Behavioral Patterns

Behavioral Patterns

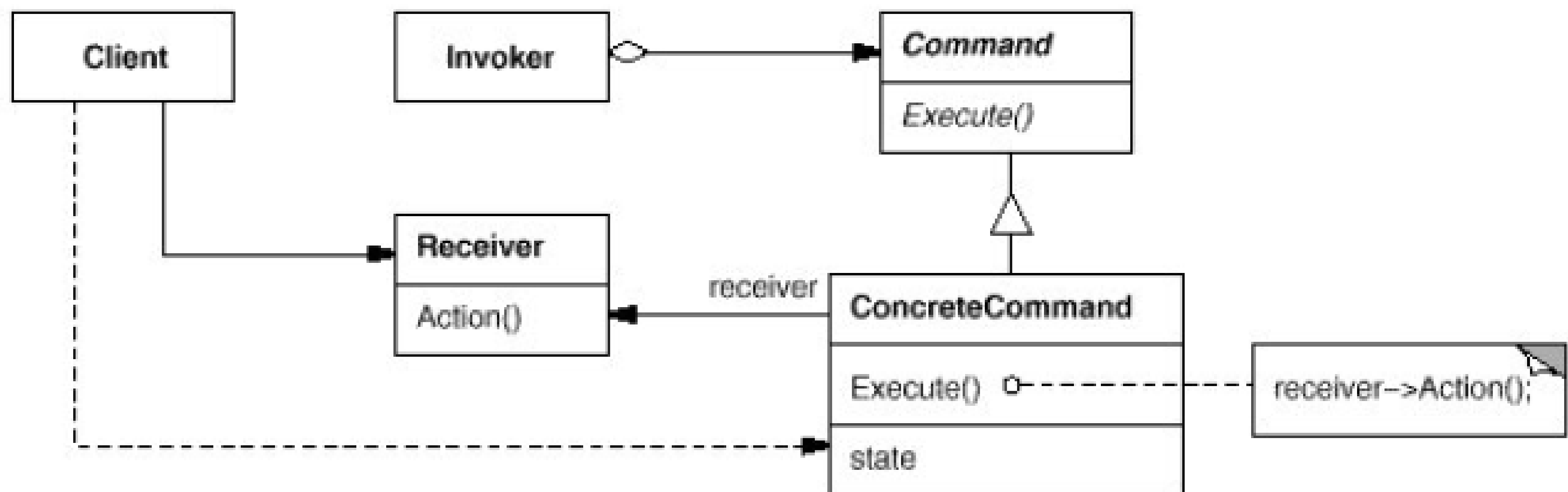
- هذه الـ patterns توصف :
  - خوارزميات
  - السلوك بين الأغراض
  - أشكال الاتصال بين الأغراض

Behavioral Patterns

- Chain of Responsibility
- **Command**
- Iterator
- Mediator
- **Observer**
- **Strategy**
- **Template Method**
- **Visiitor**

