SoftServe

Empowering your Business
through Software Development

Design patterns

GoF for .Net Developers

Agenda

- History
- What is patterns?
- The most uses GoF patterns:
  - Singleton;
  - Factory Method;
  - Abstract Factory;
  - Builder;
  - Adapter;
  - Bridge;
  - Facade;
  - Composite;
  - Iterator.

SoftServe

Empowering your Business
through Software Development

History

History

Patterns originated as an architectural concept by Christopher Alexander (1977/79). In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming – specifically pattern languages – and presented their results at the OOPSLA conference that year. In the following years, Beck, Cunningham and others followed up on this work.

Design patterns gained popularity in computer science after the book Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994 by the so-called "Gang of Four" (Gamma et al.), which is frequently abbreviated as “GoF".

Although design patterns have been applied practically for a long time, formalization of the concept of design patterns languished for several years.

What is patterns?

What is patterns?

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Design pattern – a description of the interaction of objects and classes that are adapted to solve the general problem of designing in context.

What is patterns?

- Any pattern describes the problem that arises again and again in our work.
- In general, pattern consists of **four** main elements:
  - **Name**. Referring to it, we can immediately describe the problem, designing of, and its decisions and their consequences. Assigning names of patterns allows design at a higher level of abstraction.
  - **Problem**. A description of when to apply the pattern. Necessary to formulate the problem and its context.
  - **Solution**. Description of design elements, relations between them, the functions each item.
  - **The Results** – a consequence of the pattern. In describing the impact of design decisions are often not mentioned. You must choose between different options and evaluate the advantages and disadvantages of the pattern.

What is patterns?

- **Design Patterns** are divided into groups
- **Creational patterns** are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
  - Abstract Factory groups object factories that have a common theme.
  - Builder constructs complex objects by separating construction and representation.
  - Factory Method creates objects without specifying the exact class to create.
  - Prototype creates objects by cloning an existing object.
  - Singleton restricts object creation for a class to only one instance.
- **Behavioral patterns**. Most of these design patterns are specifically concerned with **communication** between objects.
  - Chain of responsibility delegates commands to a chain of processing objects.
  - Iterator accesses the elements of an object sequentially without exposing its underlying representation.
  - Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.

What is patterns?

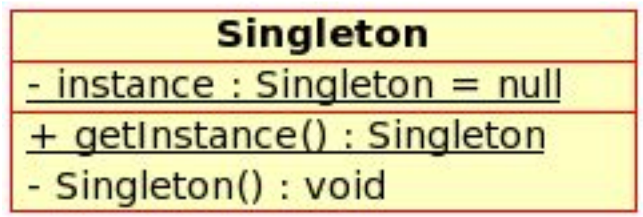
- **Structural patterns**. These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain **new functionality**.
  - Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
  - Bridge decouples an abstraction from its implementation so that the two can vary independently.
  - Composite composes zero-or-more similar objects so that they can be manipulated as one object.
  - Decorator dynamically adds/overrides behaviour in an existing method of an object.
  - Facade provides a simplified interface to a large body of code.
  - Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.
- Design patterns reside in the domain of modules and interconnections. At a higher level there are **architectural patterns** that are larger in scope, usually describing an overall pattern followed by an entire system.

SoftServe | Empowering your Business through Software Development

Singleton

Singleton

In software engineering, the singleton pattern is a design pattern that restricts the instantiation of a class to one object.



Sigleton: Example

SingletoneObject.cs

```
public class SingletoneObject
{
        private static SingletoneObject instance = new SingletoneObject();

        private SingletoneObject(){}

        public static SingletoneObject getInstance()
        {
                return instance;
        }

        public void showMessage()
        {
                Console.WriteLine("Hello Singletone!");
        }
}
```

Program.cs

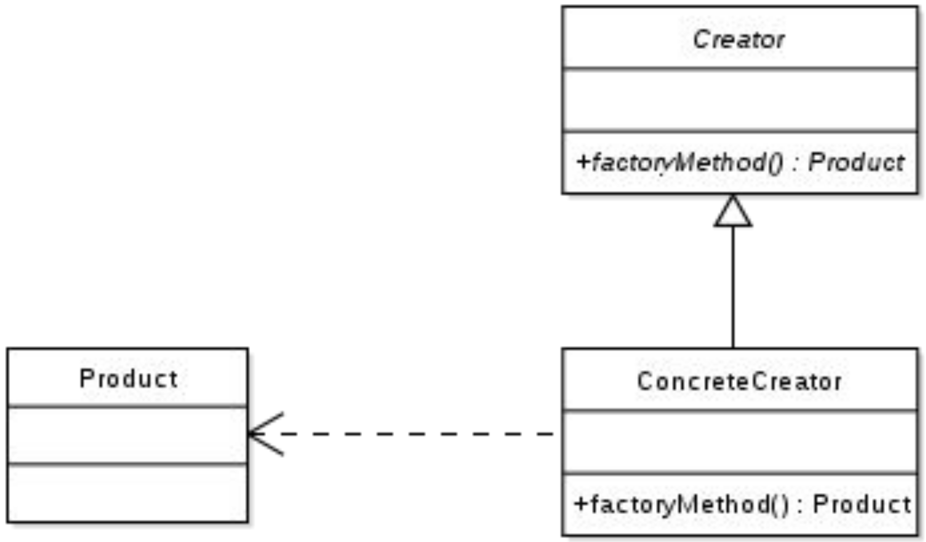
```
class MainClass
        {
                public static void Main (string[] args)
                {
                        SingletoneObject obj = SingletoneObject.getInstance ();

                        obj.showMessage ();
                }
        }
```

