Object Oriented and Agile Software Development

Part 2: Object Oriented Design and Design Patterns

Christine Solnon

INSA de Lyon - 4IF - 2022/2023

Overview

1. Introduction
2. Illustration of design patterns with PlaCo
3. Other GoF Patterns

Fundamental Principles of Object Oriented Design

Protected Variations: Identify points of variation or evolution, and separate them from other parts

Low Coupling: Reduce the impact of modifications by minimising inter class dependencies

High Cohesion: Ease the understanding, management and reuse of classes by designing classes with single goals

Indirection: Decrease coupling and protect variations by adding intermediate objects

Programming for interfaces: Decrease coupling and protect variations by hiding implementation

Compose rather than inherit: Use composition instead of inheritance to delegate a task to an object, dynamically change its behavior, etc

etc...

These principles are applied in many Design Patterns!

Design Patterns

What is a Design Pattern?

- Generic solution to a frequent problem
  $\rightsquigarrow$ Formalisation of best practices

How to describe a Design Pattern?

- Name $\rightsquigarrow$ Design vocabulary
- Problem: Description of the problem and its context
- Solution: Description of the components and their relations/cooperations/roles for solving the problem
  - Generic description
  - Illustration on an example
- Consequence analysis: Time/memory complexity, impact on flexibility, portability, variation protection, coupling, cohesion, ...

23 Patterns of the Gang of Four (GoF)

[E. Gamma, R. Helm, R. Johnson, J. Vlissides 1994]

Patterns illustrated with PlaCo at the beginning of this course:

- Creation: Factory, Singleton
- Behaviour: Iterator, State, Observer, Command, Visitor
- Structure: FlyWeight

Patterns introduced at the end of this course:

- Creation: Abstract factory
- Behaviour : Strategy
- Structure: Decorator, Adaptator, Facade, Composite

Pattern introduced for the project:

- Behaviour: Template

Patterns that won't be studied in this course:

- Creation: Prototype, Builder
- Behaviour: Chain of resp., Interpretor, Mediator, Memento
- Structure: Bridge, Proxy

Overview

1. Introduction
2. Illustration of design patterns with PlaCo
3. Other GoF Patterns

PlaCo (Recalls from Part 1)

A sawmill wants a system for drawing plans and transfer them to a wood cutting machine.

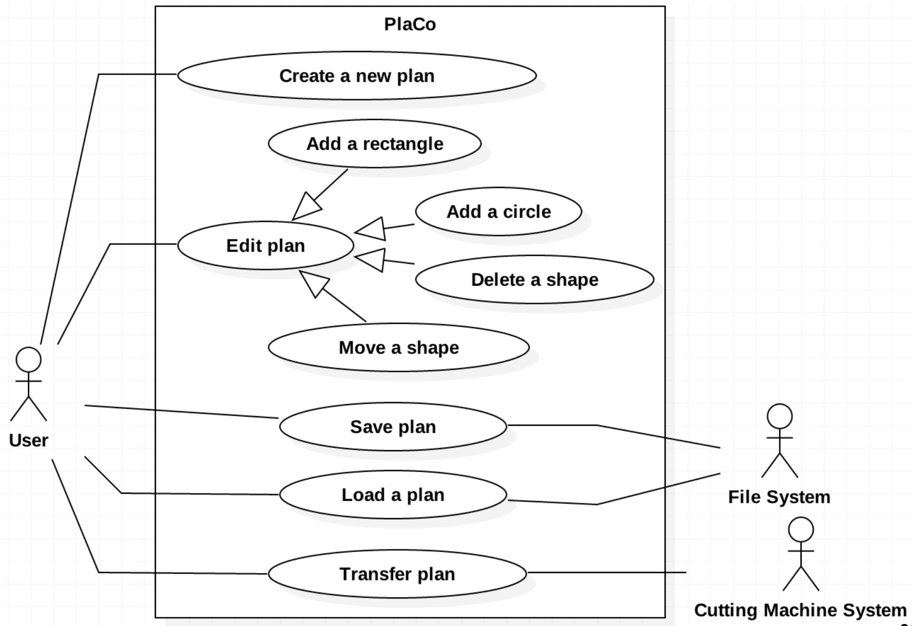
- A plan is a rectangle with an height and a width.
- The system must be able to add, delete and move shapes on a plan, to save and load plans, and to transfer a plan to the cutting machine.
- A shape is a rectangle or a circle:
  - A rectangle has an height and a width, and its position is defined by its upper left corner coordinates;
  - A circle has a radius, and its position is defined by its centre coordinates.

  Coordinates and length are integer values expressed with respect to some given unit. Shapes must have empty intersections.

Download the Java code of PlaCo on Moodle or at:

`http://perso.citi-lab.fr/csolnon/PlaCo.jar`

Use Case Diagram of PlaCo (Recalls from Part 1)



Polymorphism (not a GoF pattern...)

Problem:

In the future, the client would like to cut other kinds of shapes (triangles, ellipses, ...)

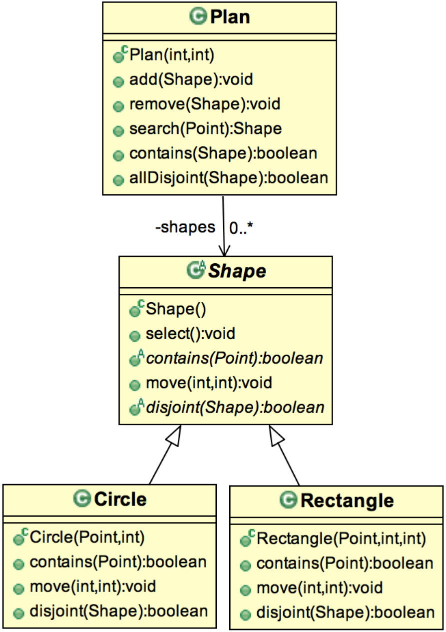
Solution: Use polymorphism

- Define an interface or an abstract class *Shape*
  $\leadsto$ Declare public methods common to all shapes
- Define classes (*Circle*, *Rectangle*, ...) that implement or extend *Shape*
- Use polymorphism to treat instances of these classes in a uniform way

Implemented principles:

- Programming for interfaces
- Protected variations

```java
public class Plan {
    private Collection<Shape> shapes;
    .....
}
public abstract class Shape{
    private boolean isSelected;
    public Shape(){
        isSelected = false;
    }
    public void select(){
        isSelected = true;
    }
    public abstract boolean contains(Point p);
    public abstract void move(int deltaX, int delt
    public abstract boolean disjoint(Shape s);
}
public class Circle extends Shape{
    private Point center;
    private int radius;
    public Circle(Point c, int r){
        super();
        this.radius = r;
        this.center = c;
    }
    @Override
    public boolean contains(Point p) {
        return center.distance(p) <= radius;
    }
    @Override
    public void move(int deltaX, int deltaY) {
        center = center.move(deltaX, deltaY);
    }
    .....
}
```



GoF Pattern: Iterator (1/3)

Problem:

The development team may change the data structure used to store shapes

Solution:

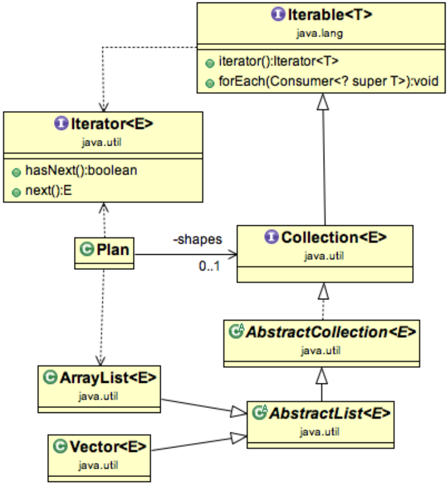
Use Iterators to traverse all elements of a collection without knowing the data structure used to implement the collection

Implemented principles:

- Programming for interfaces
- Protected variations
- High cohesion

GoF Pattern: Iterator (2/3)

```java
public class Plan {
    private Collection<Shape> shapes;
    public Plan(){
        shapes = new ArrayList<Shape>();
    }
    public void selectAll(){
        Iterator<Shape> it = shapes.iterator()
        while (it.hasNext())
            it.next().select();
    }
    public void selectAllJava5(){
        for (Shape s : shapes)
            s.select();
    }
    public void selectAllJava8(){
        shapes.forEach(s -> s.select());
    }
}
```



- What should we change to use *Vector* instead of *ArrayList*?
- Why separating *Iterator* from *Collection* ?