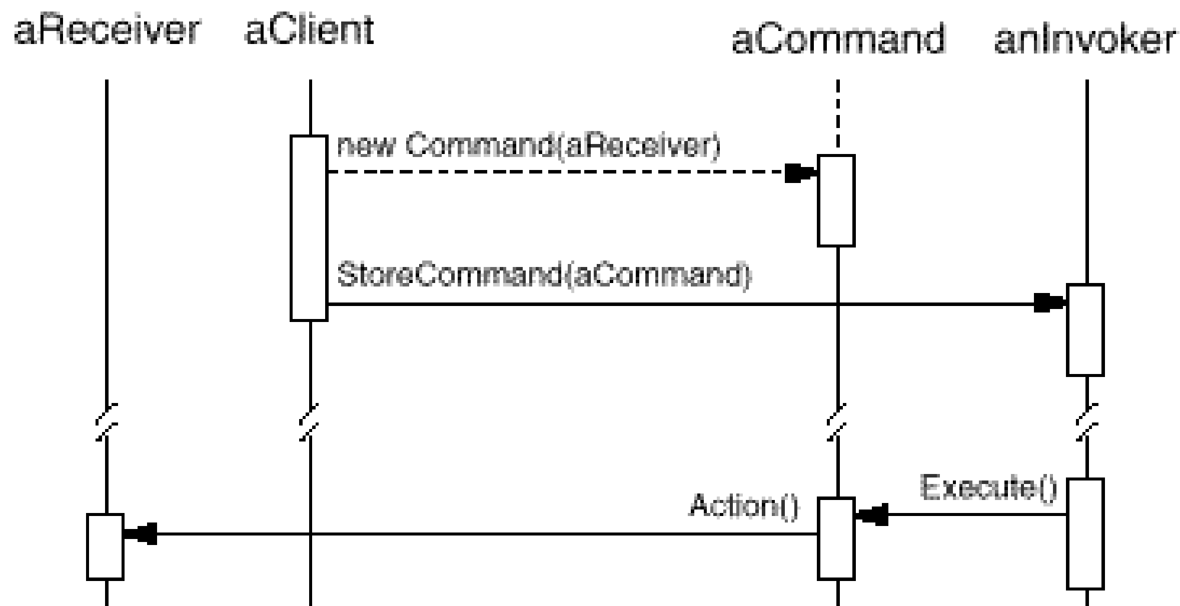
Command

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

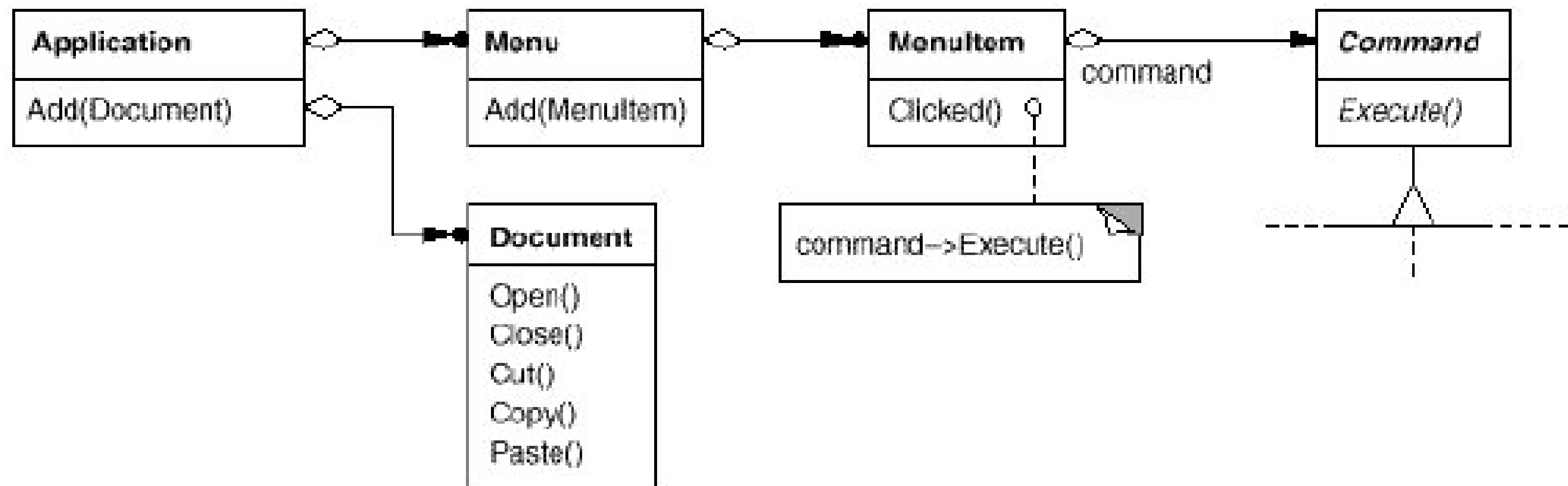


Command

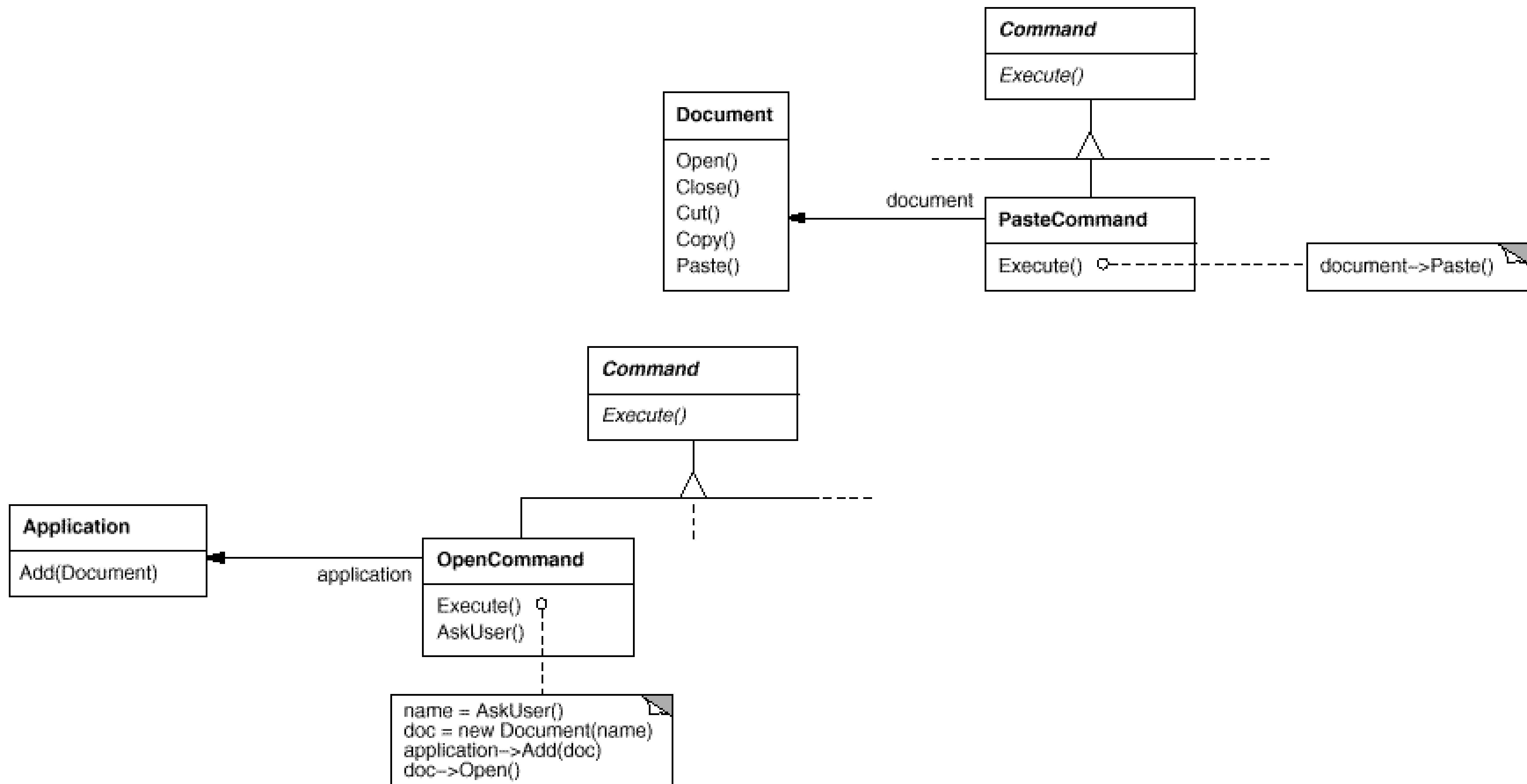
- Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it.
- We can replace commands dynamically



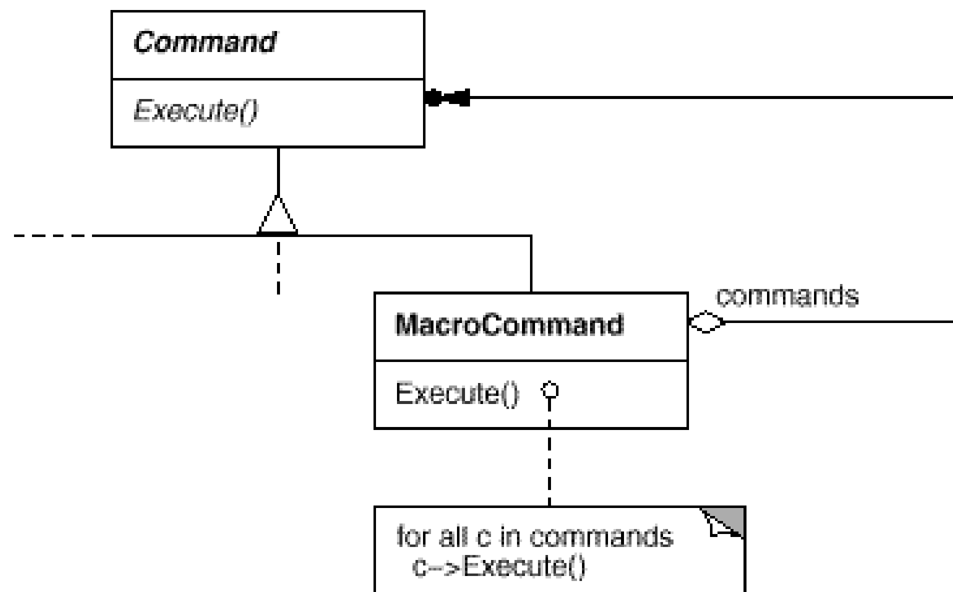
Command Example : Menu Managment



Command Example : Menu Managment



Command Example : MacroCommand



Command

```
public interface Command {
        public boolean execute();
}
public class Menu {
        private HashMap<String,Command> commands = new HashMap<String,Command>();
        public Menu () {init();}
        private void init() {
                commands.put("save",new SaveCommand());
                commands.put("load",new LoadCommand());
                commands.put("help",new HelpCommand());
                (...)
        }
        public void act() {
                System.out.println("what do you want to do (typing 'help' can use) ?");
                command = Input.readString();
                commands.get(command).execute();
        }
}
public class SaveCommand implements Command { public void execute() {(...) }}
public class LoadCommand implements Command {public void execute() { (...) }}
public class HelpCommand implements Command { public void execute() { (...) }}
-----------------------------------------------------------------------------------------------

commands.put("save",new SaveCommand()); → Dependency ? Inversion of Control ?
```

Command/ inversed dependency

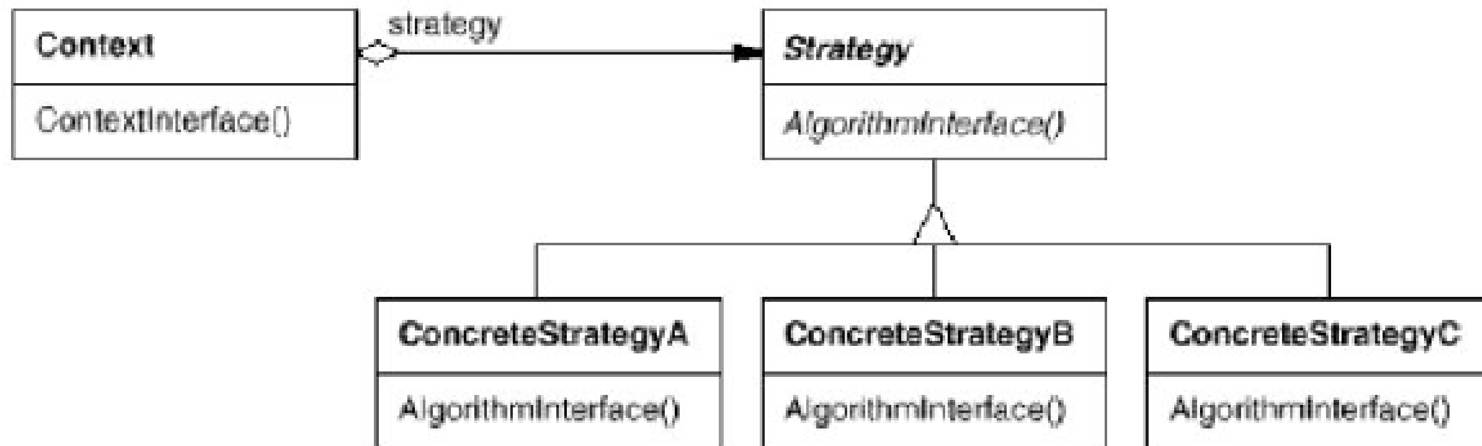
```
public interface Command {
	public boolean execute();
	public String getHelpMsg();
}
public class Menu implements Command {
	private HashMap<String,Command> commands = new HashMap<String,Command>();
	public Menu (){
		this.addCommand("help",this);
	}
	private void addCommand(String key, Command command) {
		// inversion of control
		commands.put(key,command);
	}
	public void act() { System.out.println("what do you want to do (typing 'help' can use) ?");
	commands.get(Input.readString()).execute();
	}
	public void execute() {
		// foreach keys and call getHelpMsg on the values
	}
	public String getHelpMsg() { return "display this help"; }
}
public class SaveCommand implements Command {
	public void execute() {(...)}
	public String getHelpMsg() { return "saving ..." }
}
public class LoadCommand implements Command {
	public void execute() { (...) }
	public String getHelpMsg() { return " loding ..." }
}
```

Command

- To use when:
  - OO Mannar to realize **callback** functions
  - Queue of requests + Command Object can have a lifetime indepedent of the orginal request + we can transfer Command Objects to another space of addresses
  - Undo operation : Command (with unexecute operation + state) + history stores executed commands
  - Recoveringfrom crash = rexecute the stored commandes
  - Build system of transactions (set of changes on data)

Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.



