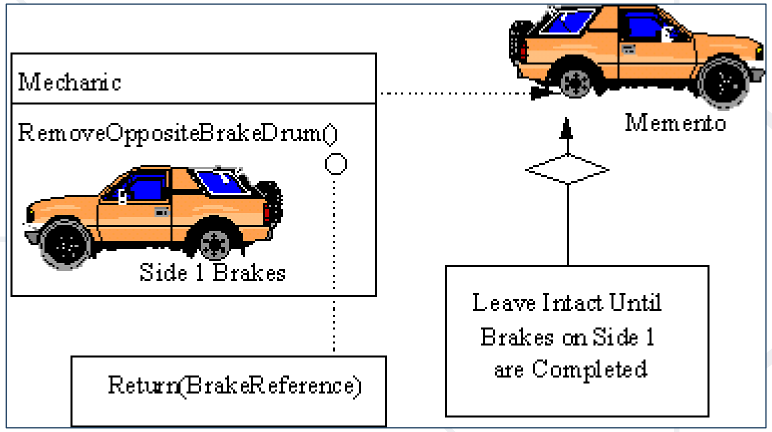
- Mediator
- Memento
- State
- Interpreter
- Visitor

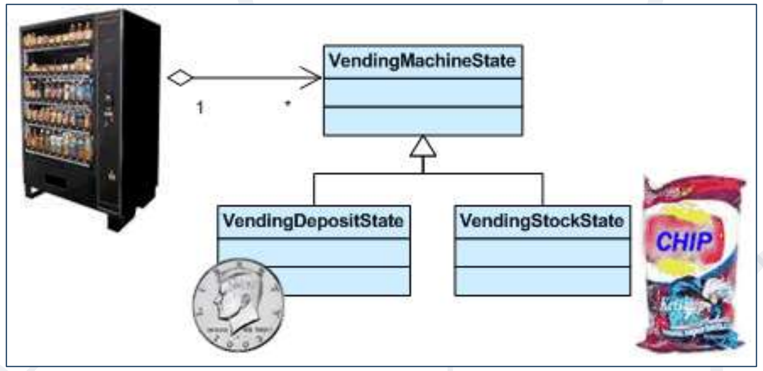


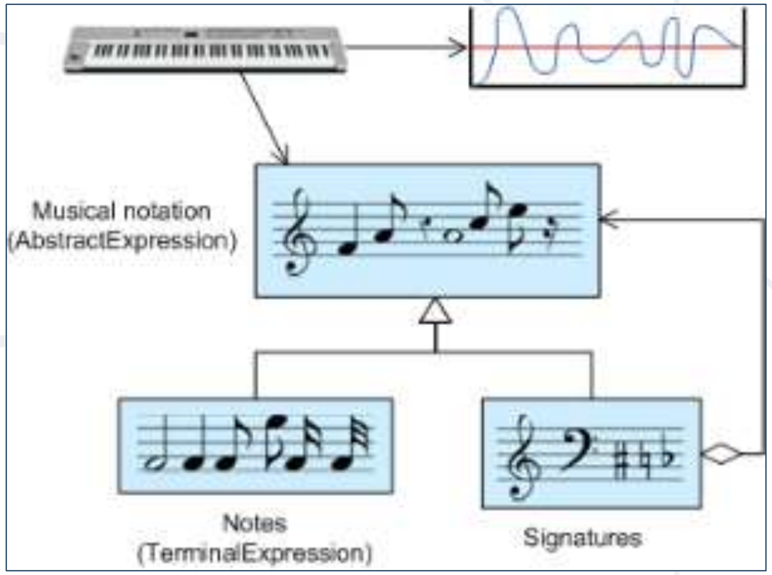
Mediator

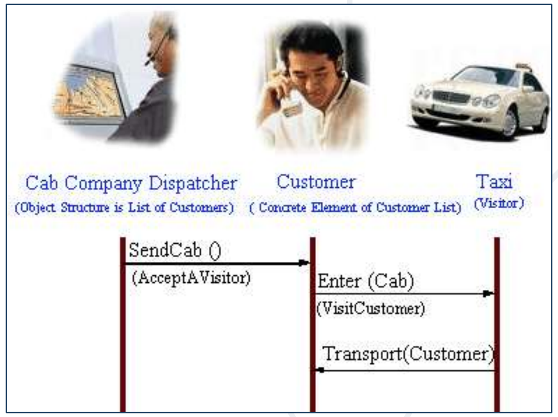
Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.


