

## Structural Patterns :

### Adapter Design Pattern :

Les Classes et objets participants au patron :

*Target :*

- Définit un domaine spécifique dans une interface utilisée par le Client.

*Adapter :*

- Adapte l'interface Adaptee l'interface Target

*Adaptee :*

- A le rôle d'interface nécessitant l'adaptation

*Client:*

- Implémente l'interface Target

## Code Csharp généré automatiquement

```

//Adaptee
public class Adaptee {
    public void SpecificRequest() {
    }
    Console.WriteLine('Called SpecificRequest');
}
//Adapter
public class Adapter : Target {
    adaptee = new Adaptee();
    public override void Request() {
        //Possibly do some other work
        //and then call SpecificRequest
    }
    adaptee.SpecificRequest();
}
private Adaptee adaptee;
}

```

## Code PADL

```

//*****Adapter Class*****
IClass adap = Factory.getInstance().createClass("Adapter");
adap.addActor(adap);
adap.addInheritedActor(target);
//*****Adaptee Class*****
IClass Adaptee = Factory.getInstance().createClass("Adaptee");
adap.addActor(Adaptee);
IField adaptee = Factory.getInstance().createField("adaptee", "Adaptee", 2);
adap.addActor(adaptee);
adaptee.setPublic(false);
adaptee.setPrivate(true);
IInstantiation inst = StatementFactory.getStateInstance().createInstantiation( adaptee, Adaptee);
adap.addActor(inst);
//Request Method
ICSharpMethod request1 = CSharpFactory.getCsharpInstance().createNewMethod("Request");
adap.addActor(request1);
request1.setOverride(true);
request1.setPublic(true);
request1.setComment(" //Possibly do some other work \n //and then call SpecificRequest ");
//SpecificRequest Method
IMethod SpecificRequest = Factory.getInstance().createMethod("SpecificRequest");
Adaptee.addActor(SpecificRequest);
IOutput out = StatementFactory.getStateInstance().createOutputPut( "Called SpecificRequest");
SpecificRequest.addActor(out);
IMethodInvocation invoc = Factory.getInstance().createMethodInvocation(2, 1, 1, adap);
invoc.addCallingField(adaptee);
invoc.setCalledMethod(SpecificRequest);
request1.addActor(invoc);

```

```

//Target
public abstract class Target {
    public abstract void Request();
} Le Code C# Correpondant Adapter Design Motif

ICodeLevelModel adapter =
Factory.getInstance().createCodeLevelModel("Adapter");
try {
    //*****Target Class*****
    IClass target = Factory.getInstance().createClass("Target");
    adapter.addAction(target);
    target.setAbstract(true);
    //Request Method
    ISharpMethod request2 =
    CSharpFactory.getSharpInstance().createNewMethod("Request");
    target.addAction(request2);
    request2.setAbstract(true);
    request2.setPublic(true);
    request2.setComment("//Called Target Request");
}

```

## Bridge Design Pattern :

Les Classes et objets participants au patron :

*Abstraction* :

- A le rôle d'interface définissant des méthodes abstraite.
- Maintient une référence à un objet de type *Implementor*.

*RefinedAbstraction* :

- Étend l'interface définie par *Abstraction*

*Implementor* :

- A le rôle d'interface pour l'implémentation des classes. Cette interface ne doit pas forcément correspondre à l'interface *Abstraction*. en effet les deux interfaces peuvent être très différentes.