Factory Method

Factory method

Factory pattern is one of most used design pattern. In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



Factory Method: Example

We're going to create a *Mechanism* interface and concrete classes implementing the *Mechanism* interface. A factory class *MechanismFactory* is defined as a next step.

MainClass, our demo class will use *MechanismFactory* to get a *Mechanism* object. It will pass information (*BALLISTA / CATAPULT / TREBUCHET*) to *MechanismFactory* to get the type of object it needs.

Factory Method: Example

Mechahism.cs

```
public interface Mechanism
{
        void choice();
}
```

Ballista.cs

```
public class Ballista : Mechanism
{
        public Ballista ()
        {
        }

        public void choice ()
        {
                Console.WriteLine ("Choised Ballista");
        }
}
```

Catapult.cs

```
public class Catapult : Mechanism
{
        public Catapult ()
        {
        }

        public void choice ()
        {
                Console.WriteLine ("Choised Catapult");
        }
}
```

Factory Method: Example

MechanismFactory.cs

```
public class MechanismFactory
{
        public MechanismFactory ()
        {
        }

        public Mechanism getMechanism(String mechanismType)
        {
                if (mechanismType == null) {
                        return null;
                }

                if (mechanismType.ToUpper () == "BALLISTA") {
                        return new Ballista ();
                }  else if (mechanismType.ToUpper () == "CATAPULT") {
                        return new Catapult ();
                }  else if (mechanismType.ToUpper () == "TREBUCHET") {
                        return new Trebuchet ();
                }

                return null;
        }
}
```

Factory Method: Example

Trebuchet.cs

```
public class Trebuchet : Mechanism
{
        public Trebuchet ()
        {
        }

        public void choice ()
        {
                Console.WriteLine ("Choised Trebuchet");
        }
}
```

MainClass.cs

```
class MainClass
        {
                public static void Main (string[] args)
                {
                        MechanismFactory mechanismFactory = new MechanismFactory ();

                        Mechanism mechanism1 = mechanismFactory.getMechanism ("catapult");
                        mechanism1.choice ();

                        Mechanism mechanism2 = mechanismFactory.getMechanism ("ballista");
                        mechanism2.choice ();

                        Mechanism mechanism3 = mechanismFactory.getMechanism ("trebuchet");
                        mechanism3.choice ();
                }
        }
```

SoftServe

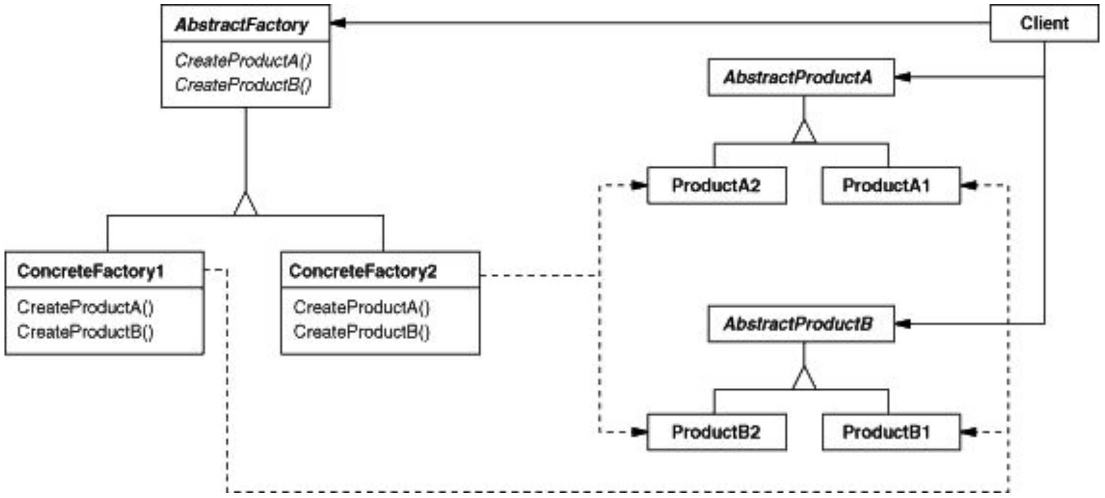
Empowering your Business
through Software Development

Abstract Factory

Abstract Factory

Abstract Factory patterns work around a super-factory which creates other factories.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.