GoF Pattern: Iterator (3/3)

Separating *Iterator* from *Collection* makes it possible to have several iterators on a same collection at a same time

```
public boolean allDisjoint(){
    for (Shape s1 : shapes){
        for (Shape s2 : shapes){
            if ( s1 != s2 && !s2.disjoint(s1))
                return false;
        }
    }
    return true;
}
```

Model-View-Controller Architecture (Recalls from 3IF)

Problems:

- The user may require to change the way she interacts with PlaCo:
  - Use a dropdown menu (instead of buttons) to trigger use cases
  - Add a textual description of the plan (besides the graphical view)
  - Change the way coordinates are entered when adding a new shape to the plan
  - etc
- The technology used for the GUI may change
- Plan is less cohesive if it contains instructions for displaying shapes

Solution:

MVC Architecture!

MVC Architecture: Illustration with PlaCo

Model: Update and treat Data

- Update Data when adding/deleting/moving shapes in the plan
- Check that shapes have empty intersections

View: Display *Model* and interact with the user

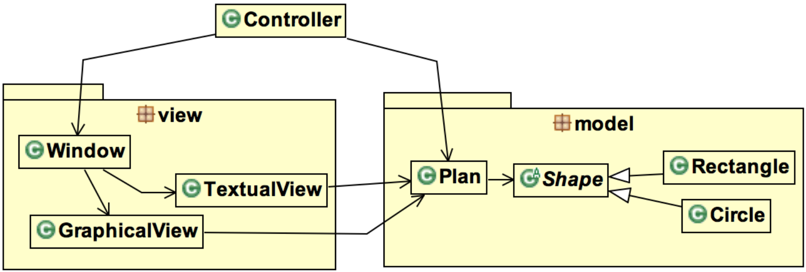
- Display the plan (graphically and as a textual list of shapes)
- Detect actions from the user (mouse click, key pressed, etc)

Controller: Translates user interactions with *View* into actions

- Ask *Model* to move selected shapes when the user presses arrows
- ... etc

Implemented principles: Protected variations and High cohesion

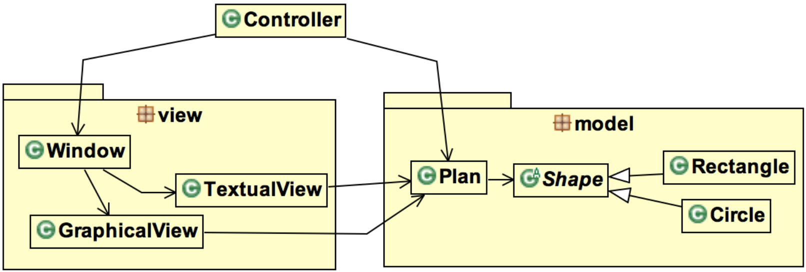
MVC Architecture: Illustration with Placo



Problem: How to notify *View* that *Model* has been modified?

- Solution 1: *Model* sends messages to *View* each time it is modified
  Drawback: *Model* becomes dependent from *View*
- Solution 2: Use the pattern *Observer*

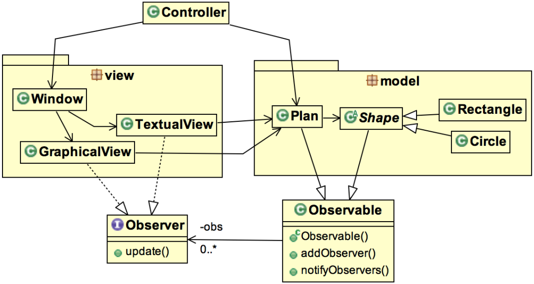
MVC Architecture: Illustration with Placo



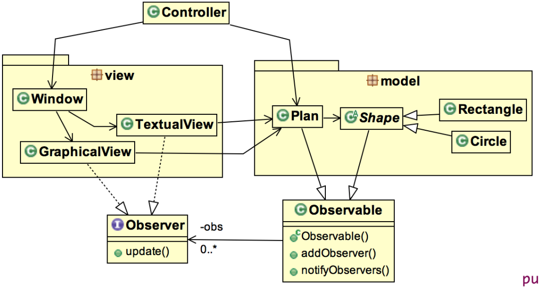
Problem: How to notify *View* that *Model* has been modified?

- Solution 1: *Model* sends messages to *View* each time it is modified
  Drawback: *Model* becomes dependent from *View*
- Solution 2: Use the pattern *Observer*

GoF Pattern: Observer (aka Publish/Subscribe) (1/2)



GoF Pattern: Observer (aka Publish/Subscribe) (1/2)

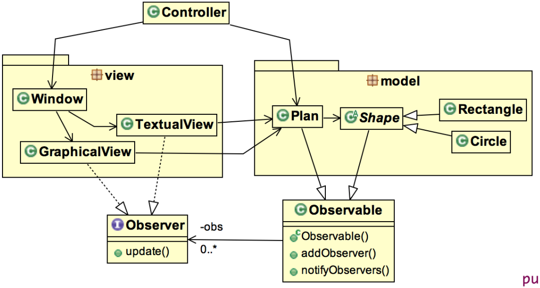



```java
public class Plan extends Observable {
    public void add(Shape s){
        shapes.add(s);
        notifyObservers(s);
    }
```

...etc

```java
public class Observable {
    private Collection<Observer> obs;
    public Observable(){
        obs = new ArrayList<Observer>();
    }
    public void addObserver(Observer o){
        if (!obs.contains(o)) obs.add(o);
    }
    public void notifyObservers(Object arg){
        for (Observer o : obs)
            o.update(this, arg);
    }
}
```

GoF Pattern: Observer (aka Publish/Subscribe) (1/2)




```java
public class Plan extends Observable {
    public void add(Shape s){
        shapes.add(s);
        notifyObservers(s);
    }
```

...etc

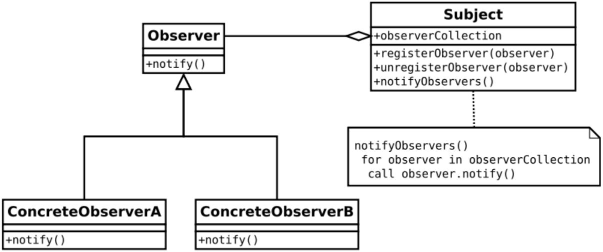
```java
public interface Observer {
    public void update(Observable observed, Object arg);
}
```

```java
public class GraphicalView implements Observer{
    public GraphicalView(Plan plan) {
        plan.addObserver(this);
        ....
    }
    @Override
    public void update(Observable o, Object arg) {
        // code for displaying this
        ....
    }
```

```java
public class Observable {
    private Collection<Observer> obs;
    public Observable(){
        obs = new ArrayList<Observer>();
    }
    public void addObserver(Observer o){
        if (!obs.contains(o)) obs.add(o);
    }
    public void notifyObservers(Object arg){
        for (Observer o : obs)
            o.update(this, arg);
    }
}
```

GoF Pattern: Observer (aka Publish/Subscribe) (2/2)

Generic Solution [Wikipedia]:



Principles implemented:

- Low coupling between *ConcreteObserver* and *Subject*
- Protected variations: *Observers* are added without modifying *Subject*

How does *ConcreteObserver* get *Subject* data?