*ConcreteImplementor* :

- Implémente l'interface *Implementor*

Code Csharp généré automatiquement	Code PADL
<pre> //Abstraction public class Abstraction {     public Implementor Implementor     {         set {implementor= value; }     }     public virtual void Operation() {         implementor.Operation();     }     protected Implementor implementor; } //Implementor public abstract class Implementor {     public abstract void Operation(); } </pre>	<pre> //*****Abstraction Class***** IClass abstraction = Factory.getInstance().createClass("Abstraction"); bridge.addActor(abstraction);  IField implementor = Factory.getInstance().createField("implementor", "Implementor",2); abstraction.addActor(implementor); implementor.setProtected(true); implementor.setProtected(true);  IPropriety Implementor = CSharpFactory.getCSharpInstance().createPropriety( "Implementor","Implementor",implementor); abstraction.addActor(Implementor); Implementor.setPublic(true);  ISetterProprety set = CSharpFactory.getCSharpInstance().createSetAccesneur(); Implementor.setterProprety(set); IAffectation affecte = StatementFactory.getStateInstance().createAffectation(implementor,"value");  //Operation Method ICSharpMethod Operation = CSharpFactory.getCSharpInstance().createNewMethod("Operation"); abstraction.addActor(Operation); Operation.setVirtual(true);  IMethodInvocation invoc = Factory.getInstance().createMethodInvocation(2,1,1,abstraction); Operation.addActor(invoc); invoc.addCallingField(implementor); invoc.setCalledMethod(Operation);  //*****Implementor Class***** IClass Implementor1 = Factory.getInstance().createClass("Implementor"); bridge.addActor(Implementor1); Implementor1.setAbstract(true);  //Operation Method IMethod operation = Factory.getInstance().createMethod("Operation"); Implementor1.addActor(operation); operation.setAbstract(true); </pre>

```

//ConcreteImplementorA
public class ConcreteImplementorA : Implementor {
public override void Operation() {
//Dfinir une implementation concrte de la Methode Operation
}
}
//ConcreteImplementorB
public class ConcreteImplementorB : Implementor {
public override void Operation() {
//Dfinir une implementation concrte de la Methode Operation
}
}
//RefinedAbstraction
public class RefinedAbstraction : Abstraction {
public override void Operation() {
}
}
} Le Code C# Correspondant Bridge Design Motif
implementor.Operation();
}

```

```

//*****RefinedAbstraction Class*****
IClass RefinedAbstraction = Factory.GetInstance().createClass("RefinedAbstraction");
bridge.addActor(RefinedAbstraction);
RefinedAbstraction.addInheritedActor(abstraction);

MethodInvocation invoc1 = Factory.GetInstance().createMethodInvocation(2,1,1,abstraction);
invoc1.addCallingField(Implementor);
invoc1.setCalledMethod(Operation);

//Operation Method
ICSharpMethod operation2 = CSharpFactory.getCSharpInstance().createNewMethod("Operation");
RefinedAbstraction.addActor(operation2);
operation2.setOverride(true);
operation2.addActor(invoc);

//*****ConcreteImplementorA Class*****
IClass ConcreteImplementorA = Factory.GetInstance().createClass("ConcreteImplementorA");
bridge.addActor(ConcreteImplementorA);
ConcreteImplementorA.addInheritedActor(Implementor1);

//Operation Method
ICSharpMethod operation2 = CSharpFactory.getCSharpInstance().createNewMethod("Operation");
ConcreteImplementorA.addActor(operation2);
operation2.setOverride(true);
operation2.setComment( " //Dfinir une implementation concrte de la Methode Operation");

//*****ConcreteImplementorB Class*****
IClass ConcreteImplementorB = Factory.GetInstance().createClass("ConcreteImplementorB");
bridge.addActor(ConcreteImplementorB);
ConcreteImplementorB.addInheritedActor(Implementor1);

//Operation Method
ICSharpMethod operation3 = CSharpFactory.getCSharpInstance().createNewMethod("Operation");
ConcreteImplementorB.addActor(operation3);
operation3.setOverride(true);
operation3.setComment(
" //Dfinir une implementation concrte de la Methode Operation");

```

## Composite Design Pattern :

Les Classes et objets participants au patron :

*Component :*

- A le rôle d'interface pour la composition des objets.
- Définit un comportement par défaut pour une interface commune à toutes les classes.
- Définit une interface pour gérer et accéder toutes les composantes enfants.

*Leaf :*

- Représente les objets feuilles dans une composition. les feuilles ne sont pas des composants enfants.
- définit le comportement des objets primitifs dans la composition.