Abstract Factory: Example

We are going to create a *Race* and *Army* interfaces and concrete classes implementing these interfaces. We create an abstract factory class *AbstractFactory* as next step. Factory classes *RaceFactory* and *ArmyFactory* are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.

MainClass, our demo class uses *FactoryProducer* to get a *AbstractFactory* object. It will pass information (*ORKS / ELVES / DWARFS* for *Race*) to *AbstractFactory* to get the type of object it needs. It also passes information (*WARIOR / ARCHER / CAVALARY* for *Army*) to *AbstractFactory* to get the type of object it needs.

Abstract Factory: Example

Race.cs

```
public interface Race
{
        void whoAmI();
}
```

Orks.cs

```
public class Orks : Race
{
        public Orks ()
        {
        }

        void Race.whoAmI()
        {
                Console.WriteLine ("I am Ork");
        }
}
```

Elves.cs

```
public class Elves : Race
{
        public Elves ()
        {
        }

        void Race.whoAmI()
        {
                Console.WriteLine ("I am Elve");
        }
}
```

Abstract Factory: Example

Dwarfs.cs

```
public class Dwarfs : Race
{
        public Dwarfs ()
        {
        }
        void Race.whoAmI()
        {
                Console.WriteLine ("I am Dwarf");
        }
}
```

Army.cs

```
public interface Army
{
        void setType();
}
```

Archer.cs

```
public class Archer : Army
{
        public Archer ()
        {
        }

        void Army.setType()
        {
                Console.WriteLine ("My type is Archer");
        }
}
```

Factory Method: Example

Warior.cs

```
public class Warior : Army
{
        public Warior ()
        {
        }

        void Army.setType()
        {
                Console.WriteLine ("My type is Warior");
        }
}
```

Cavalary.cs

```
public class Cavalary : Army
{
        public Cavalary ()
        {
        }

        void Army.setType()
        {
                Console.WriteLine ("My type is Cavalary");
        }
}
```

Abstract Factory: Example

AbstractFactory.cs

```
public abstract class AbstractFactory
{
        abstract public Race getRace(String race);
        abstract public Army getArmy(String army);
}
```

FactoryProducer.cs

```
public class FactoryProducer
{
        public static AbstractFactory getFactory(String choice)
        {
                if (choice.ToUpper() == "RACE") {
                        return new RaceFactory ();
                }  else if (choice.ToUpper() == "ARMY") {
                        return new ArmyFactory ();
                }
                return null;
        }
}
```

Factory Method: Example

ArmyFactory.cs

```
public class ArmyFactory : AbstractFactory
{
        override public Race getRace (string raceType)
        {
                return null;
        }

        override public Army getArmy (string armyType)
        {
                if (armyType == null) {
                        return null;
                }

                if (armyType.ToUpper () == "WARIOR") {
                        return new Warior ();
                }  else if (armyType.ToUpper () == "ARCHER") {
                        return new Archer ();
                }  else if (armyType.ToUpper () == "CAVALARY") {
                        return new Cavalary ();
                }

                return null;
        }
}
```

SoftServe Empowering your Business through Software Development

Factory Method: Example

RaceFactory.cs

```
public class RaceFactory : AbstractFactory
{
        override public Race getRace(String raceType)
        {
                if (raceType == null) {
                        return null;
                }

                if (raceType.ToUpper () == "ORKS") {
                        return new Orks ();
                }  else if (raceType.ToUpper () == "ELVES") {
                        return new Elves ();
                }  else if (raceType.ToUpper () == "DWARFS") {
                        return new Dwarfs ();
                }

                return null;
        }

        override public Army getArmy (string armyType)
        {
                return null;
        }
}
```

Factory Method: Example

MainClass.cs

```
class MainClass
{
        public static void Main (string[] args)
        {
                AbstractFactory raceFactory = FactoryProducer.getFactory ("RACE");

                Race race1 = raceFactory.getRace ("ORKS");
                race1.whoAmI ();

                Race race2 = raceFactory.getRace ("ELVES");
                race2.whoAmI ();

                Race race3 = raceFactory.getRace ("DWARFS");
                race3.whoAmI ();

                AbstractFactory armyFactory = FactoryProducer.getFactory ("ARMY");

                Army army1 = armyFactory.getArmy ("WARIOR");
                army1.setType();

                Army army2 = armyFactory.getArmy ("ARCHER");
                army2.setType();

                Army army3 = armyFactory.getArmy ("CAVALARY");
                army3.setType();
        }
}
```