- Push data with *notify* or pull them with getters

java.util.Observer **and** *java.util.Observable* **deprecated since Java 9**

Why? (according to *Oracle*)

- The event model supported by *Observer* and *Observable* is quite limited
- The order of notifications delivered by *Observable* is unspecified
- State changes are not in one-for-one correspondence with notifications

Alternative solutions:

- *java.beans* for a richer event model
- *java.util.concurrent* for reliable and ordered messaging among threads
- *Flow API* for reactive streams style programming

But this doesn't mean that the design pattern isn't good!

- It is used in *Listeners*
- It is easy to implement and customise

GoF Pattern: Visitor (1/3)

Problem:

The actual classes of *Shape* instances are lost

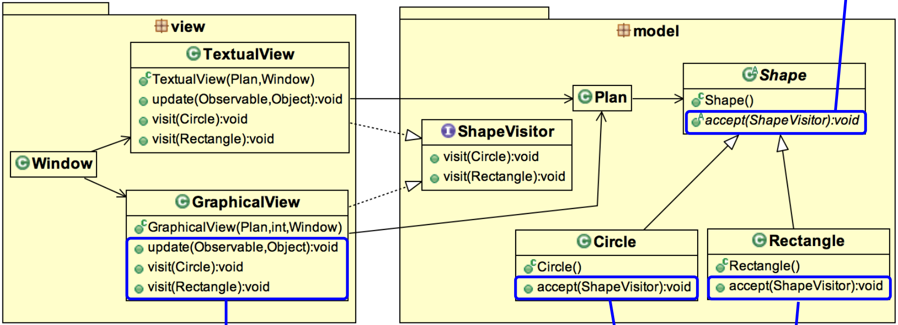
Solution 1: Test the classes of *Shape* instances before displaying them

```
@Override
public void update(Observable o, Object arg) {
    for (Shape s : plan.getShapes()) display(s);
}
private void display(Shape s){
    if (s instanceof Circle) display((Circle)s);
    else display((Rectangle)s);
}
private void display(Circle c){
    // code for displaying a circle
}
private void display(Rectangle r){
    // code for displaying a rectangle
}
```

Solution 2: Use *Visitor*

GoF Pattern: Visitor (2/3)

```java
public abstract void accept(ShapeVisitor v);
```

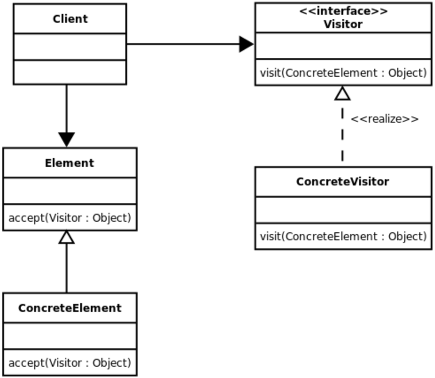



```java
public void update(Observable o, Object arg) {
    for (Shape s : plan.getShapes())
        s.accept(this);
}
public void visit(Circle c) {
    // code for displaying a circle
}
public void visit(Rectangle r) {
    // code for displaying a rectangle
}
```

```java
@Override
public void accept(ShapeVisitor v){
    v.visit(this);
}
```

GoF Pattern: Visitor (3/3)

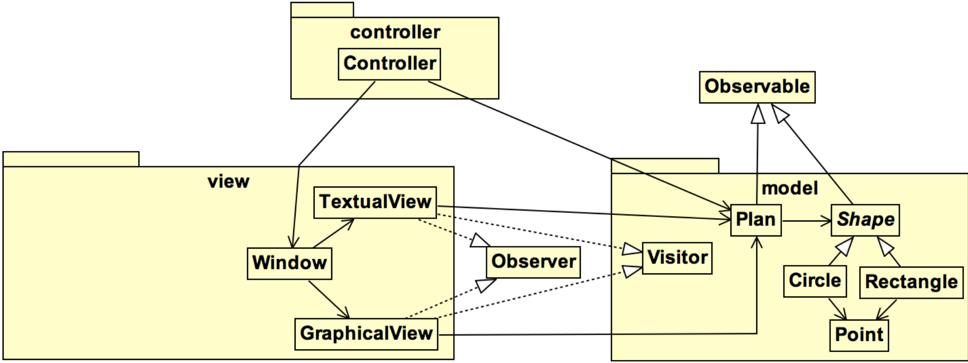
Generic solution [Wikipedia]:



Implemented principles:

- High cohesion: Group into each *Visitor* realisation all methods related to a same goal (graphical view, textual view, XML serialisation, ...) for all subclasses of *Element*
- Protected variations: New *Visitor* realisations may be added without modifying *ConcreteElement*

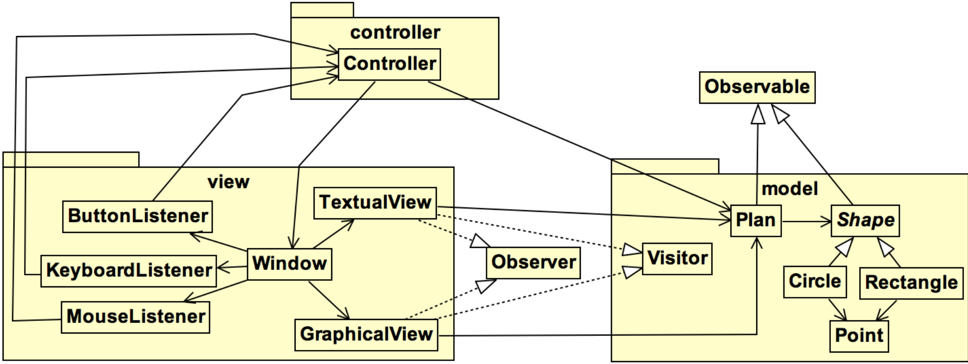
Current Architecture of PlaCo



How does the user interact with PlaCo?

How to identify the events that must be listened?

Current Architecture of PlaCo

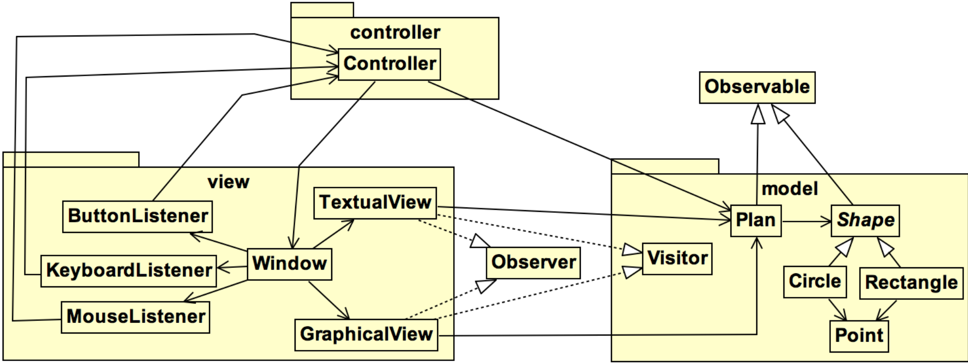


How does the user interact with PlaCo?