Mechanic
RemoveOppositeBrakeDrum()
Side 1 Brakes
Return(BrakeReference)
Memento
Leave Intact Until Brakes on Side 1 are Completed

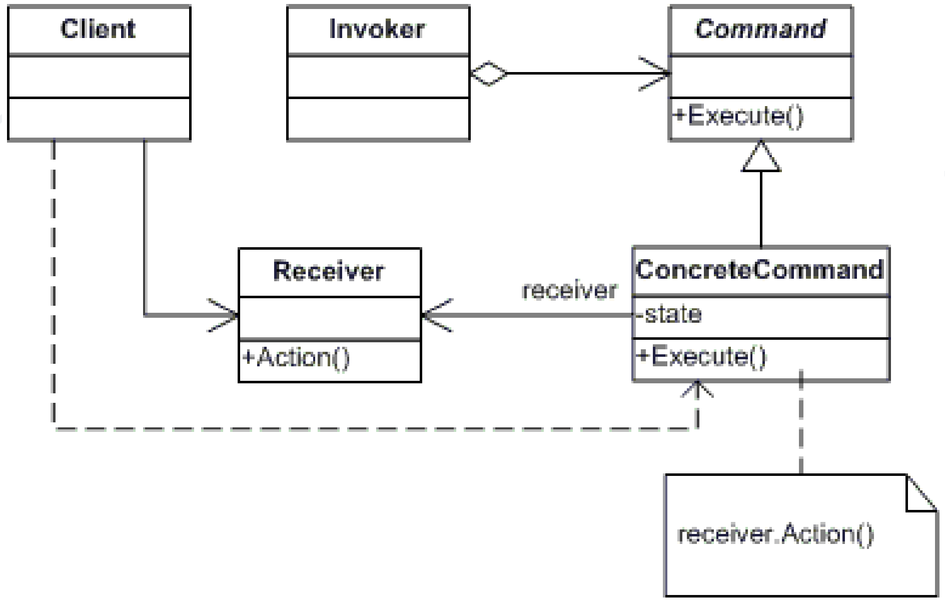



Musical notation (AbstractExpression)
Notes (TerminalExpression)
Signatures



Command Pattern

- An object **encapsulates** all the information needed to call a method at a later time
- Lets you **parameterize** clients with different requests, queue or log requests, and support undoable operations



The Command Abstract Class

```
abstract class Command
{
  protected Receiver receiver;

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

  public abstract void Execute();
}
```

Concrete Command Class

```
class ConcreteCommand : Command
{
  public ConcreteCommand(Receiver receiver)
    : base(receiver) { }

  public override void Execute()
    => receiver.Action();
}
```

The Receiver Class

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

The Invoker Class

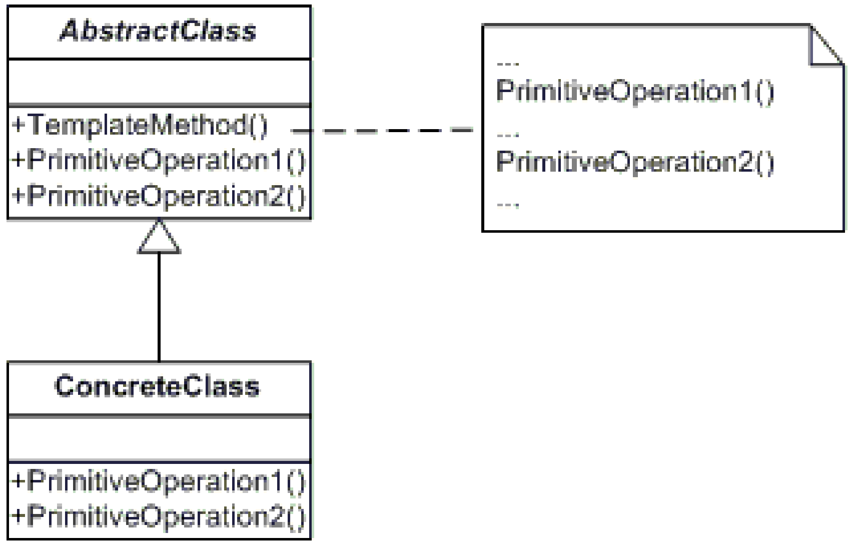
```
class Invoker
{
  private Command _command;

  public void SetCommand(Command command)
    => this._command = command;

  public void ExecuteCommand()
    => _command.Execute();
}
```

Template Method Pattern

- Define the **skeleton** of an algorithm in a method, leaving some implementation to its subclasses
- Allows the subclasses to **redefine** the implementation of some of the **parts** of the algorithm, but not its structure



The Abstract Class

```
abstract class AbstractClass
{
  public abstract void PrimitiveOperation1();
  public abstract void PrimitiveOperation2();

  public void TemplateMethod() {
    PrimitiveOperation1();
    PrimitiveOperation2();
    Console.WriteLine(""); }
}
```

A Concrete Class

```
class ConcreteClassA : AbstractClass
{
  public override void PrimitiveOperation1()
    => Console.WriteLine("ConcreteClassA.
       PrimitiveOperation1()");


  public override void PrimitiveOperation2()
    => Console.WriteLine("ConcreteClassA
       .PrimitiveOperation2()");
}
```

Summary

- Design Patterns
  - Provide solution to common problems
  - Add additional layers of abstraction
- Three main types of Design Patterns
  - Creational
  - Structural
  - Behavioral

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

SoftUni Diamond Partners

Software University

Educational Partners

Software University

CODE IT UP

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License