

Behavioral Patterns :

Command Design Pattern :

Les Classes et objets participants au patron :

Command :

- A le rôle d'interface pour exécuter une opération

ConcreteCommand :

- définit un lien entre l'objet Receiver et une action
- Implémente la méthode Execute en invoquant l'opération correspondante sur le Receiver.

Invoker :

- Demande à la command de procéder à la requête

Receiver :

- Effectue les opérations liées à l'exécution de la requête

Client :

- crée un objet ConcreteCommand et fixe son Receiver

Code Csharp généré automatiquement

```

//Command
public abstract class Command {
    public Command(Receiver receiver) {
        this.receiver = receiver;
    }
    public abstract void Execute();
}
protected Receiver receiver;
//ConcreteCommand
public class ConcreteCommand : Command {
    public ConcreteCommand(Receiver receiver) {
    }
    public override void Execute() {
        receiver.Action();
    }
}

```

Code PADL

```

//*****Command Class*****
IClass command = Factory.GetInstance().createClass("Command");
Command.addActor(command);
command.setAbstract(true);
IField receiver = Factory.GetInstance().createField("receiver", "Receiver", 2);
command.addActor(receiver);
receiver.setProtected(true);
//Command Constructor
IConstructor command1 = Factory.GetInstance().createConstructor("Command");
command.addActor(command1);
IParameter receiver1 = Factory.GetInstance().createParameter("receiver", "Receiver");
command.addActor(receiver1);
IAffection affecte = StatementFactory.getStateInstance().createAffection(receiver, "receiver");
command1.addActor(affecte);
//Execute Method
IMethod execute = Factory.GetInstance().createMethod("Execute");
command.addActor(execute);
execute.setAbstract(true);
//*****ConcreteCommand Class*****
IClass ConcreteCommand = Factory.GetInstance().createClass("ConcreteCommand");
Command.addActor(ConcreteCommand);
ConcreteCommand.addInheritedActor(command);
//ConcreteCommand Constructor
IConstructor concreteC = Factory.GetInstance().createConstructor("ConcreteCommand");
ConcreteCommand.addActor(concreteC);
concreteC.addActor(receiver1);
//Execute Method
ICSharpMethod Execute = CSharpFactory.getSharpInstance().createNewMethod("Execute");
ConcreteCommand.addActor(Execute);
Execute.setOverride(true);
IMethodInvocation invoc = Factory.GetInstance().createMethodInvocation(2,1,1,ConcreteCommand);
invoc.addCallingField(receiver);
invoc.setCalledMethod(action);
Execute.addActor(invoc);

```

```

//Invoker
public class Invoker {
    public void ExecuteCommand() {
        commamd.Execute();
    }
    public void SetCommand(Command command) {
        this.command = command;
    }
    private Command command;
}
//Receiver
public class Receiver {
    public void Action() {
        //Called Receiver.Action()
    }
}
} Le Code C# Correspondant Command Design Motif

```

```

//*****Invoker Class*****
IClass invoker = Factory.getInstance().createClass("Invoker");
Command.addActor(invoker);

IField commande = Factory.getInstance().createField("command", "Command", 2);
invoker.addActor(commande);
commande.setPrivate(true);

//SetCommand Method
IMethod SetCommand = Factory.getInstance().createMethod("SetCommand");
invoker.addActor(SetCommand);

IParameter commandp = Factory.getInstance().createParameter("command", "Command");
SetCommand.addActor(commandp);

IAffectation affectel =
StatementFactory.getStateInstance().createAffectation(commande, "command");
SetCommand.addActor(affectel);

//ExecuteCommand Method
IMethod ExecuteCommand = Factory.getInstance().createMethod("ExecuteCommand");
invoker.addActor(ExecuteCommand);

IMethodInvocation invoce2 = Factory.getInstance().createMethodInvocation(2, 1, 1, invoker);
invoce2.addCallingField(commande);
invoce2.setCalledMethod(Execute);
ExecuteCommand.addActor(invoce2);

//*****Receiver Class*****
IClass Receiver = Factory.getInstance().createClass("Receiver");
Command.addActor(Receiver);

//Action Method
IMethod action = Factory.getInstance().createMethod("Action");
Receiver.addActor(action);
action.setComment("//Called Receiver.Action()");

```

Memento Design Pattern :

Les Classes et objets participants au patron :

Memento :

- enregistre l'état interne de l'objet Originator. Le Memento peut stocker plus d'un état interne de l'originator.