SoftServe

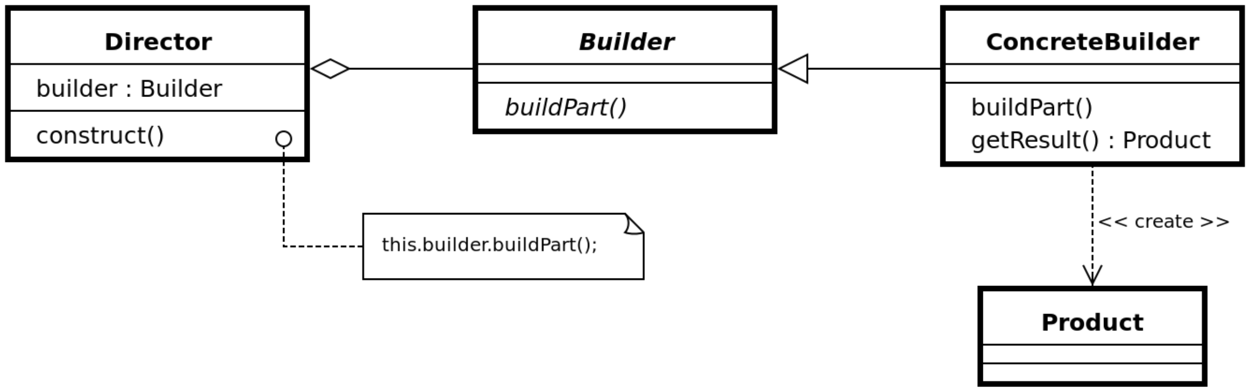
Empowering your Business through Software Development

Builder

Builder

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.



Builder: Example

We have considered a case of army stack forming where a typical army could be a hero and a platoon. Hero could be either a Dwarf or Elf and will be move by a horse. Platoon could be either a Wariors or Archers and will be move afoot.

We are going to create an Stack interface representing stack of army such as heros and platoons and concrete classes implementing the Stack interface and a Movement interface representing type of army moving and concrete classes implementing the Movement interface as hero would be move by a horse and platoon would be move afoot.

We then create a *Army* class having *List* of *Stack* and a *ArmyBuilder* to build different types of *Army* objects by combining *Stack*. *MainClass*, our demo class will use *ArmyBuilder* to build a *Army*.

Builder: Example

Stack.cs

```
public interface Stack
{
        String name ();
        Movement movement ();
        int dexterity ();
}
```

Heroes.cs

```
public abstract class Heroes : Stack
{
        public abstract String name();

        public Movement movement()
        {
                return new Horse ();
        }

        public abstract int dexterity ();
}
```

Platoon.cs

```
public abstract class Platoon : Stack
{
        public abstract String name();

        public Movement movement()
        {
                return new Afoot ();
        }

        public abstract int dexterity();
}
```

Builder: Example

ElfesHeroe.cs

```
public class ElfesHeroe : Heroes
{
        public override int dexterity()
        {
                return 7;
        }

        public override String name()
        {
                return "Elfes Heroe";
        }
}
```

DwarfHero.cs

```
public class DwarfHero : Heroes
{
        public override int dexterity()
        {
                return 4;
        }

        public override String name()
        {
                return "Darf Heroe";
        }
}
```

Builder: Example

Archers.cs

```
public class Archers : Platoon
{
        public override int dexterity()
        {
                return 4;
        }

        public override String name()
        {
                return "Archers Platton";
        }
}
```

Wariors.cs

```
public class Wariors : Platoon
{
        public override int dexterity()
        {
                return 2;
        }

        public override String name()
        {
                return "Wariors Platoon";
        }
}
```

Builder: Example

Movement.cs

```
public interface Movement
{
        String setMovement();
}
```

Afoot.cs

```
public class Afoot : Movement
{
        public String setMovement()
        {
                return "Afoot";
        }
}
```

Horse.cs

```
public class Horse : Movement
{
        public String setMovement()
        {
                return "Horse";
        }
}
```

Builder: Example

Army.cs

```
public class Army
{
        private List<Stack> stacks = new List<Stack> ();

        public void addStack(Stack stack)
        {
                stacks.Add (stack);
        }

        public int getDexterity()
        {
                int dexterity = 0;

                foreach (Stack stack in stacks) {
                        dexterity += stack.dexterity ();
                }

                return dexterity;
        }

        public void showStackItems()
        {
                foreach (Stack stack in stacks) {
                        Console.Write ("Stack name: " + stack.name ());
                        Console.Write (", Movement: " + stack.movement());
                        Console.WriteLine (", Dexterity: " + stack.dexterity());
                }
        }
}
```

Builder: Example

ArmyBuilder.cs

```
public class ArmyBuilder
{
        public Army prepareDwarfsWariors()
        {
                Army army = new Army ();
                army.addStack (new DwarfHero ());
                army.addStack (new Wariors ());
                return army;
        }

        public Army prepareElvesArchers()
        {
                Army army = new Army ();
                army.addStack (new ElfesHeroe ());
                army.addStack (new Archers ());
                return army;
        }
}
```

Builder: Example

MainClass.cs

```
class MainClass
{
        public static void Main (string[] args)
        {
                ArmyBuilder armyBuild = new ArmyBuilder ();

                Army dwarfs = armyBuild.prepareDwarfsWariors ();
                Console.WriteLine ("Dwarfs Army:");
                dwarfs.showStackItems ();
                Console.WriteLine ("Total dexterity: " + dwarfs.getDexterity ());

                Army elves = armyBuild.prepareElvesArchers ();
                Console.WriteLine ("\n\nElves Army:");
                elves.showStackItems ();
                Console.WriteLine ("Total dexterity: " + elves.getDexterity ());
        }
}
```