*Composite :*

- définit un comportement pour les composants ayant des enfants
- stocke les composant enfants
- implmente la gestions des enfants de l'interface Component

*Client :*

- manipule les objets de la composition à travers l'interface Composite

## Code Csharp généré automatiquement

```

//Component
public abstract class Component {
public abstract void Add(Component c);
public Component(string name) {
    this.name = name;
}
public abstract void Remove(Component c);
public abstract void getChild(int index);
protected string name;
}

```

## Code PADL

```

//*****Component Class*****
IClass component = Factory.GetInstance().createClass("Component");
component.setAbstract(true);
Composite.addActor(component);
IField name = Factory.GetInstance().createField("name", "string", 2);
component.addActor(name);
name.setProtected(true);
IConstructor Component = Factory.GetInstance().createConstructor("Component");
component.addActor(Component);
IParameter name1 = Factory.GetInstance().createParameter("name", "string");
Component.addActor(name1);
//*****Add Method*****
IMethod add = Factory.GetInstance().createMethod("Add");
component.addActor(add);
add.setAbstract(true);
IParameter c = Factory.GetInstance().createParameter("c", "Component");
add.addActor(c);
//*****Remove Method*****
IMethod remove = Factory.GetInstance().createMethod("Remove");
component.addActor(remove);
remove.setAbstract(true);
IParameter C = Factory.GetInstance().createParameter("c", "Component");
remove.addActor(C);
//*****getChild Method*****
IMethod getChild = Factory.GetInstance().createMethod("getChild");
component.addActor(getChild);
getChild.setAbstract(true);
IParameter index = Factory.GetInstance().createParameter("index", "int");
IAffectation affecter = StatementFactory.getStateInstance().createAffectation(
name, "name");
Component.addActor(affecter);

```

```

//Composite
public class Composite : Component {
    private ArrayList children = new ArrayList();
    public override void Add(Component component) {
        children.Add(component);
    }
    public override void Remove(Component component) {
        children.Remove(component);
    }
    public boolean componentInChildren() {
    }
    public override void getChild(int index) {
        for (this.componentInChildren)
        {
            // Recursively getChild nodes
            component.getChild(index + 2);
        }
    }
}

//***** Composite Class*****
IClass composite = Factory.getInstance().createClass("Composite");
Composite.addActor(composite);
composite.addInheritedActor(component);
composite.setComment(" private ArrayList children = new ArrayList();");
//*****Add Method*****
ICSharpMethod Add = CSharpFactory.getCsharpInstance().createNewMethod("Add");
composite.addActor(Add);
Add.setOverride(true);
IParameter compo = Factory.getInstance().createParameter("component", "Component");
Add.addActor(compo);
Add.setComment(" children.Add(component);");
//*****Remove Method*****
ICSharpMethod Remove = CSharpFactory.getCsharpInstance().createNewMethod("Remove");
Remove.setOverride(true);
composite.addActor(Remove);
Remove.addActor(compo);
Remove.setComment("children.Remove(component);");
//*****getChild Method*****
ICSharpMethod getChild = CSharpFactory.getCsharpInstance().createNewMethod("getChild");
composite.addActor(getChild);
getChild.setOverride(true);
getChild.addActor(index);
//*****componentInChildren Method*****
IMethod ChildInChildren = Factory.getInstance().createMethod("componentInChildren");
ChildInChildren.setReturnType("boolean");
composite.addActor(ChildInChildren);
IEnumerator it = StatementFactory.getStateInstance().createIteratorS(ChildInChildren);
it.setComment(" Recursively getChild nodes \n component.getChild(index + 2);");
getChild.addActor(it);