Strategy : Example 1

```
public interface IncrementFunction {
        public int increment(int value);
}
public class SimpleIncrement implements IncrementFunction { ... }
public class ModularIncrement implements IncrementFunction { ... }
public class AnotherIncrement implements IncrementFunction { ... }
public class Counter {
        private int value;
        private IncrementFunction incrementF;
        public Counter(int value, IncrementFunction incrementF) {
                this.value = value;
                this.incrementF = incrementF;
        }
        public int getCurrentValue() {
                return value;
        }
        public void increment() {
                value = incrementF.increment(value);
        }
         public void initValue(int init) { this.value = init; }
}
// ... usage
Counter simpleCounter = new Counter(0, new SimpleIncrement());
Counter modularCounter = new Counter(0, new ModularIncrement(7));
Counter anotherCounter = new Counter(0, new AnotherIncrement());
```

Strategy : Example of Games

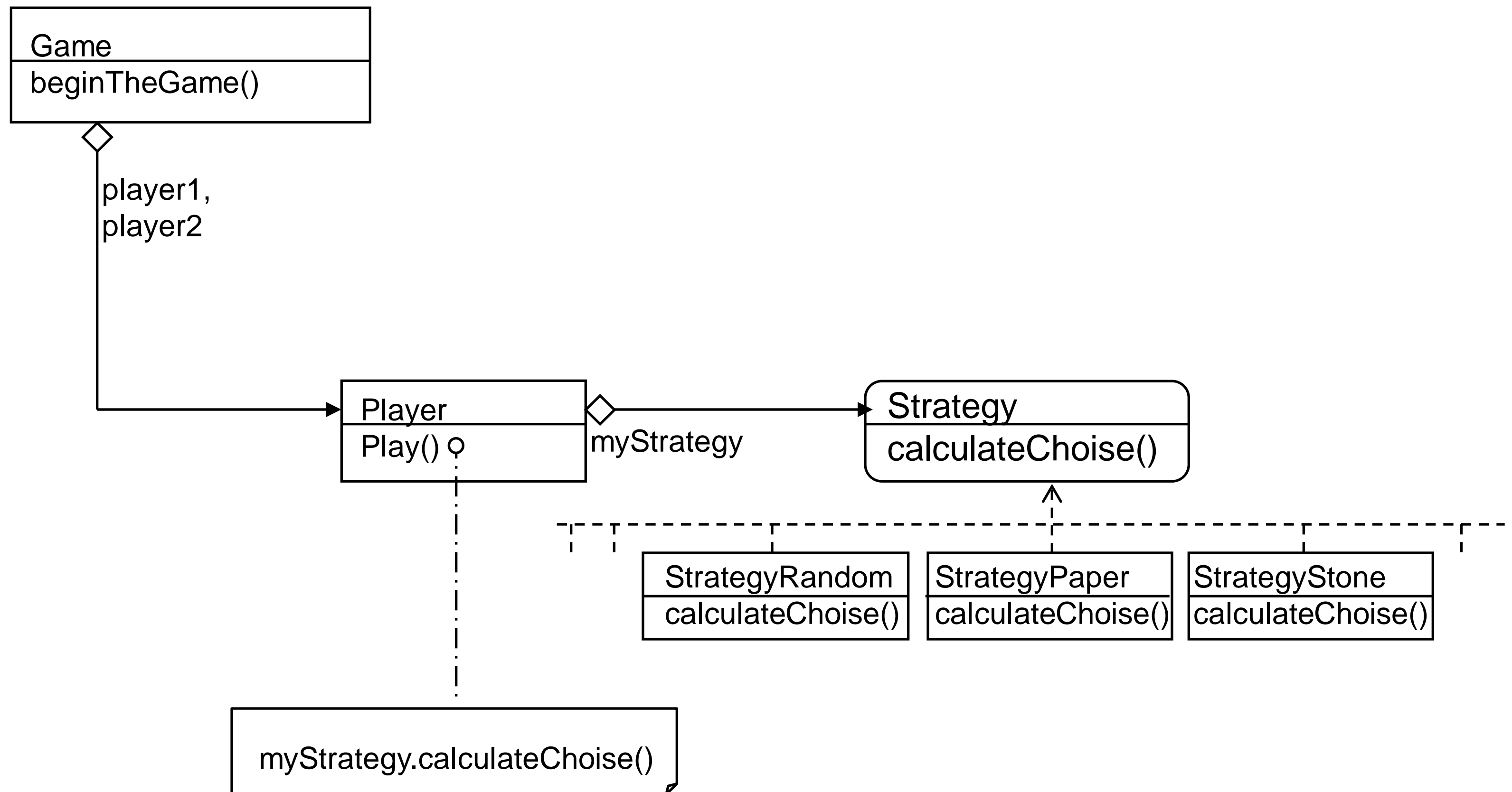
```
public interface Strategie {
        public Coup calculatePlayerChoise(...);
}
public class Player {
        private Strategie myStrategie;
        public Player(Strategie strategie, ....) {
                myStrategie = strategie;
                ...
        }
        public Coup play() {
                return myStrategie. calculatePlayerChoise(...);
        }
}
public class StrategieRandom implements Strategie {
        public Coup calculatePlayerChoise(...) {... }; // random choise
}
public class StrategieImpl implements Strategie {
        public Coup calculatePlayerChoise(...) { ... }; // another calcul
}
--------------------------------------------------------------------------------
// ... usage
Joueur player1 = new player(new StrategieRandom());
Joueur player2 = new player(new StrategieImpl());
new Game(player1, player2).beginTheGame();
```

Strategy : Game

ورقة ـ حجر ـ مقص–مسألة من السنة الثالثة (عملي لغات البرمجة) >>> لعبة

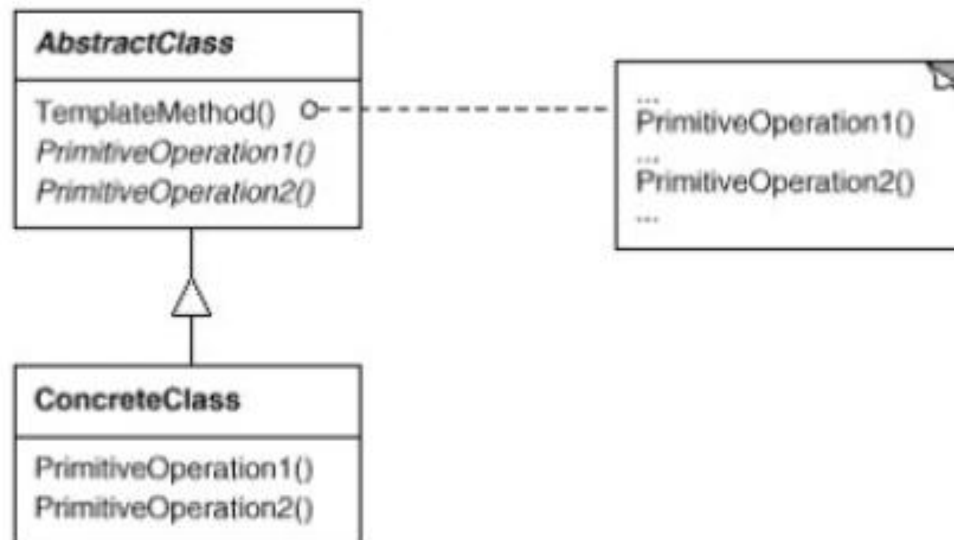
تلعب هذه اللعبة بلاعبين اثنين وفي كل دورة يلعب كلا اللاعبين إحدى الخيارات التالية: ورقة، حجر أو مقص.
نتيجة الدورة تحسب بالشكل التالي:
ـ إذا اللاعبين يلعبان نفس الخيار <تعادل> وكلا اللاعبين يحصل على نقطة
ـ الحجر يكسر المقص و من لعب الحجر يحصل على نقطتين
ـ الورقة تغلف الحجر و من لعب الورقة يحصل على نقطتين
ـ المقص يقص الورقة و من لعب المقص يحصل على نقطتين
اللعبة تلعب كسلسلة من الدورات و بعدد محدد منذ البداية. الرابح هو من يحصل على أكبر عدد من النقاط بعد انتهاء كل الدورات.
نرغب بإمكانية اللعب مع الحاسب حيث أن عدة استراتيجيات للعب ممكنة:
ـ دائماً نلعب نفس الخيار
ـ نلعب بشكل عشوائي
ـ نستخدم لوحة المفاتيح لتحديد خيارنا
- ....
- ....
اكتبوا ما يلزم لتحقيق هذه اللعبة :
ـ أنماط التعداد لخيارات اللعب
ـ الواجهة Strategy interface
ـ الصف Game
ـ الصف Player
....