Originator :

- crée un Memento contenant une image de son état interne
- utilise le Memento pour restaurer son état interne

Caretaker :

- Est responsable de garder le Memento
- Ne modifie pas le contenu du Memento

Code Csharp généré automatiquement

Code PADL

```

//Caretaker
public class Caretaker {
    public Memento Memento
    {
        get { return memento; }
    }
    private Memento memento;
}
//Memento
public class Memento {
    public string GetState() {
        return state;
    }
    public Memento(string state) {
        this.state = state;
    }
    private string state;
}

```

```

//*****Caretaker Class*****
IClass caretaker = Factory.GetInstance().createClass("Caretaker");
Memento.addActor(caretaker);

IField memen = Factory.GetInstance().createField("memento", "Memento", 2);
memen.setPrivate(true);
caretaker.addActor(memen);

IPriority MementoP =
CSharpFactory.getCSharpInstance().createProperty( "Memento", "Memento", memen);
caretaker.addActor(MementoP);

IGetterProperty getC = CSharpFactory.getCSharpInstance().createGetAccesreur();
MementoP.setGetterProperty(getC);
//*****Memento Class*****
IClass memento = Factory.GetInstance().createClass("Memento");
Memento.addActor(memento);
IField statem = Factory.GetInstance().createField("state", "String", 2);
memento.addActor(statem);
statem.setPrivate(true);

//Memento Constructor
IConstructor mementoC = Factory.GetInstance().createConstructor("Memento");
memento.addActor(mementoC);
IParameter stateC = Factory.GetInstance().createParameter("state", "String");
mementoC.addActor(stateC);
IAffectation affectem = StatementFactory.getStateInstance().createAffectation(statem, "state");
mementoC.addActor(affectem);

//GetState Method
ICSharpMethod Statem1 = CSharpFactory.getCSharpInstance().createNewMethod("getState");
Statem1.setReturnType("string");
Statem1.setReturnValue("state");
memento.addActor(Statem1);

```

```

//Originator
public class Originator {
    public Memento CreateMemento() {
        return (new Memento(state));
    }
    public void SetMemento(Memento memento) {
        Console.WriteLine('Restoring state:');
        this.state = memento.GetState();
    }
    private string state;
    public string State
    {
        get { return state; }
        set {state= value; }
    }
} Le Code C# Correspondant Memento Design Motif

//*****Originator Class*****
IClass originator = Factory.GetInstance().createClass("Originator");
Memento.addActor(originator);

IField state = Factory.GetInstance().createField("state", "String", 2);
originator.addActor(state);
state.setPrivate(true);

IPropriety State =
CSharpFactory.getCSharpInstance().createPropriety( "State", "string", state);
originator.addActor(State);
IGetterPropriety get =CSharpFactory.getCSharpInstance().createGetAccesneur();
State.setGetterPropriety(get);
ISetterPropriety set = CSharpFactory.getCSharpInstance().createSetAccesneur();
State.setSetterPropriety(set);

//CreateMemento Method
ICSharpMethod createMemento = CSharpFactory.getCSharpInstance().createNewMethod("CreateMemento");
originator.addActor(createMemento);
createMemento.setReturnType("Memento");
createMemento.SetReturnValue("new Memento(state)");

//SetMemento Method
IMethod SetMemento = Factory.GetInstance().createMethod("SetMemento");
originator.addActor(SetMemento);

IParameter memento = Factory.GetInstance().createParameter("memento", "Memento");
SetMemento.addActor(memento);

IOutPut out = StatementFactory.getStateInstance().createOutPut("Restoring state:");
SetMemento.addActor(out);

IAffectation affecte =
StatementFactory.getStateInstance().createAffectation(state,
"memento.GetState()");
SetMemento.addActor(affecte);

```

Observer Design Pattern :

Les Classes et objets participants au patron :

Subject :

- Représente l'objet observé, le Sujet reconnaît tous ses observateurs et peut être observé par plus d'un observateur.
- fournit une interface pour attacher et détacher l'objet Observer
- fournit une interface pour attacher et détacher l'objet Observer

ConcreteSubject :

- Stocke l'état l'intérêt du ConcreteObserver
- Notifie ses observateurs dès que son état change

Observer :

- A le rôle d'interface pour une mise à jour des objets qui doivent être notifiés du changement du Subject.