Window uses event listeners

How to identify the events that must be listened?

Current Architecture of PlaCo



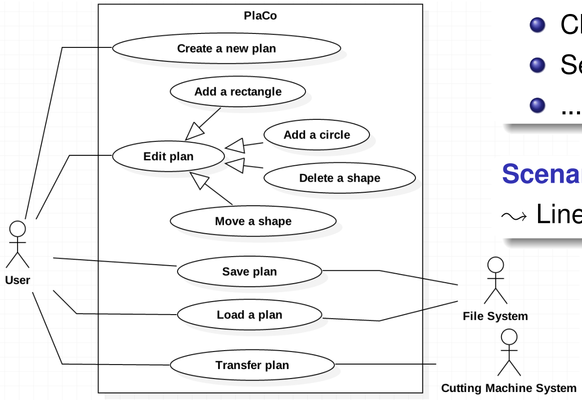
How does the user interact with PlaCo?

Window uses event listeners

How to identify the events that must be listened?

By looking at Use Cases

Using Use Cases to Identify Events



Each use case is activated by an event:

- Click on a button
- Selection of a menu item
- ...etc...

Scenarios describe user actions:

$\leadsto$ Lines starting by "The user ..."

Identification of events from scenarios

Example: Add a rectangle

1. The user tells the system she wants to add a rectangle
2. The system asks to enter the coordinates of a first corner
3. The user enters the coordinates of a point p_1
4. The system asks to enter the coordinates of the opposite corner
5. The user enters the coordinates of a point p_2
6. The system adds the rectangle defined by (p_1, p_2) in the plan and displays the plan

Extension [1-5a]: The user tells the system she wants to cancel the action

User events:

- Left click on the button "Add a rectangle"
- Left click on the graphical view of the plan
- Right click or [Esc]

Identification of events from scenarios

Example: Add a rectangle

1. The user tells the system she wants to add a rectangle
2. The system asks to enter the coordinates of a first corner
3. The user enters the coordinates of a point p_1
4. The system asks to enter the coordinates of the opposite corner
5. The user enters the coordinates of a point p_2
6. The system adds the rectangle defined by (p_1, p_2) in the plan and displays the plan

Extension [1-5a]: The user tells the system she wants to cancel the action

User events:

- Left click on the button "Add a rectangle"
- Left click on the graphical view of the plan
- Right click or [Esc]

Identification of events from scenarios

Example: Add a rectangle

1. The user tells the system she wants to add a rectangle
2. The system asks to enter the coordinates of a first corner
3. The user enters the coordinates of a point p_1
4. The system asks to enter the coordinates of the opposite corner
5. The user enters the coordinates of a point p_2
6. The system adds the rectangle defined by (p_1, p_2) in the plan and displays the plan

Extension [1-5a]: The user tells the system she wants to cancel the action

User events:

- Left click on the button "Add a rectangle"
- Left click on the graphical view of the plan
- Right click or [Esc]

Identification of events from scenarios

Example: Add a rectangle

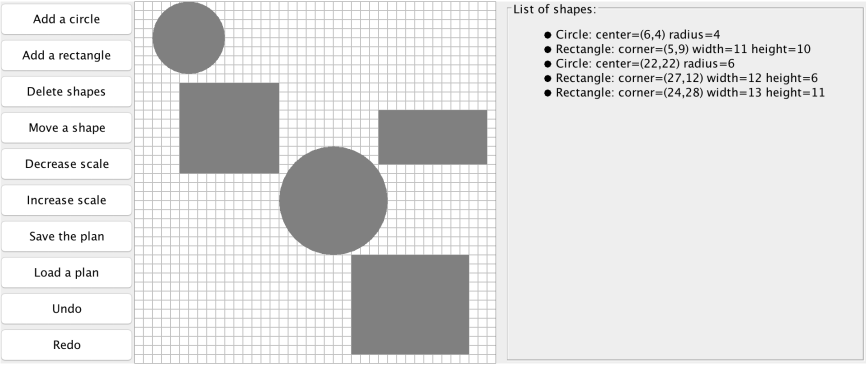
1. The user tells the system she wants to add a rectangle
2. The system asks to enter the coordinates of a first corner
3. The user enters the coordinates of a point p_1
4. The system asks to enter the coordinates of the opposite corner
5. The user enters the coordinates of a point p_2
6. The system adds the rectangle defined by (p_1, p_2) in the plan and displays the plan

Extension [1-5a]: The user tells the system she wants to cancel the action

User events:

- Left click on the button "Add a rectangle"
- Left click on the graphical view of the plan
- Right click or [Esc]

Example of GUI and List of User Events for PlaCo



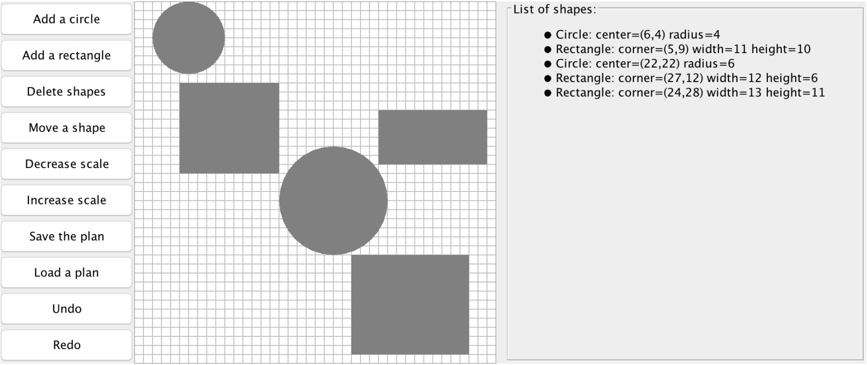
User Events:

- Click on a button: Add a circle, Add a rectangle, . . . , Undo, Redo
- Key stroke: [$\rightarrow$], [$\leftarrow$], [$\uparrow$], [$\downarrow$], [Ctr Z], [Shift Ctr Z], [Esc]
- Left click on the graphical view
- Right click on the graphical view
- Mouse move on the graphical view

Note: This GUI may not be the most user-friendly one...

We study here how to design PlaCo so that we can easily change the GUI!

Example of GUI and List of User Events for PlaCo



User Events:

- Click on a button: Add a circle, Add a rectangle, . . . , Undo, Redo
- Key stroke: [→], [←], [↑], [↓], [Ctr Z], [Shift Ctr Z], [Esc]
- Left click on the graphical view
- Right click on the graphical view
- Mouse move on the graphical view

Note: This GUI may not be the most user-friendly one...

We study here how to design PlaCo so that we can easily change the GUI!

What do Listeners do when catching a user event?

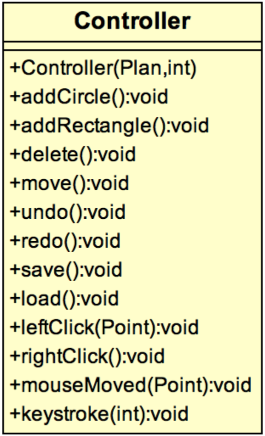
They send a message to *Controller*

Illustration with *ButtonListener*:

```
public class ButtonListener implements ActionListener {
    private Controller controller;
    public ButtonListener(Controller controller){
        this.controller = controller;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        switch (e.getActionCommand()){
        case Window.ADD_CIRCLE: controller.addCircle(); break;
        case Window.ADD_RECTANGLE: controller.addRectangle(); break;
        case Window.DELETE: controller.delete(); break;
        case Window.SAVE: controller.save(); break;
        case Window.LOAD: controller.load(); break;
        case Window.UNDO: controller.undo(); break;
        case Window.REDO: controller.redo(); break;
        case Window.MOVE: controller.move();break;
        }
    }
}
```

What Does *Controller* Do?