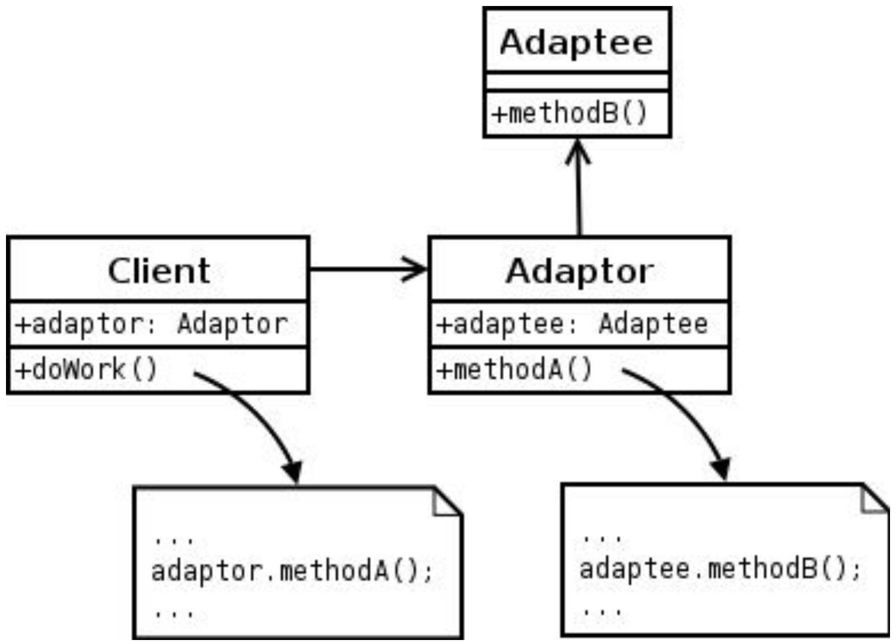
SoftServe

Empowering your Business through Software Development

Adapter

Adapter

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.



Adapter: Example

We have a NatureSpells interface and a concrete class Mage implementing the NatureSpells interface. Mage can cast Nature spells by default.

We are having another interface MoreSpells and concrete classes implementing the MoreSpells interface. These classes define the other type of spells.

We want to make Mage to cast other spells as well. To attain this, we have created an adapter class SpellsAdapter which implements the NatureSpells interface and uses MoreSpells objects to cast the required spell.

Mage uses the adapter class SpellsAdapter passing it the desired spell type without knowing the actual class which can cast the desired spell. MainClass, our demo class will use Mage class to play various spells.

Adapter: Example

NatureSpells.cs

```
public interface NatureSpells
{
        void cast(String typeSpell, String nameSpell);
}
```

Mage.cs

```
public class Mage : NatureSpells
{
        SpellsAdapter splellAdapter;

        public void cast(String typeSpell, String nameSpell)
        {
                if (typeSpell.ToUpper () == "NATURE") {
                        Console.WriteLine ("Use nature spell: " + nameSpell);
                }  else if ((typeSpell.ToUpper () == "FIRE") || (typeSpell.ToUpper () == "WATER")) {
                        splellAdapter = new SpellsAdapter (typeSpell);
                        splellAdapter.cast (typeSpell, nameSpell);
                }  else {
                        Console.WriteLine ("I can't use this spell type: " + typeSpell);
                }
        }
        public Mage ()
        {
        }
}
```

Adapter: Example

MoreSpells.cs

```
public interface MoreSpells
{
        void castFireSpell(String nameSpell);
        void castWaterSpell(String nameSpell);
}
```

FireSpells.cs

```
public class FireSpells : MoreSpells
{
        public void castFireSpell(String nameSpell)
        {
                Console.WriteLine ("Use fire spell: " + nameSpell);
        }

        public void castWaterSpell(String nameSpell) { }
}
```

WaterSpells.cs

```
public class WaterSpells : MoreSpells
{
        public void castFireSpell(String nameSpell) { }

        public void castWaterSpell(String nameSpell)
        {
                Console.WriteLine ("Use water spell: " + nameSpell);
        }
}
```

Adapter: Example

SpellsAdapter.cs

```
public class SpellsAdapter : NatureSpells
{
        MoreSpells moreSpellsForHeroe;

        public SpellsAdapter (String typeSpell)
        {
                if (typeSpell.ToUpper () == "FIRE") {
                        moreSpellsForHeroe = new FireSpells ();
                }  else if (typeSpell.ToUpper () == "WATER") {
                        moreSpellsForHeroe = new WaterSpells ();
                }
        }

        public void cast(String typeSpell, String nameSpell)
        {
                if (typeSpell.ToUpper () == "FIRE") {
                        moreSpellsForHeroe.castFireSpell (nameSpell);
                }  else if (typeSpell.ToUpper () == "WATER") {
                        moreSpellsForHeroe.castWaterSpell (nameSpell);
                }
        }
}
```