ConcreteObserver :

- maintient une référence à un objet ConcreteObserver
- Stocke l'ensemble des états qui doivent être compatibles avec le Subject
- implémente l'interface de mise à jour de l'Observer pour garder son état compatible avec le Subject.

Code Csharp généré automatiquement	Code PADL
<pre> //ConcreteObserver public class ConcreteObserver : Observer { public ConcreteObserver() { this.subject = subject; this.name = name; } public ConcreteSubject Subject { get { return subject; } set {subject= value; } } public override void Update() { this.observerState = subject.GetSubjectState; Console.WriteLine ('observer new state is : name,observerState'); } private string name; private string observerState; private ConcreteSubject subject; } </pre>	<pre> /***** ConcreteObserver Class***** IClass concreteObserver = Factory.GetInstance().createClass("ConcreteObserver"); Observer.addAction(concreteObserver); concreteObserver.addInheritedActor(observer1); IField name = Factory.GetInstance().createField("name", "String",2); concreteObserver.addAction(name); name.setPrivate(true); IField observerState = Factory.GetInstance().createField("observerState", "String", 2); concreteObserver.addAction(observerState); observerState.setPrivate(true); IField ConcreteSubject = Factory.GetInstance().createField("subject", "ConcreteSubject",2); concreteObserver.addAction(ConcreteSubject); ConcreteSubject.setPrivate(true); //ConcreteObserver Constructor IConstructor ConcreteObserverc = Factory.GetInstance(). createConstructor("ConcreteObserver"); concreteObserver.addAction(ConcreteObserverc); IAffectation affecte = StatementFactory.getStateInstance().createAffectation(ConcreteSubjectc,"subject"); ConcreteObserverc.addAction(affecte); IAffectation Name = StatementFactory.getStateInstance().createAffectation(name, "name"); ConcreteObserverc.addAction(Name); //Update Method ICSharpMethod update1 = CSharpFactory.getCSharpInstance().createNewMethod("Update"); concreteObserver.addAction(update1); update1.setOverride(true); IPriority subjects=CSharpFactory.getCSharpInstance().createPropriety ("Subject", "ConcreteSubject",ConcreteSubject); concreteObserver.addAction(subjects); IGetterProperty get1 = CSharpFactory.getCSharpInstance().createGetAccesseur(); subjects.setGetterPropriety(get1); ISetterProperty set1 = CSharpFactory.getCSharpInstance().createSetAccesseur(); subjects.setSetterPropriety(set1); IAffectation affecte3 = StatementFactory.getStateInstance().createAffectation (observerState,"subject.GetSubjectState"); update1.addAction(affecte3); IDOutPut out = StatementFactory.getStateInstance().createOutPut("observer new state is : name,observerState"); update1.addAction(out); </pre>

```

//ConcreteSubject
public class ConcreteSubject : Subject {
    public string GetSubjectState() {
        return subjectState;
    }
    public void SetSubjectState() {
    }
    this.subjectState = value;
}
public string subjectState;
}
//Observer
public abstract class Observer {
    public abstract void Update();
}

//*****ConcreteSubject Class*****
IClass concreteSubject = Factory.GetInstance().createClass("ConcreteSubject");
Observer.addActor(concreteSubject);
concreteSubject.addInheritedActor(subject);
IField subjectState = Factory.GetInstance().createField("subjectState", "String", 2);
concreteSubject.addActor(subjectState);
//GetSubjectState Method
ICSharpMethod concretesub =
CSharpFactory.getCSharpInstance().createNewMethod("GetSubjectState");
concretesub.setReturnType("string");
concretesub.SetReturnValue("subjectState");
concreteSubject.addActor(concretesub);
//SetSubjectState Method
ICSharpMethod concretesub2 = CSharpFactory.getCSharpInstance().createNewMethod("SetSubjectState");
concreteSubject.addActor(concretesub2);
IAffection affecte2 = StatementFactory.getStateInstance().createAffection(subjectState, "value");
concretesub2.addActor(affecte2);
//*****Observer Class*****
IClass observer1 = Factory.GetInstance().createClass("Observer");
Observer.addActor(observer1);
observer1.setAbstract(true);
//Update Method
IMethod update = Factory.GetInstance().createMethod("Update");
observer1.addActor(update);
update.setAbstract(true);
IMethodInvocation invoc = Factory.GetInstance().createMethodInvocation(2, 1, 1, subject);
invoc.addCallingField(observer);
invoc.setCalledMethod(update);
iterative.addActor(invoc);