Controller has a method for each user event:



How to define these methods?

- Exploit use case scenarios

Illustration on *leftClick(Point p)*

Main scenario of the use case "Add a rectangle":

1. The user tells the system she wants to add a rectangle
2. The system asks to enter the coordinates of a first corner
3. The user enters the coordinates of a point p_1
4. The system asks to enter the coordinates of the opposite corner
5. The user enters the coordinates of a point p_2
6. The system adds the rectangle defined by (p_1, p_2) in the plan

Steps 3 and 5 are triggered by *leftClick(Point p)*

Problem:

The behaviour of *leftClick(Point p)* depends on the current scenario step:

- Step 1: Ignore the event
- Step 3: Ask the user to enter the coordinates of a second point
- Step 5: Add the rectangle to the plan

$\leadsto$ **Draw a Statechart diagram**

StateChart Diagram for “Add a rectangle”

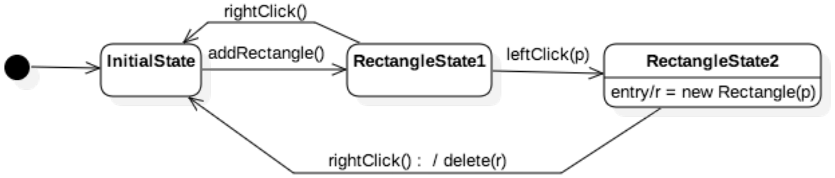
1. The user clicks on the button "Add a rectangle"
2. The system asks to enter the coordinates of a first corner
3. The user clicks on a point p
4. The system creates a small rectangle r at point p and visualizes it
5. The user moves the mouse to another point p
6. The system updates the size of r wrt p
7. The user clicks on another point p
8. The system updates the size of r wrt p and returns to the initial state

Extension [1-8a]: The user cancels the action with a right click

StateChart Diagram for “Add a rectangle”

1. The user clicks on the button "Add a rectangle"
2. The system asks to enter the coordinates of a first corner
3. The user clicks on a point p
4. The system creates a small rectangle r at point p and visualizes it
5. The user moves the mouse to another point p
6. The system updates the size of r wrt p
7. The user clicks on another point p
8. The system updates the size of r wrt p and returns to the initial state

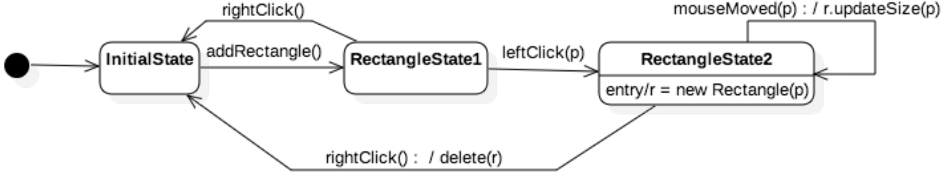
Extension [1-8a]: The user cancels the action with a right click



StateChart Diagram for “Add a rectangle”

1. The user clicks on the button "Add a rectangle"
2. The system asks to enter the coordinates of a first corner
3. The user clicks on a point p
4. The system creates a small rectangle r at point p and visualizes it
5. The user moves the mouse to another point p
6. The system updates the size of r wrt p
7. The user clicks on another point p
8. The system updates the size of r wrt p and returns to the initial state

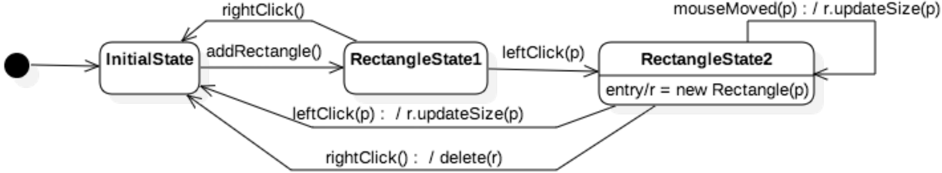
Extension [1-8a]: The user cancels the action with a right click



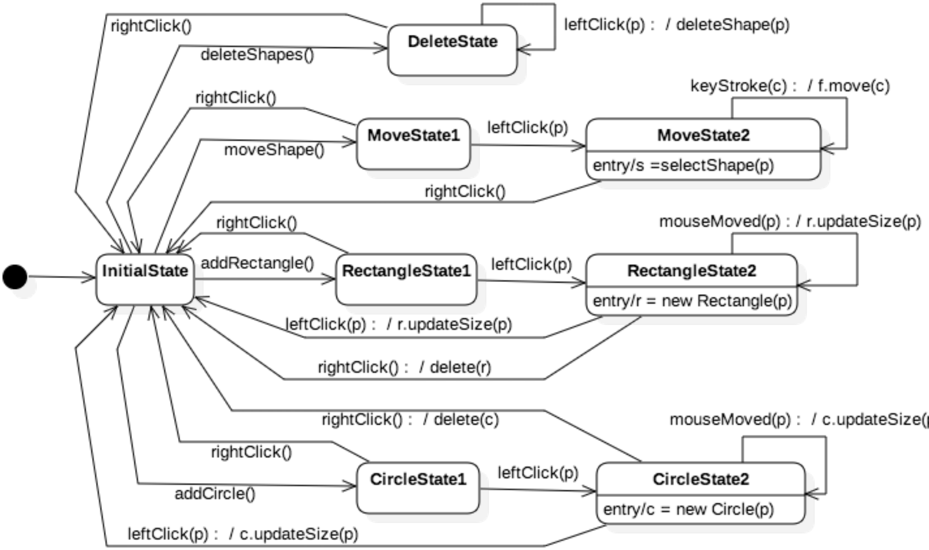
StateChart Diagram for “Add a rectangle”

1. The user clicks on the button "Add a rectangle"
2. The system asks to enter the coordinates of a first corner
3. The user clicks on a point p
4. The system creates a small rectangle r at point p and visualizes it
5. The user moves the mouse to another point p
6. The system updates the size of r wrt p
7. The user clicks on another point p
8. The system updates the size of r wrt p and returns to the initial state

Extension [1-8a]: The user cancels the action with a right click



StateChart Diagram of PlaCo



Each transition event corresponds to a method of *Controller*

GoF Pattern: State (1/3)

Problem:

The behaviour of *Controller* when receiving *leftClick(p)* depends on its state

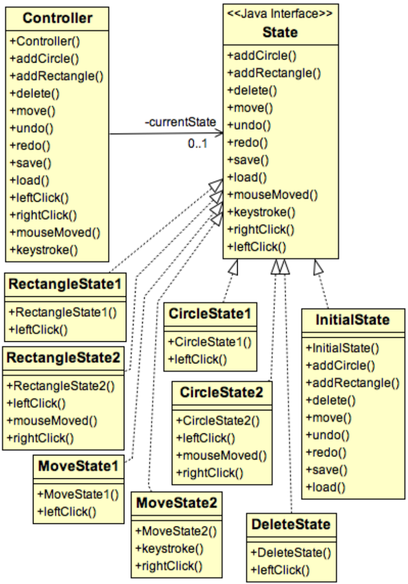
Solution 1:

- *Controller* has an attribute *currentState* to memorise its state
  - When launching PlaCo, *currentState* is set to *INITIAL_STATE*
  - When events occur, *currentState* is updated according to the Statechart Diagram
- *leftClick(p)* contains a case for each possible state:
  - If *currentState = INITIAL_STATE* then ignore left clicks
  - If *currentState = CIRCLE_STATE1* then create a new circle and set *currentState* to *CIRCLE_STATE2*
  - ...etc...