Strategy : Game



Template Method

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Template Method : Example

At university, a teacher does some teaching hours. According to the teacher situation, some number of hours may be considered as complements. The complement hours are separately paid. The price of complement hour is 600 SP.

There are Three types of teachers :

Extern Teachers : all their teaching hours are considered as complement hours.

University Teachers : only the teaching hours that pass 192 hours are considered complement hours.

Master Students : the students in Master can give courses to the under-graduate students. All their teaching hours are considered as complements in the limit of 96 hours.

At the creation of a teacher, we precise the teacher name and the all his/her teaching hours.

We want to write a Java program that allow us to know : the teacher name (name() method)? and his/her complement hours (ch() method)? and the sum that he/she will take for his/her complement teaching hours (sum() method)?

1) Draw the inheritance diagram? Which classes are abstracts? Where the methods must be implemented ?

2) Write the classes in Java?

Template Method

- Applicability
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- Participants :
  - **AbstractClass**
    - defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
    - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
  - **ConcreteClass**
    - implements the primitive operations to carry out subclass-specific steps of the algorithm.

مسألة نقل البضائع

نريد نمذجة من خلال برنامج Javaحساب تكلفة نقل البضائع. البضائع المنقولة سيكونون أفراد من الصف Merchandise المعطى بالشكل التالي :

```
public class Merchandise {
        private int  weight ;      // en kg
        private int volume ;       // en dm3

        public Merchandise (int weight, int volume) {
                        this.weight = weight ;
                        this.volume = volume ;
        }

        public int weight() {
                        return weight ;
        }

        public int volume() {
                        return volume ;
        }
}
```

مسألة نقل البضائع

لبضائع يتم نقلها ضمن حاويات. الوظائف العامة للحاويات هم :

Add : التي تسمح بإضافة بضاعة ضمن الحاوية إذا كان هذا ممكن.

Cost : التي تعيد كعدد صحيح من الليرات التكلفة الكاملة لنقل هذه الحاوية.

الحاوية موصفة أيضاً بالمسافة التي ستقطعها. هذه المعلومة يتم تمريرها عند بناء الحاوية و كعدد صحيح من الكيلومترات.

الحاوية لا يمكنها أن تضم إلا عدد محدد من البضائع و التي تعتمد على كمية كاملة من البضائع التي لا يجب تعديها.
هذه الكمية يمكن تحديدها إما من خلال الوزن الكلي للبضائع أو من خلال الحجم الكلي للبضائع و ذلك حسب نوع النقل المستخدم.
نوع النقل يؤثر أيضاً على حساب تكلفة نقل الحاوية والذي يعتمد أيضاً على كمية البضائع ضمن الحاوية.

بالتالي فإننا نميز بين عدة أنماط من الحاويات حسب نوع وسيلة النقل المستعملة. مع ذلك فإنه يمكننا أن نجد بعض الصفات المشتركة بين الحاويات و التي يجب تحديدها.
الأنماط المختلفة للحاويات و صفاتهم معطاة بالجدول التالي:

مسألة نقل البضائع

النوع	الكمية	التكلفة	المحددات
نهري	وزن	مسافة × كمية	كمية <= ٣٠٠٠٠٠٠
بري	وزن	٤ × مسافة × كمية	كمية <= ٣٨٠٠٠٠
جوي	حجم	١٠ × مسافة × كمية	كمية <= ٨٠٠٠٠٠
جوي مستعجل	حجم	٢ × تكلفة الحاوية الجوية	كمية <= ٨٠٠٠٠٠

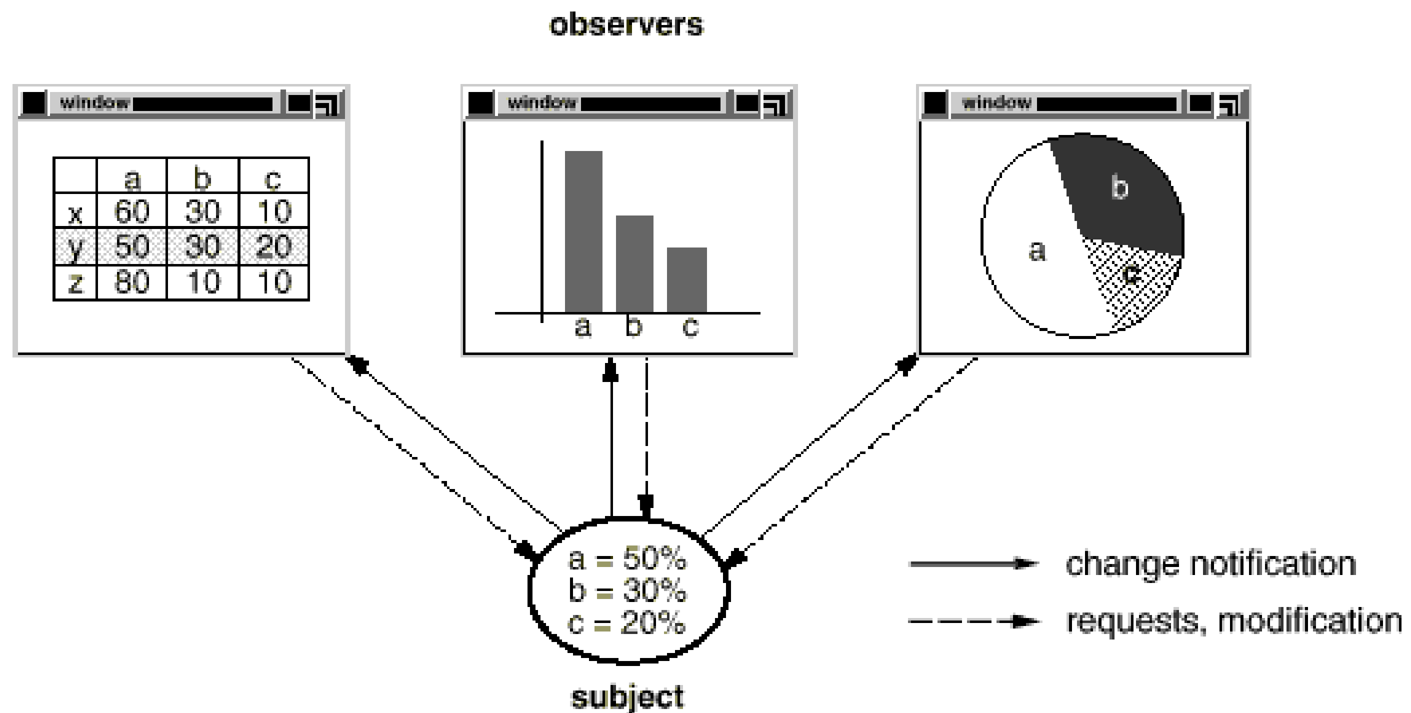
١- ارسموا مخطط الوراثة لمختلف صفوف البرنامج

٢- اكتبوا هذه الصفوف بلغة الـ Java

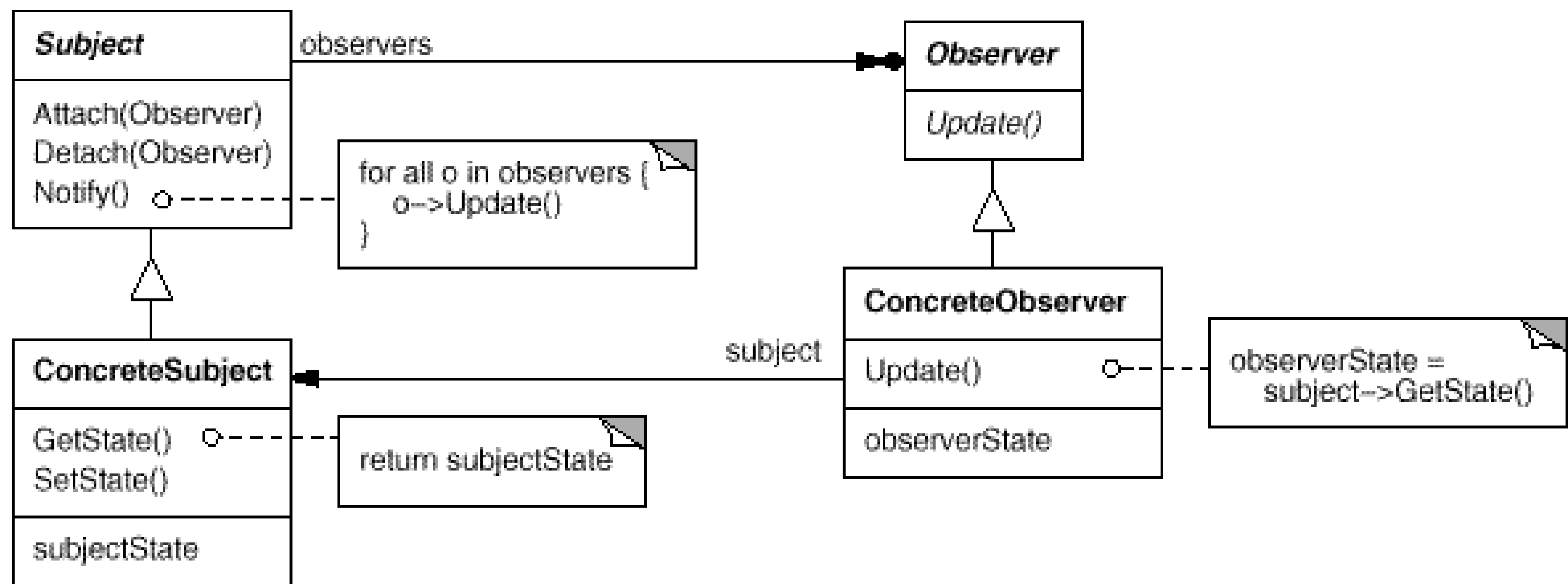
٣- اكتبوا الصف TransportCompany و الذي يدير عملية تعبئة الحاويات بالبضائع و عملية الفوترة.
يملك هذا الصف منهجاً يستقبل طلب لنقل مجموعة من البضائع و مسافة الطريق و نوعية النقل المستخدمة ثم يقوم بإنشاء الحاويات اللازمة لنقل هذه البضائع.
كما يملك هذا الصف منهجاً آخر والذي يسمح بحساب تكلفة الحاويات التي تم نقلها عبر الشركة حتى لحظة استدعاء هذا المنهج.