Adapter: Example

MainClass.cs

```
class MainClass
        {
                public static void Main (string[] args)
                {
                        Mage greatMage = new Mage ();

                        greatMage.cast ("Nature", "Stone Shield");
                        greatMage.cast ("Fire", "FireBall");
                        greatMage.cast ("Water", "Water Shield");
                        greatMage.cast ("Chaos", "Armagedon");
                }
        }
```

SoftServe

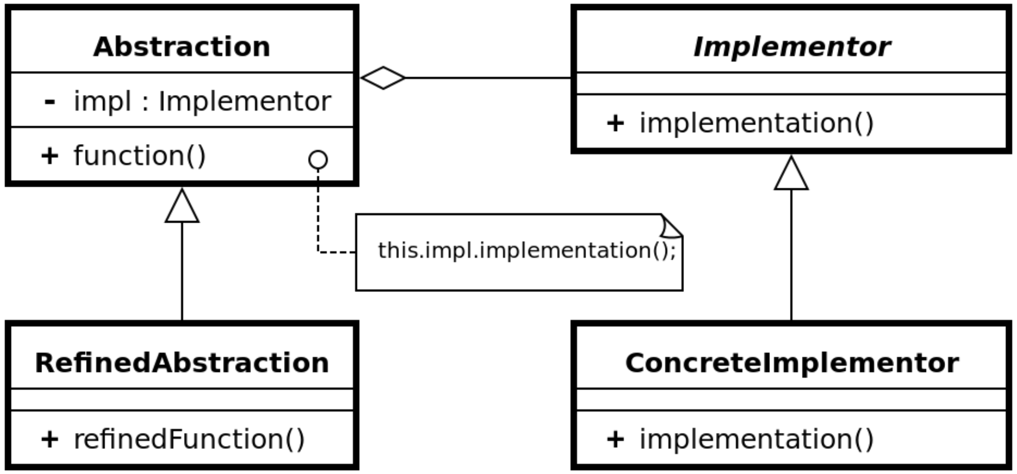
Empowering your Business
through Software Development

Bridge

Bridge

Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.



Bridge: Example

We have a PlatoonAPI interface which is acting as a bridge implementer and concrete classes EliteArcher, SharpShooter implementing the PlatoonAPI interface. Platoon is an abstract class and will use object of PlatoonAPI.

MainClass, our demo class will use Platoon class to draw different archer type.

PlatoonAPI.cs

```
public interface PlatoonAPI
{
        void abilityPlatoon(int dexterity, int accuracy, int wisdom);
}
```

Bridge: Example

MoreSpells.cs

```
public abstract class Platoon
{
        protected PlatoonAPI platoonAPI;
        public abstract void ability();
}
```

EliteArcher.cs

```
public class EliteArcher : PlatoonAPI
{
        public void abilityPlatoon(int dexterity, int accuracy, int wisdom)
        {
                Console.WriteLine ("Platoon type: 'Elite Archer';");
                Console.WriteLine ("Ability:");
                Console.WriteLine ("Dexterity: " + dexterity);
                Console.WriteLine ("Wisdom: " + wisdom);
                Console.WriteLine ("Accuracy: " + accuracy);
        }
}
```

SharpShooter.cs

```
public class SharpShooter : PlatoonAPI
{
        public void abilityPlatoon(int dexterity, int accuracy, int wisdom)
        {
                Console.WriteLine ("Platoon type: 'Sharpshooter';");
                Console.WriteLine ("Ability:");
                Console.WriteLine ("Dexterity: " + dexterity);
                Console.WriteLine ("Wisdom: " + wisdom);
                Console.WriteLine ("Accuracy: " + accuracy);
        }
}
```

Bridge: Example

Archer.cs

```
public class Archer : Platoon
{
        private int dexterity, accuracy, wisdom;

        public Archer (int dexterity, int accuracy, int wisdom, PlatoonAPI platoonAPI)
        {
                this.platoonAPI = platoonAPI;
                this.accuracy = accuracy;
                this.dexterity = dexterity;
                this.wisdom = wisdom;
        }

        public override void ability()
        {
                platoonAPI.abilityPlatoon (dexterity, accuracy, wisdom);
        }
}
```

MainClass.cs

```
class MainClass
{
        public static void Main (string[] args)
        {
                Platoon eliteArcher = new Archer (3, 2, 1, new EliteArcher ());
                Platoon sharpShooter = new Archer (5, 7, 2, new SharpShooter ());

                eliteArcher.ability ();
                sharpShooter.ability ();
        }
}
```