Pros and Cons?

Solution 2: Use the State Pattern

GoF Pattern: State (2/3)



Controller delegates actions to *currentState*:

```
public void leftClick(Point p) {
    currentState.leftClick(this, window, plan, p);
}
```

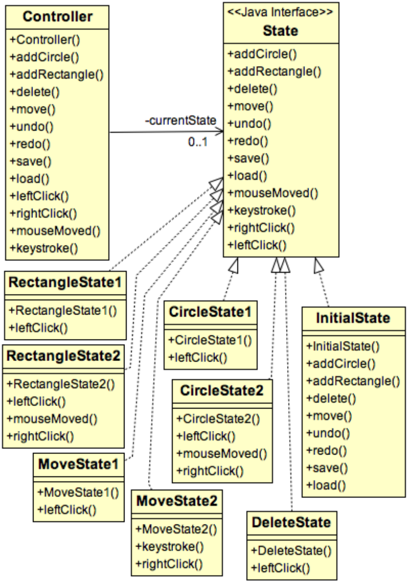
State defines default actions:

```
public interface State {
    public default void addCircle(Controller c, Window w){};
    public default void addRectangle(Controller c, Window w
    public default void delete(Controller c, Window w){};
    public default void move(Controller c, Window w){};
    public default void undo(){};
    public default void redo(){};
    public default void save(Plan p, Window w){};
    public default void load(Plan p, Window w){};
    public default void mouseMoved(Plan plan, Point p){};
    public default void keystroke(Plan p, int charCode){};
    public default void leftClick(Controller c, Window w, Po
    public default void rightClick(Controller c, Window w){
        w.allow(true);
        c.setCurrentState(c.initialState);
        w.displayMessage("");
    }
}
```

How to define method signatures?

Parameters = all objects needed to achieve actions

GoF Pattern: State (2/3)



Each class that implements *State* overrides some methods according to the Statechart Diagram

How does *Controller* change its state?

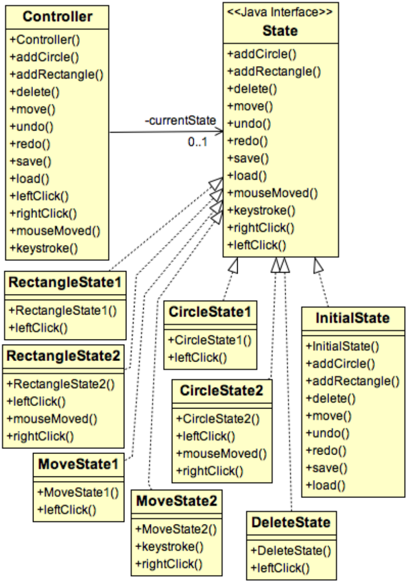
Protected method *setCurrentState* in *Controller*

How do we get *State* instances?

- Solution 1: Create a new instance for each state change
- Solution 2: Use Singletons (see later)
- Solution 3: *Controller* has a protected attribute for each state

Pros and Cons?

GoF Pattern: State (2/3)



Each class that implements *State* overrides some methods according to the Statechart Diagram

How does *Controller* change its state?

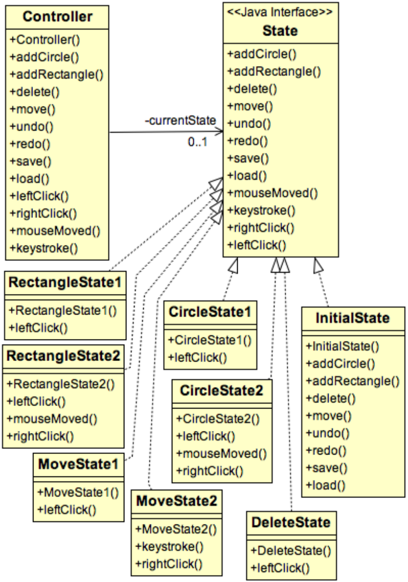
Protected method *setCurrentState* in *Controller*

How do we get *State* instances?

- Solution 1: Create a new instance for each state change
- Solution 2: Use Singletons (see later)
- Solution 3: *Controller* has a protected attribute for each state

Pros and Cons?

GoF Pattern: State (2/3)



Each class that implements *State* overrides some methods according to the Statechart Diagram

How does *Controller* change its state?

Protected method *setCurrentState* in *Controller*

How do we get *State* instances?

- Solution 1: Create a new instance for each state change
- Solution 2: Use Singletons (see later)
- Solution 3: *Controller* has a protected attribute for each state

Pros and Cons?

Code of the *leftClick* method

- In *Controller*:

```
public void leftClick(Point p) {
    currentState.leftClick(this, window, plan, p);
}
```

- In *State*:

```
public default void leftClick(Controller c, Window w, Plan plan, Point p){};
```

- In *CircleState1*:

```
public void leftClick(Controller c, Window w, Plan pl, Point pt) {
    if (pl.search(pt) != null){
        w.displayMessage("...error message...");
    } else {
        c.circleState2.entryAction(pt);
        c.setCurrentState(c.circleState2);
    }
}
```

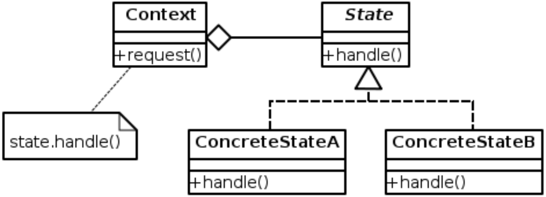
- In *CircleState2*:

```
public void leftClick(Controller c, Window w, Plan pl, Point pt) {
    circle.updateRadius(pt, pl);
    c.setCurrentState(c.initialState);
}
protected void entryAction(Point p) {
    circle = new Circle(p, 1);
}
```

GoF Pattern: State (3/3)

Generic solution:

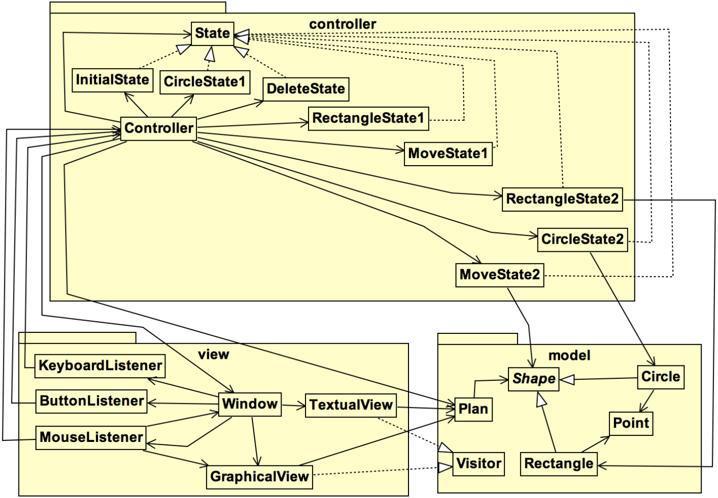
[Wikipedia]



Implemented principles:

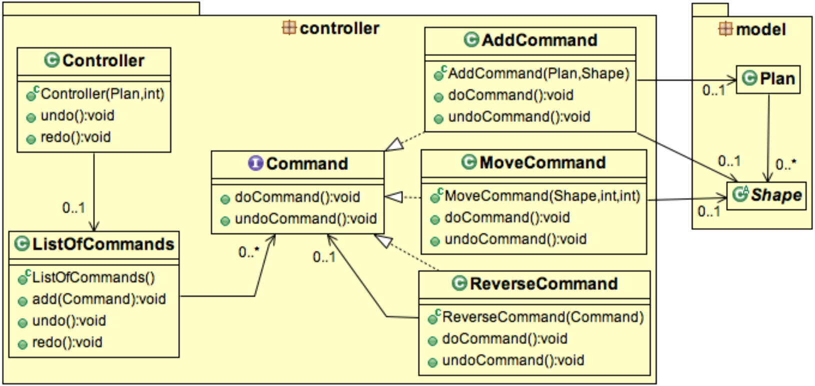
- High cohesion: Each *ConcreteState* contains all methods of events that have an effect on it
- Protected variations: Adding a new state is easy (but adding a new event is more tedious)
- Programming for interfaces

Current Architecture of PlaCo



Problem: How can we implement undo/redo?