٤- اكتبوا الصف Test و الذي يقوم بإنشاء غرض من الشركة و مجموعة بضائع و من ثم نطلب من الشركة نقل جزء من هذه البضائع بشكل جوي مستعجل و جزء آخر بشكل بري.
بعد ذلك نطلب من الشركة حساب التكلفة النهائية.

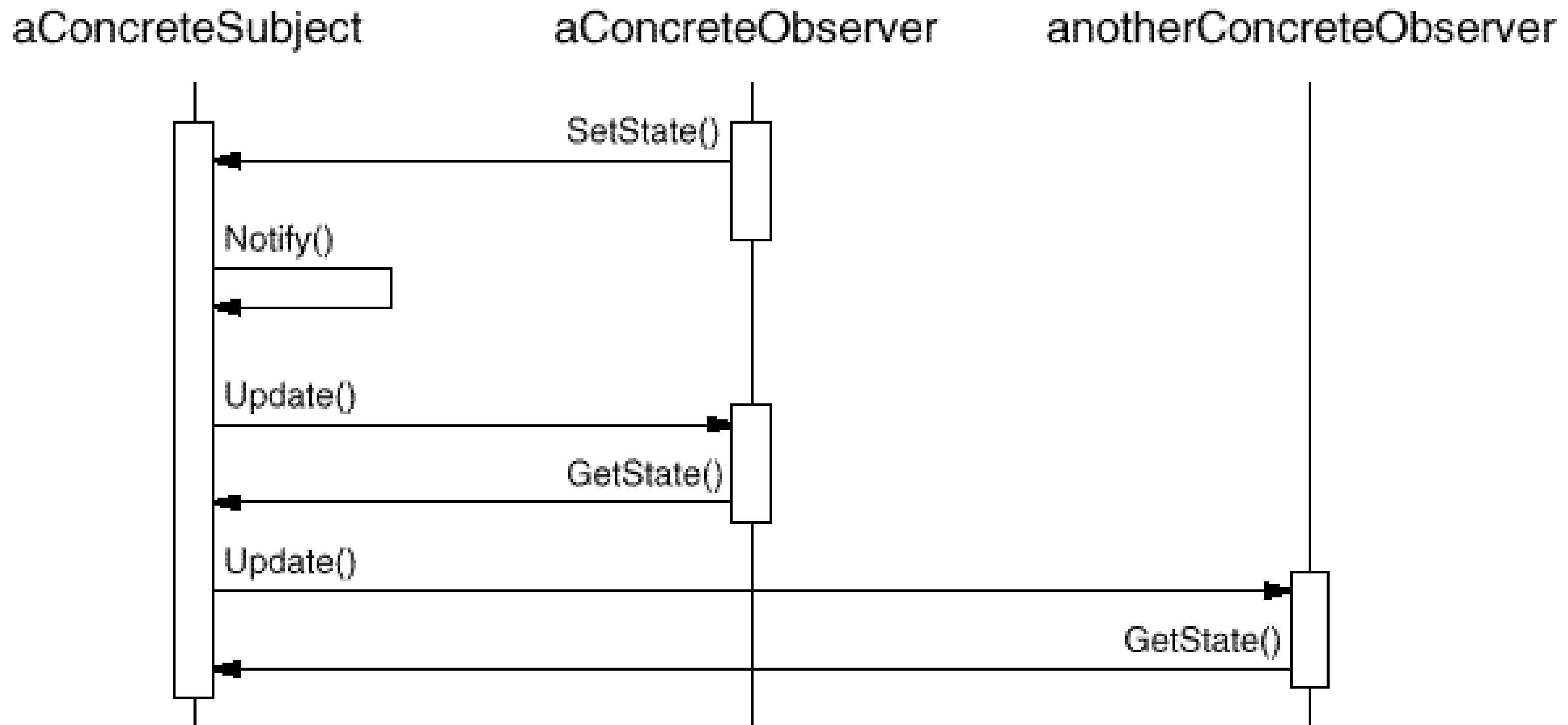
Observer/Subject



Observer/Subject : Structure



Observer/Subject : Collaboration



Observer/Subjet

- المشكلة :
  - غرض (actor) يجب أن يعمل عندما غرض آخر يلاحظ حدث ما
  - لكن ليس بالضرورة أن يعي هذا الأخير بوجود الـ actor أو الـ actors
  - الـ (re)actors يمكنهم :
    - أن يكونوا من طبيعة مختلفة
    - أن يتغيروا دون أن يشعر بذلك مصدر الحدث
  - مثال :
    - إدارة الأحداث ضمن الـ awt/swing

Observer/Subject

- المحددات :
  - عدة أغراض يمكن أن يتم إنذارهم بالأحداث المنبعثة من مصدر ما
  - عدد و طبيعة الأغراض المنذرين ليس بالضرورة معروفين عند زمن الترجمة و يمكن لهم أن يتغيروا مع الزمن
  - مصدر الحدث و مستقبله غير مرتبطين بشكل قوي
  - ==> نحتاج لآلية توكيل بين منبع الحدث (event generator) و المستقبل (listener)
- سيناريو :
  - عندما يرن هاتف فإن عدداً من الأشخاص أو الآلات يمكن أن يكونوا معنيين بالاتصال
  - إن الأشخاص يمكنهم أن يدخلوا أو يخرجوا من الغرفة و بالتالي الدخول و الخروج من حقل الانتفاع من الهاتف
- مراحل التصميم بالـ Java
  - تعريف صفوف الأحداث
  - تعريف واجهات الـ Listeners و الصفوف الـ adapters (خيارياً)
  - تعريف الصف المصدر للأحداث (event generator)
  - تعريف الصفوف المستقبلة للأحداث (Listeners)

Observer/Subject

- المرحلة ١: تعريف صفوف الأحداث
  - تعريف صف بنوع الأحداث و الذي يمكن إصداره من قبل الـ event generator
  - جعل هذه الصفوف ترث من الصف java.util.EventObject
  - تصميم صفوف الأحداث بحيث يضم الحدث كل المعلومات الضرورية الواجب إرسالها إلى الـ Observer
  - إعطاء صف الحدث اسماً من الصيغة XXXEvent

```
public class TelephoneEvent extends java.util.EventObject {
	public TelephoneEvent(Telephone source) {
		super(source);
	}
}
```

- EventObject.getSource() يسمح بالانتساب لعدة مصادر لنفس نوع الحدث
- لدينا نمطين من الـ Observer pattern
  - Pull-model حيث أن الـ Observer يستخرج المعلومات من الـ Source
  - Push-model حيث أن كل المعلومات أصلاً مرسلة مع الحدث

Observer/Subject

- **المرحلة ٢ : تعريف واجهات الـ Listeners**
  - نعرف لكل نوع من الأحداث واجهة ترث من java.util.EventListener و تحوي من أجل كل حدث على منهج و الذي سيتم إطلاقه (استدعاؤه) عند الإعلام (notification) بوقوع حدث
  - اسم هذه الواجهة هو اسم صف الحدث مع استبدال Event بـ Listener
  - أسماء مناهج الواجهة تبنى من فعل ماض يدل على حالة الـ activation
  - كل منهج يعيد void و يأخذ وسيطاً من صف الحدث

```
public interface TelephoneListener extends
    java.util.EventListener {
    public void telephoneRang(TelephoneEvent e);
    public void telephoneAnswered(TelephoneEvent e);
}
```