```

State Design Pattern :

Les Classes et objets participants au patron :

Context :

- A le rôle d'interface pour définir le contexte Client
- Maintient une instance d'une sous-classe Concrete State qui définit l'état actuel

State :

- A le rôle d'interface pour encapsuler le comportement d'un état du contexte.

Concrete State :

- Chaque sous-classe implémente un comportement associé à un contexte

Code Csharp généré automatiquement	Code PADL
<pre> //ConcreteStateA public class ConcreteStateA : State { public override void Handle(Context context) { //implements a behavior associated with a Getstate of Context } } //ConcreteStateB public class ConcreteStateB : State { public override void Handle(Context context) { //implements a behavior associated with a Getstate of Context } } </pre>	<pre> //*****ConcreteStateA Class***** IClass concreteStateA = Factory.getInstance().createClass("ConcreteStateA"); state.addActor(concreteStateA); concreteStateA.addInheritedActor(Stat); //*****ConcreteStateB Class***** IClass concreteStateB = Factory.getInstance().createClass("ConcreteStateB"); state.addActor(concreteStateB); concreteStateB.addInheritedActor(Stat); //Handle Method ICSharpMethod handle2 = CSharpFactory.getGsharpInstance().createNewMethod("Handle"); concreteStateB.addActor(handle2); handle2.setOverride(true); handle2.setComment(" //implements a behavior associated with a Getstate of Context"); //Handle Method ICSharpMethod handle1 = CSharpFactory.getGsharpInstance().createNewMethod("Handle"); concreteStateA.addActor(handle1); handle1.setOverride(true); IParameter context2 = Factory.getInstance().createParameter("context", "Context"); handle2.addActor(context2); IParameter context1 = Factory.getInstance().createParameter("context", "Context"); handle1.addActor(context1); handle1.setComment(" //implements a behavior associated with a Getstate of Context "); </pre>

```

//Context
public class Context {
    public Context(State state) {
        this.state = state;
    }
    public abstract State GetState();
    public void Request() {
        state.Handle();
    }
    private State state;
}
//State
public abstract class State {
    public abstract void Handle(Context context);
}
} Le Code C# Correspondant State Design Motif

//*****Context Class*****
IClass contexte = Factory.GetInstance().createClass("Context");
state.addActor(contexte);
IField state1 = Factory.GetInstance().createField("state", "State", 2);
contexte.addActor(state1);
state1.setPrivate(true);
//Context Constructor
IConstructor constit = Factory.GetInstance().createConstructor("Context");
contexte.addActor(constit);
IParameter cont = Factory.GetInstance().createParameter("state", "State");
constit.addActor(cont);
IAffectation affecte = StatementFactory.getStateInstance().createAffectation(
state1, "state");
constit.addActor(affecte);
//getState Method
ICSharpMethod State = CSharpFactory.getOsharpInstance().createMethod("getState");
State.setAbstract(true);
contexte.addActor(State);
State.setReturnType("State");
//Request Method
IMethod request = Factory.GetInstance().createMethod("Request");
contexte.addActor(request);
IMethodInvocation invoc = Factory.GetInstance().createMethodInvocation(2, 1, 1, contexte);
invoc.addCallingField(state1);
invoc.setCalledMethod(handle);
request.addActor(invoc);
//*****ConcreteStateA Class*****
IClass concreteStateA = Factory.GetInstance().createClass("ConcreteStateA");
state.addActor(concreteStateA);
concreteStateA.addInheritedActor(State);

```