GoF Pattern: Command (1/2)



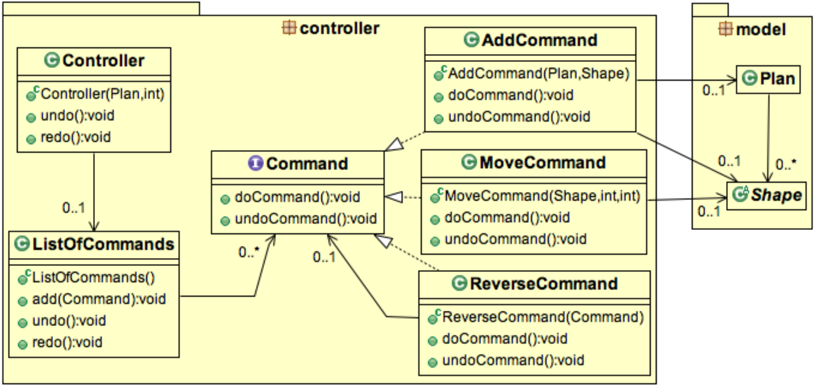
Controller:

```
public class Controller{
    private ListOfCommands l;
    public void undo(){
        currentState.undo(l);
    }
    public void redo(){
        currentState.redo(l);
    }
```

InitialState:

```
public class InitialState implements State {
    public void undo(ListOfCommands l){
        l.undo();
    }
    public void redo(ListOfCommands l){
        l.redo();
    }
```

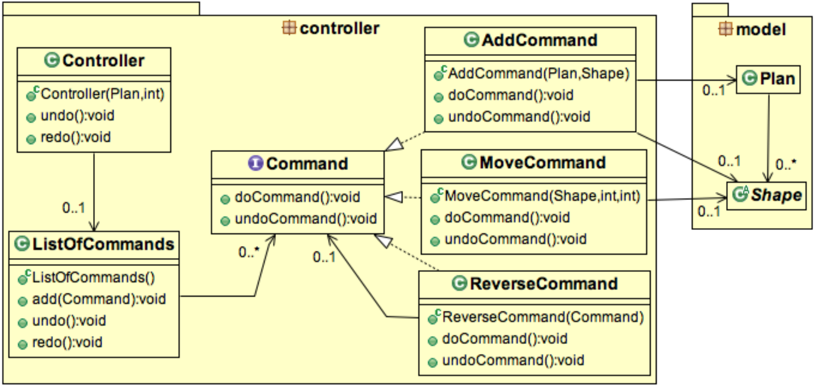
GoF Pattern: Command (1/2)



ListOfCommands:

```
public class ListOfCommands {
    private LinkedList<Command> l;
    private int i;
    public ListOfCommands(){i = -1; l = new LinkedList<Command>();}
    public void add(Command c){i++; l.add(i, c); c.doCommand();}
    public void undo(){if (i >= 0){l.get(i).undoCommand(); i--;}}
    public void redo(){i++; l.get(i).doCommand();}
}
```

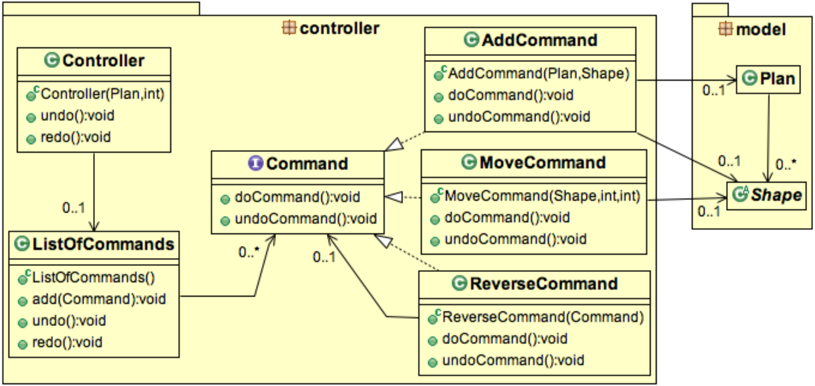
GoF Pattern: Command (1/2)



AddCommand:

```
public class AddCommand implements Command {
    private Plan plan;
    private Shape shape;
    public AddCommand(Plan p, Shape s){this.plan = p; this.shape = s;}
    public void doCommand() {plan.add(shape);}
    public void undoCommand() {plan.remove(shape);}
}
```

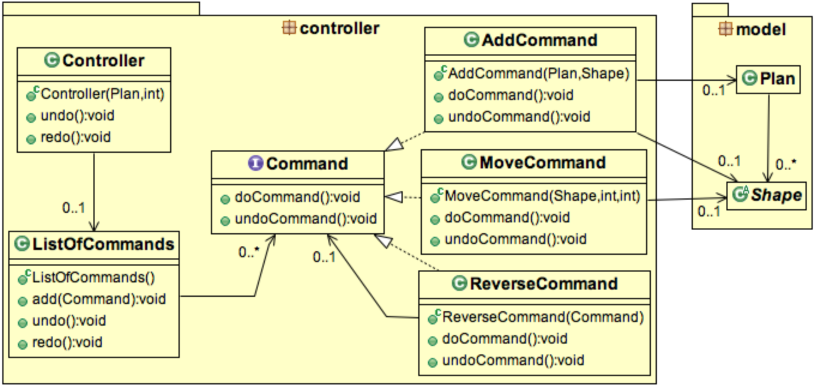
GoF Pattern: Command (1/2)



ReverseCommand:

```
public class ReverseCommand implements Command{
    private Command cmd;
    public ReverseCommand(Command cmd){this.cmd = cmd;}
    public void doCommand() {cmd.undoCommand();}
    public void undoCommand() {cmd.doCommand();}
}
```

GoF Pattern: Command (1/2)



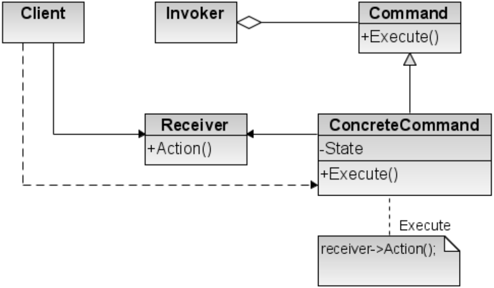
DeleteState:

```
public class DeleteState implements State {
    public void leftClick(Controller c, Window w, Plan pl,
            ListOfCommands l, Point pt) {
        Shape s = pl.search(pt);
        if (s != null) l.add(new ReverseCommand(new AddCommand(pl, s)));
    }
}
```