```

```

//leaf
public class leaf : Component {
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 getChild(int index) {
// Recursively display child nodes
}
} Le Code C# Correpndant Composite Design Motif

//***** leaf Class*****
IClass leaf = Factory.GetInstance().createClass("leaf");
Composite.addActor(leaf);
leaf.addInheritedActor(component);
//Add Method
ICSharpMethod Add1 =
CSharpFactory.getOsharpInstance().createNewMethod("Add");
Add1.setOverride(true);
leaf.addActor(Add1);
Add1.addActor(c);
IOutPut out =
StatementFactory.getStateInstance().createOutPut(
"Cannot add to a leaf");
Add1.addActor(out);
//Remove Method
ICSharpMethod remove1 =
CSharpFactory.getOsharpInstance().createNewMethod("Remove");
leaf.addActor(remove1);
remove1.setOverride(true);
remove1.addActor(c);
IOutPut out1 =
StatementFactory.getStateInstance().createOutPut(
"Cannot remove from a leaf");
remove1.addActor(out1);
//getChild Method
ICSharpMethod getChild1 =
CSharpFactory.getOsharpInstance().createNewMethod("getChild");
leaf.addActor(getChild1);
getChild1.setOverride(true);
getChild1.addActor(index);
getChild1.setComment(" // Recursively display child nodes ");

```

## **Proxy Design Pattern :**

Les Classes et objets participants au patron :

*Proxy :*

- Maintient une référence d'objet permettant au Proxy d'accéder l'objet Realsubject.
- Contrôle l'accès l'objet RealSubject et peut être responsable de le créer ou le supprimer.

*Subject :*

- définit une interface commune á RealSubject et Proxy

*RealSubject :*

- définit le RealSubject que représente le Proxy

Code Csharp généré automatiquement	Code PADL
<pre> //Proxy public class Proxy : Subject { public override void Request() {     if(this.realObjectNull)     {         realObject = new RealSubject();     }     realObject.Request(); } } public RealSubject realObject; public boolean realObjectNull() { } } </pre>	<pre> //*****Proxy Class***** IClass proxy = Factory.getInstance().createClass("Proxy"); Proxy.addActor(proxy); proxy.addInheritedActor(Subject); //realObjectNull Method IMethod realObject = Factory.getInstance().createMethod("realObjectNull"); proxy.addActor(realObject); realObject.setReturnType("boolean"); //Request Method ICSharpMethod Request1 = CSharpFactory.getOsharpInstance().createNewMethod("Request"); proxy.addActor(Request1); Request1.setOverride(true); IBlock block = StatementFactory.getStateInstance().createBlock(); Request1.addActor(block); IConditional structure = StatementFactory.getStateInstance().createConditioneIS(realObject); block.addActor(structure); structure.SetifBlock(block); IField realObject1 = Factory.getInstance().createField( "realObject", "RealSubject", 2); proxy.addActor(realObject1); IInstanciation inst = StatementFactory.getStateInstance().createInstanciation( realObject1, RealSubject); structure.addActor(inst); IMethodInvocation invoc = Factory.getInstance().createMethodInvocation(2, 1, 1, proxy); Request1.addActor(invoc); invoc.addCallingField(realObject1); invoc.setCalledMethod(Request1); </pre>

```

//RealSubject
public class RealSubject : Subject {
    public override void Request() {
        Console.WriteLine('Called RealSubject.Request()');
    }
}
//Subject
public abstract class Subject {
    public abstract void Request();
}
} Le Code C# Correspondant a Proxy Design Motif

```

```

//*****Subject Class*****
IClass Subject = Factory.GetInstance().createClass("Subject");
Proxy.addActor(Subject); Subject.setAbstract(true);

//Request Method
IMethod request = Factory.GetInstance().createMethod("Request");
Subject.addActor(request);
request.setAbstract(true);

//*****RealSubject Class*****
IClass RealSubject =
Factory.GetInstance().createClass("RealSubject");
Proxy.addActor(RealSubject); RealSubject.addInheritedActor(Subject);

ICSharpMethod Request =
CSharpFactory.getSharpInstance().createNewMethod("Request");
RealSubject.addActor(Request); Request.setOverride(true); IOutPut
out = StatementFactory.getStateInstance().createOutPut( "Called
RealSubject.Request()"); Request.addActor(out);

```