Observer/Subject

- **المرحلة ٢ الخيارية : تعريف صفوف الـ adapters**
  - لكل Listener فإننا نعرف صفاً ينفذه
  - اسم هذا الصف هو اسم الواجهة مع استبدال الـ Listener بالـ Adapter

```
public class TelephoneAdapter implements
    TelephoneListener {
    public void telephoneRang(TelephoneEvent e) { }
    public void telephoneAnswered(TelephoneEvent e) { }
}
```

Observer/Subject

- **المرحلة 3 : تعريف صفوف المصدرة للأحداث**
  - نعرف لكل نوع من الأحداث زوج من المناهج add/remove
  - أسماء المناهج الانتساب و فك الانتساب هي من الصيغة التالية :
    - addlistener-interface-name()
    - removelistener-interface-name()
  - و لكل منهج من واجهة الـ listeners فإننا نعرف منهج private لانتشار الحدث. هذا المنهج بلا وسطاء و يعيد void
  - هذا المنهج نسميه بالصيغة التالية :
    - Firelistener-method-name()
  - إطلاق الأحداث

Observer/Subject

```
import java.util.*;
public class Telephone {
	private ArrayList<TelephoneListener> telephoneListeners = new ArrayList<TelephoneListener>();
	public void ringPhone() {
		fireTelephoneRang();
	}
	public void answerPhone() {
		fireTelephoneAnswered();
	}
	public synchronized void addTelephoneListener(TelephoneListener l) {
		if (telephoneListeners.contains(l)){ return ; }
		telephoneListeners.add(l);
	}
	public synchronized void removeTelephoneListener(TelephoneListener l){
		telephoneListeners.remove(l);
	}
	private void fireTelephoneRang() {
		ArrayLis<TelephoneListener> tl = (ArrayList<TelephoneListener>) telephoneListeners.clone();
		if (tl.size() == 0) { return; }
		TelephoneEvent event = new TelephoneEvent(this);
		for (TelephoneListener listener : tl) {
				listener.telephoneRang(event);
		}
	}
	private void fireTelephoneAnswered() {... }
}
```

Observer/Subject

- المرحلة 4 : تعريف الصفوف المستقبلة
  - هذه الصفوف ما عليها إلا أن تنفذ الـ Listeners المرتبطة بها

```
public class AnsweringMachine implements TelephoneListener {
        public void telephoneRang(TelephoneEvent e) {
                System.out.println("AM hears the phone ringing.");
        }
        public void telephoneAnswered(TelephoneEvent e) {
                System.out.println("AM sees that the phone was answered.");
        }
}

class MyTelephoneAdapter extends TelephoneAdapter {
        public void telephoneRang(TelephoneEvent e) {
                System.out.println("I'll get it!");
        }
}

public class Person {
        public void listenToPhone(Telephone t) {
                t.addTelephoneListener(new MyTelephoneAdapter());
        }
}
```

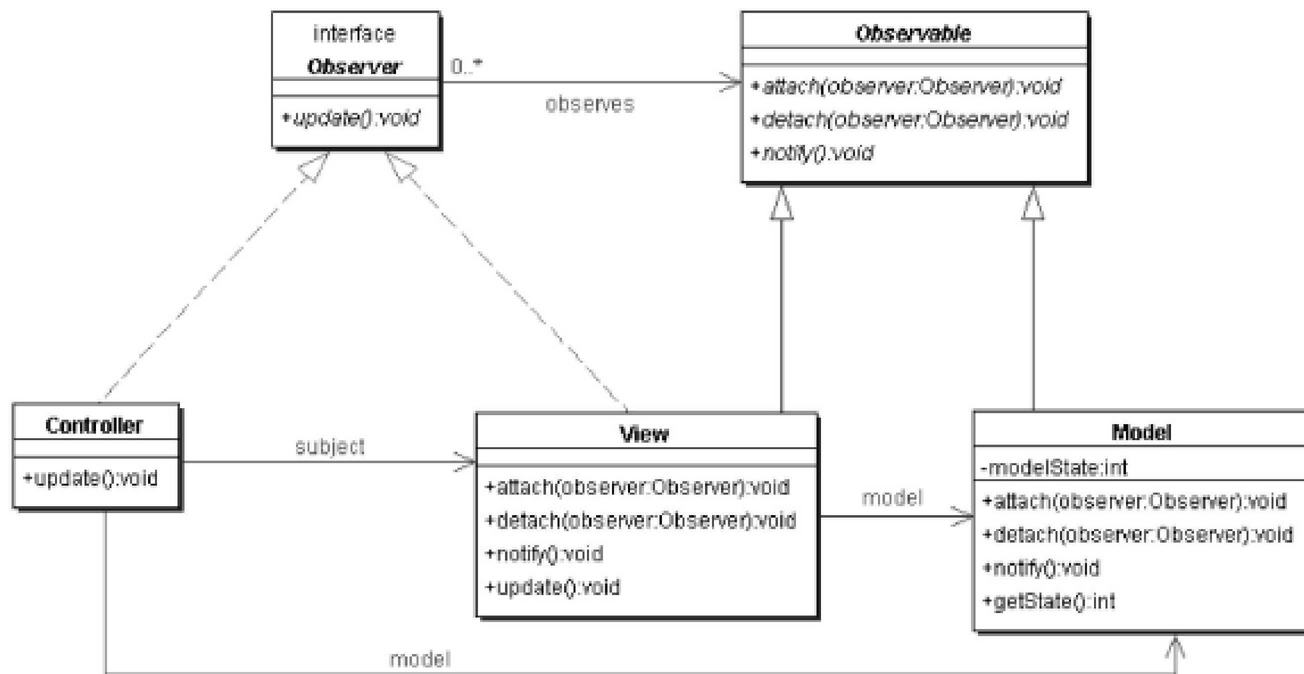
Observer/Subject

- الاستخدام :

```
public class Example1 {
    public static void main(String[] args) {
        Telephone ph = new Telephone();
        AnsweringMachine am = new AnsweringMachine();
        Person bob = new Person();

        ph.addTelephoneListener(am);
        bob.listenToPhone(ph);

        ph.ringPhone();
        ph.answerPhone();
    }
}
```

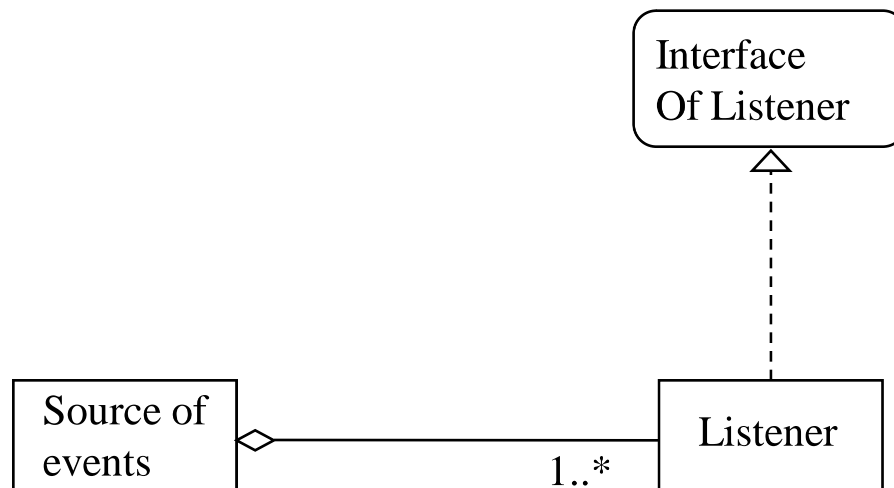
Observer/Subject : MVC

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



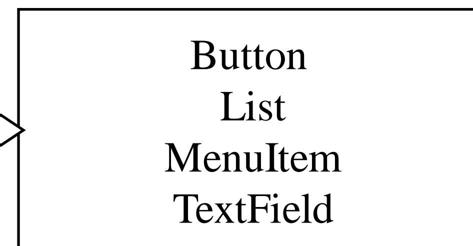
Observer/Subject

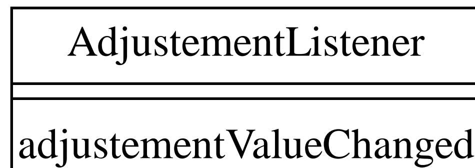
- مثال كبير : إدارة الأحداث في الواجهات الرسومية للـ Java
  - المبدأ