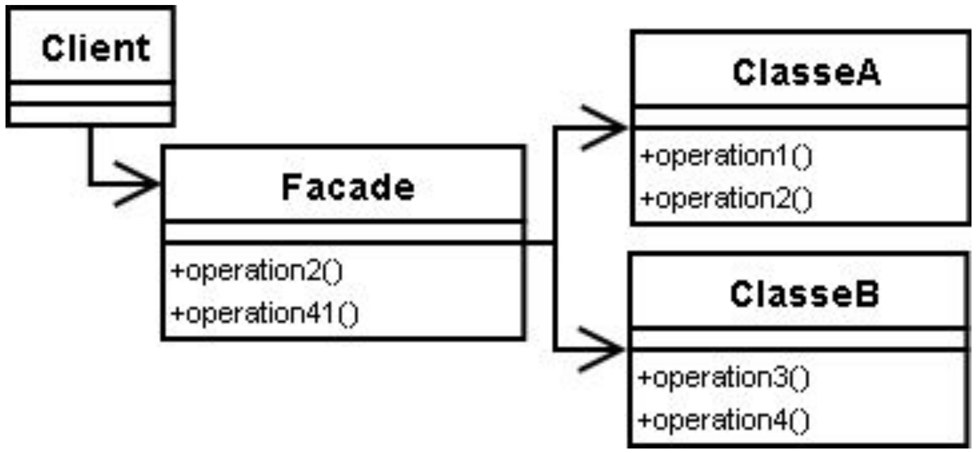
SoftServe

Empowering your Business through Software Development

Facade

Facade

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.



Facade: Example

We are going to create a Hero interface and concrete classes implementing the Hero interface. A facade class HeroeFacade is defined as a next step.

HeroeFacade class uses the concrete classes to delegate user calls to these classes. MainClass, our demo class, will use HeroeFacade class to show the results.

Hero.cs

```
public interface Hero
{
        void chooseHero ();
}
```

Facade: Example

Crusader.cs

```
public class Crusader : Hero
{
        public void chooseHero ()
        {
                Console.WriteLine ("Your choose is 'Crusader'");
        }
}
```

Barbarian.cs

```
public class Barbarian : Hero
{
        public void chooseHero ()
        {
                Console.WriteLine ("Your choose is 'Barbarian'");
        }
}
```

Wizard.cs

```
public class Wizard : Hero
{
        public void chooseHero ()
        {
                Console.WriteLine ("Your choose is 'Wizard'");
        }
}
```

Facade: Example

HeroeFacade.cs

```
public class HeroeFacade
{
        private Hero barbarian;
        private Hero crusader;
        private Hero wizard;

        public HeroeFacade ()
        {
                barbarian = new Barbarian ();
                crusader = new Crusader ();
                wizard = new Wizard ();
        }

        public void chooseBarbarian ()
        {
                barbarian.chooseHero ();
        }

        public void chooseCrusader ()
        {
                crusader.chooseHero ();
        }

        public void chooseWizard ()
        {
                wizard.chooseHero ();
        }
}
```

Facade: Example

MainClass.cs

```
class MainClass
{
        public static void Main (string[] args)
        {
                HeroeFacade facade = new HeroeFacade ();

                facade.chooseBarbarian ();
                facade.chooseCrusader ();
                facade.chooseWizard ();
        }
}
```

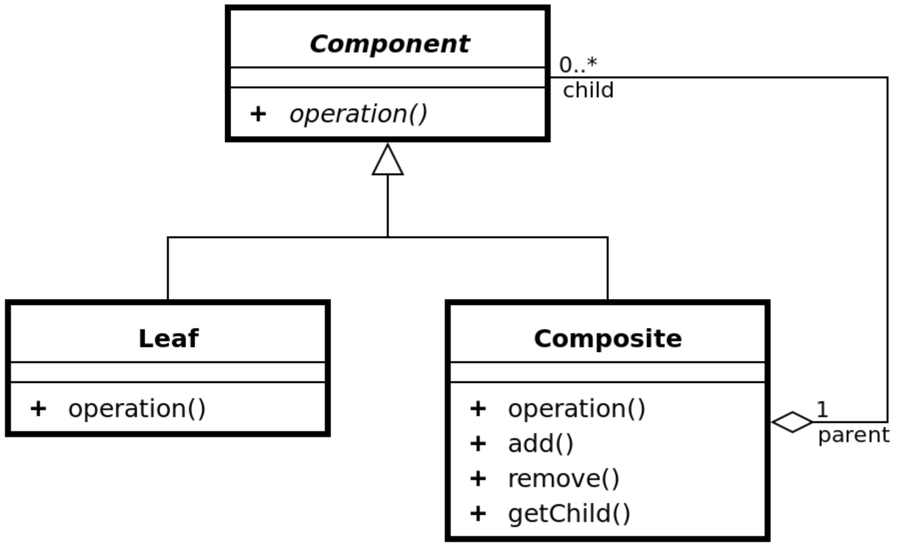
SoftServe | Empowering your Business through Software Development

Composite

Composite

Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.