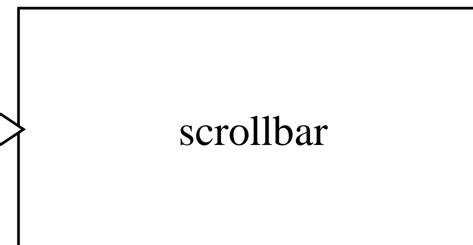


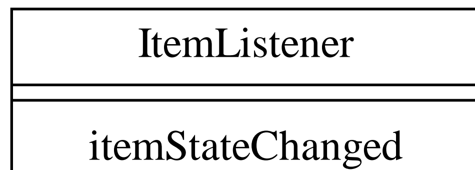
Listeners of events

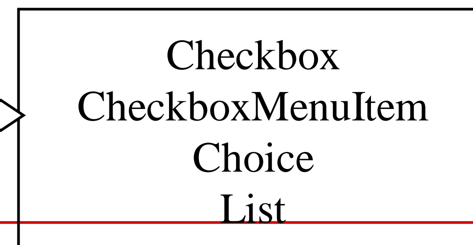
Components as sources of events





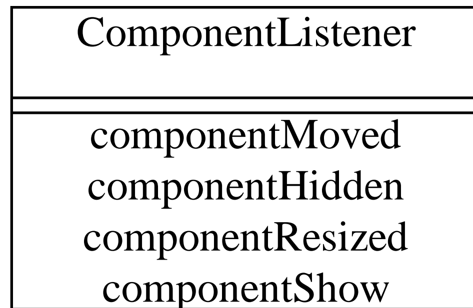


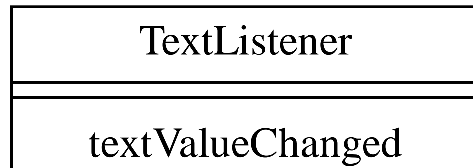


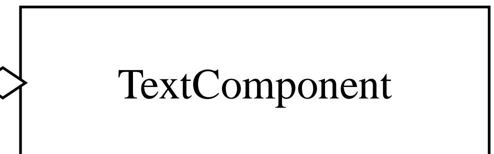


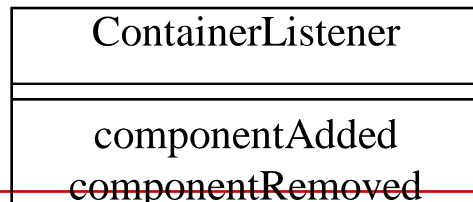
Listeners of events

Components as sources of events



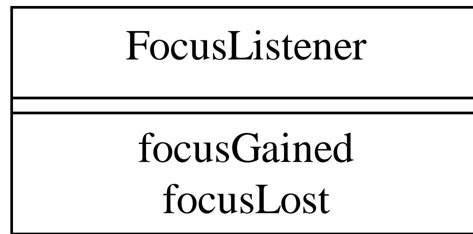


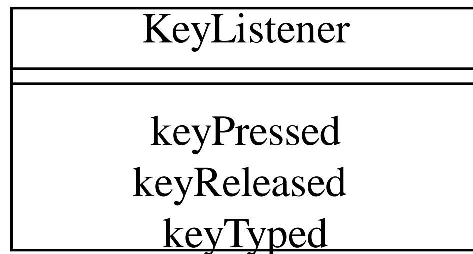


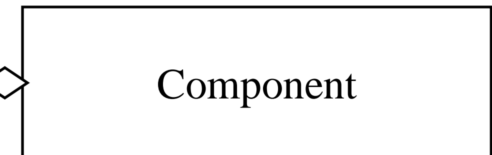


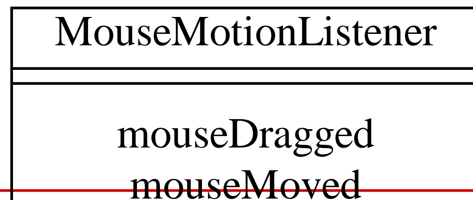
Listeners of events

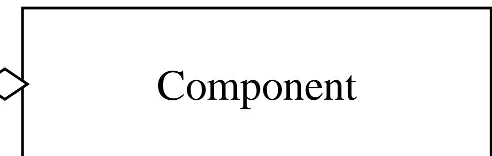
Components as sources of events





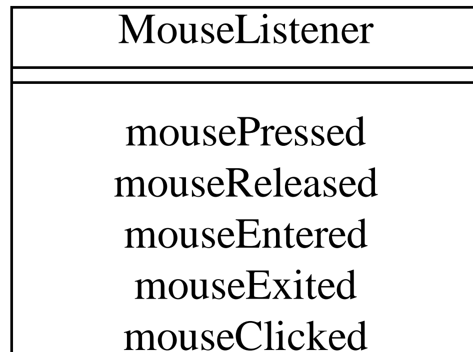


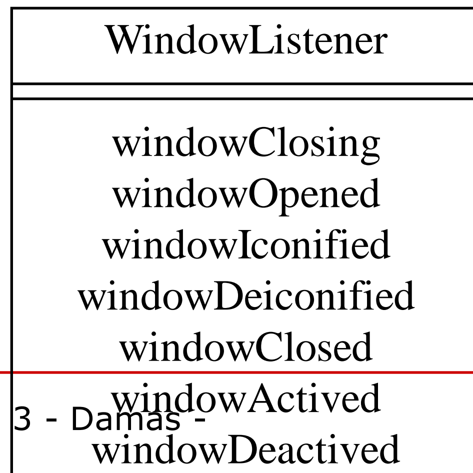




Listeners of events

Components as sources of events





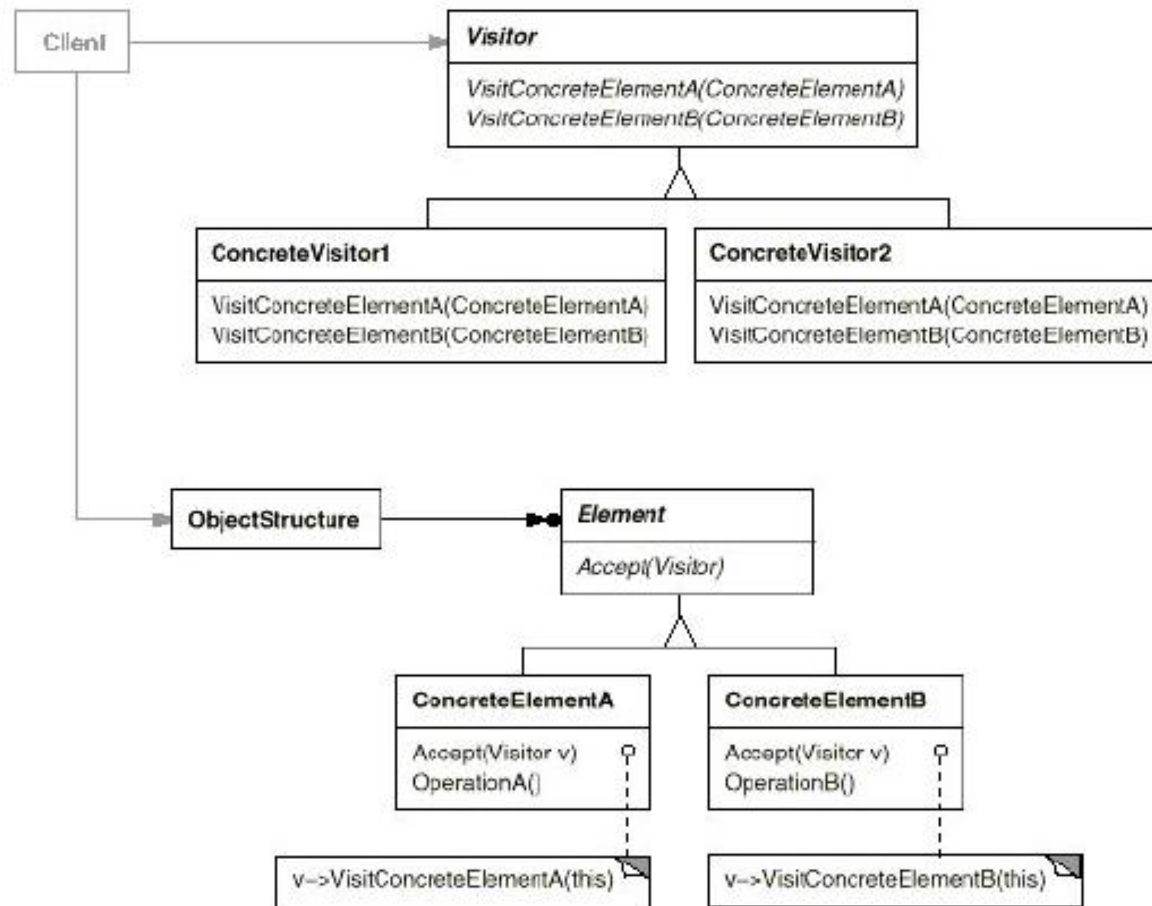
```
class ButtonsPanel extends JPanel implements
        ActionListener
{
        private JButton yellowButton, redButton,
        blueButton;

        public ButtonsPanel()
        {
                yellowButton = new JButton("Yellow");
                redButton = new JButton("Red");
                blueButton = new JButton("Blue");

                add(yellowButton);
                add(redButton);
                add(blueButton);

                yellowButton.addActionListener(this);
                redButton.addActionListener(this);
                blueButton.addActionListener(this);
        }
```

```
class ButtonsPanel extends JPanel implements
        ActionListener
{
................
        public void actionPerformed(ActionEvent e)
        {
                Object source = e.getSource();
                Color coleur = getBackground();

                if(source == yellowButton)
                                coleur = Color.yellow;
                else if (source == redButton)
                                coleur = Color.red;
                else if(source == blueButton)
                                coleur = Color.blue;

                setBackground(coleur);
                repaint();
        }
}
```

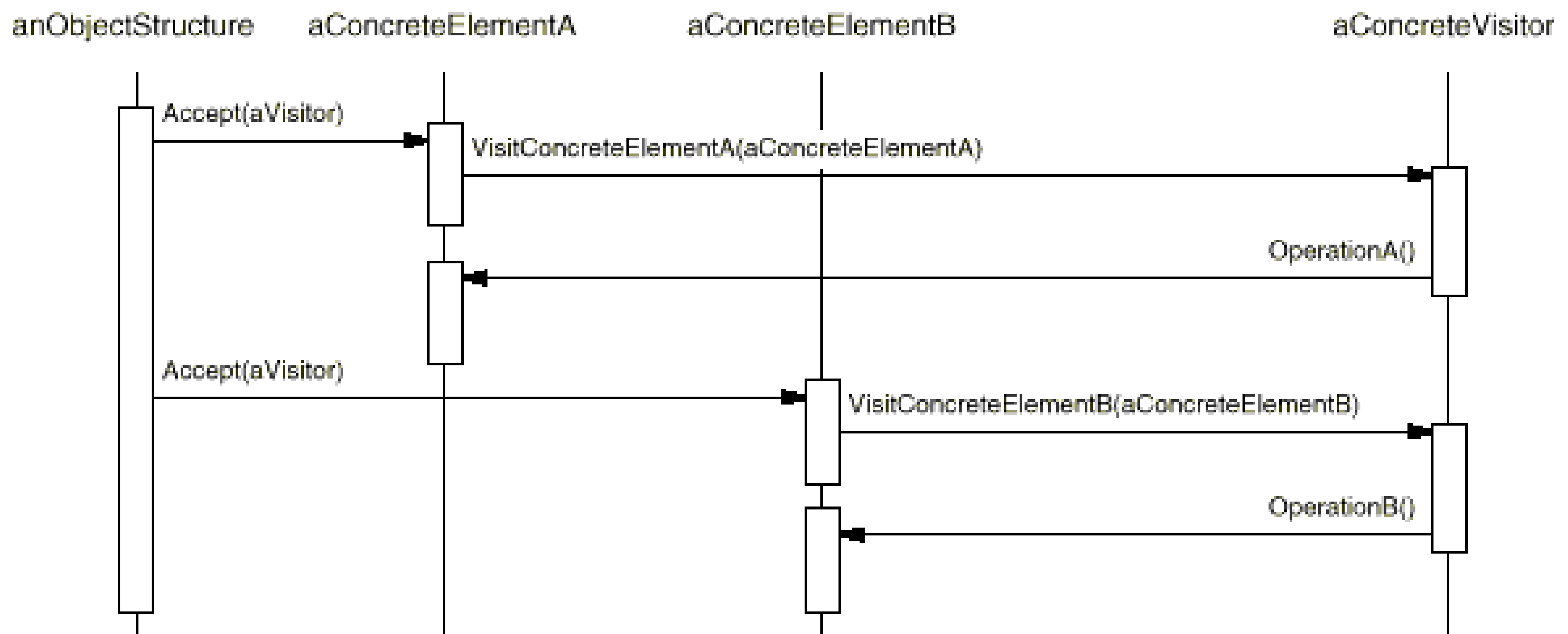
Visitor

- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- يفصل بين بنية معطيات معينة و المعالجات التي تعمل عليها
- يعمل مع الـ Composite Pattern
- لدينا هرميتين متمايزتين :
  - واحدة للمعطيات
  - واحدة للمعالجات (Visitor)
- يسمح بإضافة مناهج على هرمية المعطيات بدون تعديلها

Visitor



Visitor



Visitor

- Example : Display of collection
  - The problem = modification of object types contained in the collection

```
public void messyPrintCollection(Collection collection) {
    for (Object o : collection) {
            System.out.println(o.toString());
    }
}
```

Visitor

- Example : Display of collection
  - Introduction to processing collection of collections

```
public void messyPrintCollection(Collection collection) {
    for (Object o : collection) {
            if ( o instanceof Collection)
                    messyPrintCollection((Collection) o);
            else
                    System.out.println(o.toString());
    }
}
```

Visitor

- Example : Display of collection
  - Addition of another modification of processing

```
public void messyPrintCollection(Collection collection) {
    for (Object o : collection) {
        if (o instanceof Collection)
                    messyPrintCollection((Collection)o);
        else if (o instanceof String)
                    System.out.println("'"+o.toString()+"'");
        else if (o instanceof Float)
                    System.out.println(o.toString()+"f");
        else
                    System.out.println(o.toString());
    }
}
```

OCP?

Visitor : Solution Visitor

```
public interface Visitor {
	public void visitCollection(Collection<Visitable> collection);
	public void visitString(String string);
	public void visitFloat(Float float);
}
public interface Visitable {
	public void accept(Visitor visitor);
}
public class VisitableString implements Visitable {
	private String value;
	public VisitableString(String string) {
		value = string;
	}
	public void accept(Visitor visitor) {
		visitor.visitString(this.value);
	}
}
```

Visitor : Solution Visitor

```
public class PrintVisitor implements Visitor {
    public void visitCollection(Collection<Visitable> collection) {
        for (Visitable v : collection) {
                v.accept(this);
        }
    }
    public void visitString(String string) {
        System.out.println("'"+string+"'");
    }
    public void visitFloat(Float float) {
        System.out.println(float.toString()+"f");
    }
}
```

Visitor : Summary

- لم يعد لدينا if-then-else
- و لكن ... الكثير من الكود + تغليف أغراض للواجهة Visitable
- أيضاً، إضافة VisitorInteger <<< تغيير في الواجهة Visitor
  - الحل يكون من خلال استخدام الانعكاسية (reflection) <<< ReflectiveVisitor

ReflectiveVisitor

```
public interface ReflectiveVisitor {
	public void visit(Object o);
}
public class PrintVisitor implements ReflectiveVisitor {
	public void visit(Collection collection) { idem }
	public void visit(String string) { idem }
	public void visit(Float float) { idem }
	public void defaultVisit(Object o) { S.o.p(o.toString()); }
	public void visit(Object o) {
		try {
				Method m = getClass().getMethod("visit",
						new Class[] {o.getClass() });
				m.invoke(this, new Object[] { o });
		}
		catch (NoSuchMethodException e) {
				this.defaultVisit(o);
		}
	}
}
```

- الفوائد :
  - لم نعد بحاجة لصفوف التغليف Visitable
  - إمكانية إدارة الحالات الغير مأخوذة بعين الاعتبار (**مباشرة**) من خلال التقاط الاستثناء

And other patterns

- Selection Pattern ?
- ...
- .

Selection Pattern

A flat renting agency allows clients to choose flats according to different criteria. The agency contains several flats. The flats are defined by the number of rooms and the renting price for one month.

Give the code of the Flat class?

Give the code of the Agency class?

We defined the interface Criteria as :

```
public interface Criteria
{
    public boolean corresponds(Flat x);
}
```

- Give the code of the class CriteriaPrice? This class is an implementation of Criteria interface. This class verifies if the flat price is less than the price defined at the creation of this criteria?
- Give the code of the class CriteriaRooms? This class is an implementation of Criteria interface. It verifies if the room number of a flat is great than the room number defined at the creation of this criteria?

The Agency class has a method selection(Criteria c) that returns a list of flats that corresponds the criteria passed in parameter.

- Add the code of the selection(Criteria c) method to the Agency class?
- Add the code of the add(Flat f) method to the Agency class? This method add a flat to the agency.
- Write a Test class where :
  - You create an agency?
  - You create two flats and you add them to the agency?
  - You create a criteria that allow us to search flats whose price is less than 10000 SP?
  - Use the precedent criteria to search the Agency flats that verify this criteria? Display these flats?

Selection Pattern

Sometimes, we need to look for flats that verify several criteria.

- Write a class ComplexCriteria that can be composed from several Criteria? This class is an implementation of Criteria interface? This class verifies if a flat corresponds all the criteria composed in this Criteria?
- Write a Test class where :
  - You create an agency?
  - You create two flats and you add them to the agency?
  - You create a criteria that allow us to search flats whose price is less than 10000 SP?
  - You create a criteria that allow us to search flats whose room number is great than 5 rooms?
  - Use the precedent two criteria with an instance of ComplexCriteria in order to look for the Agency flats that verify these two criteria? Display these flats?

Selection Pattern