Composite: Example

We have a class *Heroes* which acts as composite pattern actor class. *MainClass*, our demo class will use *Heroes* class to add stack of army level hierarchy and print all heroes.

Composite: Example

Heroes.cs

```
public class Heroes
{
        private String name;
        private String race;
        private String stack;
        private List<Heroes> subordinates;

        public Heroes (String name, String race, String stack)
        {
                this.name = name;
                this.race = race;
                this.stack = stack;
                subordinates = new List<Heroes> ();
        }

        public void add (Heroes hero)
        {
                subordinates.Add (hero);
        }

        public void remove (Heroes hero)
        {
                subordinates.Remove (hero);
        }

        public List<Heroes> getSubordinates ()
        {
                return subordinates;
        }

        public override string ToString()
        {
                return ("Hero: [ Name : " + name + ", race : " + race + ", stack : " + stack + " ]");
        }
}
```

Composite: Example

MainClass.cs

```
class MainClass
{
        public static void Main (string[] args)
        {
                Heroes armyCommander = new Heroes ("Jack Hornson", "Human", "Army commander");

                Heroes wariours = new Heroes ("Crag Hack", "Human", "Wariours commander");

                Heroes archers = new Heroes ("Jelu Devil's Bane", "Elve", "Archers commander");

                Heroes wariour1 = new Heroes ("Catherine Ironfist", "Human", "Wariour");
                Heroes wariour2 = new Heroes ("Gavin Magnus", "Human", "Wariour");

                Heroes archer1 = new Heroes ("Aeris AvLee", "Elve", "Archer");
                Heroes archer2 = new Heroes ("Erutan Revol", "Elve", "Archer");

                armyCommander.add (wariours);
                armyCommander.add (archers);

                wariours.add (wariour1);
                wariours.add (wariour2);

                archers.add (archer1);
                archers.add (archer2);
```

…

MainClass.cs

```
                Console.WriteLine (armyCommander);

                foreach (Heroes armyHead in armyCommander.getSubordinates())
                {
                        Console.WriteLine (armyHead);

                        foreach (Heroes army in armyHead.getSubordinates())
                        {
                                Console.WriteLine (army);
                        }
                }
        }
}
```

SoftServe

Empowering your Business
through Software Development

Iterator

Iterator

Iterator pattern is very commonly used design pattern in .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Iterator: Example

We're going to create a abstract class Iterator which narrates navigation method and a class HeroesAggregate, implementing abstract class Aggregate, which retruns the iterator HeroesIterator.

MainClass, our demo class will use HeroesAggregate, a concrete class implementation to print a Names of heroes stored as a collection in HeroesAggregate.

Iterator.cs

```
abstract class Iterator
{
        public abstract object First();
        public abstract object Next();
        public abstract bool IsDone();
        public abstract object CurrentItem();
}
```

Iterator: Example

Aggregate.cs

```
abstract class Aggregate
{
        public abstract Iterator CreateIterator();
}
```

HeroesAggregate.cs

```
class HeroesAggregate : Aggregate
{
        private ArrayList items = new ArrayList();

        public override Iterator CreateIterator()
        {
                return new HeroesIterator(this);
        }

        public int Count
        {
                get{  return items.Count; }
        }

        public object this[int index]
        {
                get{  return items[index]; }
                set{  items.Insert(index, value); }
        }
}
```

Iterator: Example

Archer.cs

```
class HeroesIterator : Iterator
{
        private HeroesAggregate aggregate;
        private int current = 0;

        public HeroesIterator(HeroesAggregate aggregate)
        {
                this.aggregate = aggregate;
        }

        public override object First()
        {
                return aggregate[0];
        }

        public override object Next()
        {
                object ret = null;
                if (current < aggregate.Count - 1) {
                        ret = aggregate[++current];
                }
                return ret;
        }

        public override object CurrentItem()
        {
                return aggregate[current];
        }

        public override bool IsDone()
        {
                return current >= aggregate.Count ? true : false ;
        }
}
```

Iterator: Example

MainClass.cs

```
class MainClass
{
        public static void Main (string[] args)
        {
                HeroesAggregate heroesStack = new HeroesAggregate();
                heroesStack[0] = "Jelu Devil's Bane";
                heroesStack[1] = "Catherine Ironfist";
                heroesStack[2] = "Gavin Magnus";
                heroesStack[3] = "Aeris AvLee";

                HeroesIterator itemStack = new HeroesIterator(heroesStack);

                Console.WriteLine("Iterating over collection of heroes:");

                object item = itemStack.First();
                while (item != null)
                {
                        Console.WriteLine(item);
                        item = itemStack.Next();
                }
        }
}
```

SoftServe | *Empowering your Business through Software Development*

Thank you!

US OFFICES	*EUROPE OFFICES*		
Austin, TX *Fort Myers, FL* *Lehi, UT* *Newport Beach, CA* *Waltham, MA*	*Bulgaria* *Germany* *Netherlands* *Poland* *Russia*	*Sweden* *Ukraine* *United Kingdom*	www.softserveinc.com