GoF Pattern: Command (2/2)

Generic Solution:

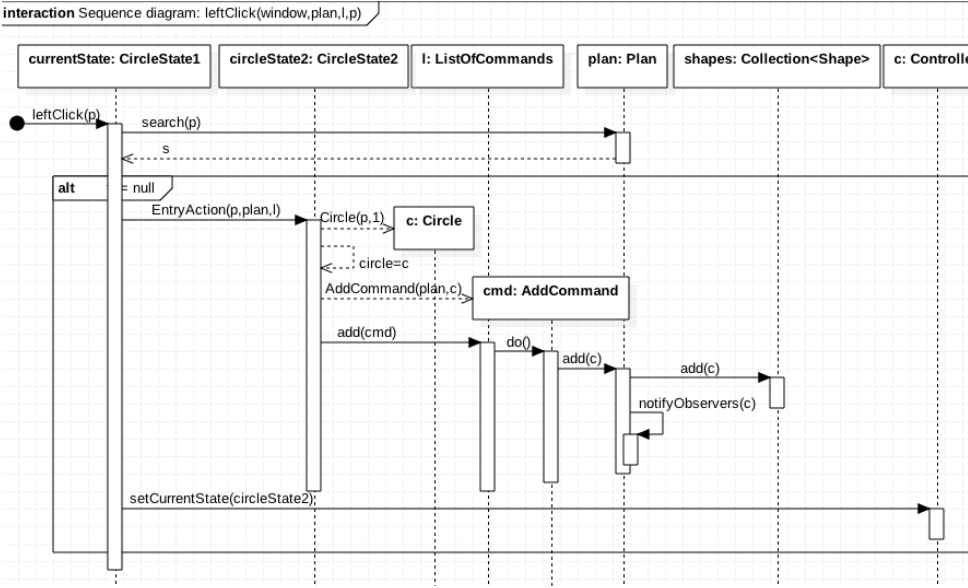
- *Client* creates instances of *ConcreteCommand*
- *Invoker* asks for commands to be executed
- *ConcreteCommande* delegates the execution to *Receiver*



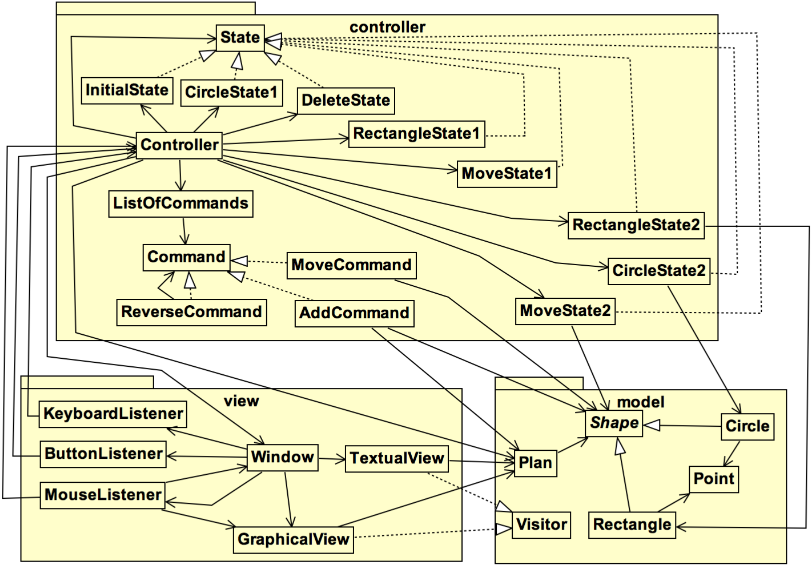
Remarks:

- The reception of a request is separated from its execution
- The roles of *Client* and *Invoker* may be played by a same class
- May be used to undo or redo some commands after a failure

Sequence Diagram



Current Architecture of PlaCo

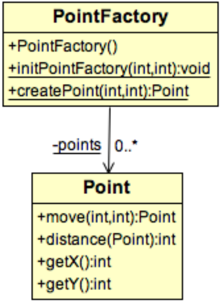


Problem: Numerous instances of *Point* are created

Patterns GoF: FlyWeight and Factory

Solution:

- A same instance is shared for all points with the same coordinates
  ⤳ Warning: the instance must be changed when moving a point!
- A factory is used to create instances

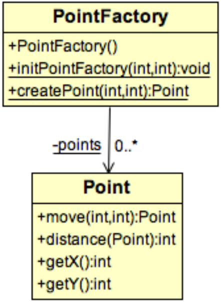



```
public class PointFactory {
    private static Point points[][];
    private static int width;
    private static int height;
    public static void initPointFactory(int width, int height){
        PointFactory.width = width;
        PointFactory.height = height;
        PointFactory.points = new Point[width+1][height+1];
    }
    public static Point createPoint(int x, int y){
        if ((x > width) || (x < 0) || (y > height) || (y < 0))
            return null;
        if (points[x][y] == null)
            points[x][y] = new Point(x,y);
        return points[x][y];
    }
}
```

Patterns GoF: FlyWeight and Factory

Solution:

- A same instance is shared for all points with the same coordinates
  $\leadsto$ Warning: the instance must be changed when moving a point!
- A factory is used to create instances

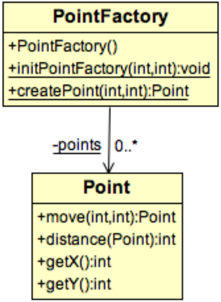



```java
public class Point {
    private int x;
    private int y;
    protected Point(int x, int y){
        this.x = x; this.y = y;
    }
    public Point move(int deltaX, int deltaY) {
        return PointFactory.createPoint(x+deltaX, y+deltaY);
    }
    public int distance(Point p){
        return (int)(Math.sqrt((x-p.getX())*(x-p.getX())
                + (y-p.getY())*(y-p.getY())));
    }
    public int getX() {return x;}
    public int getY() {return y;}
}
```

Patterns GoF: FlyWeight and Factory

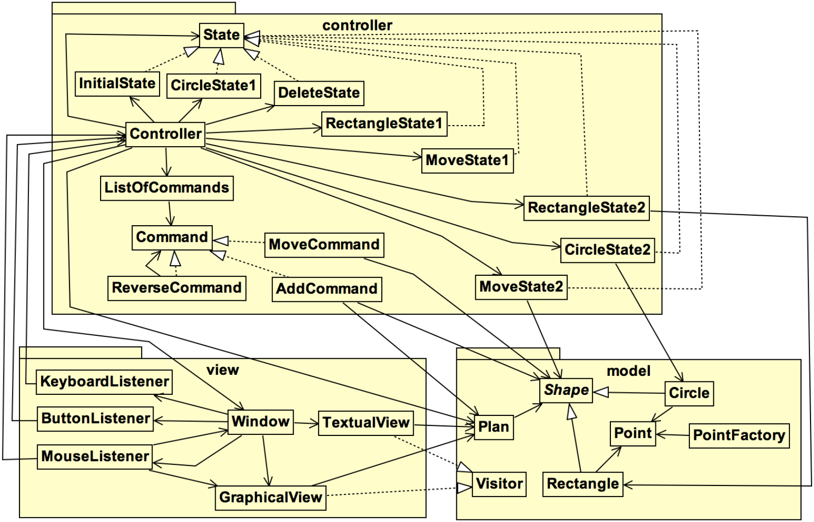
Solution:

- A same instance is shared for all points with the same coordinates
  $\leadsto$ Warning: the instance must be changed when moving a point!
- A factory is used to create instances



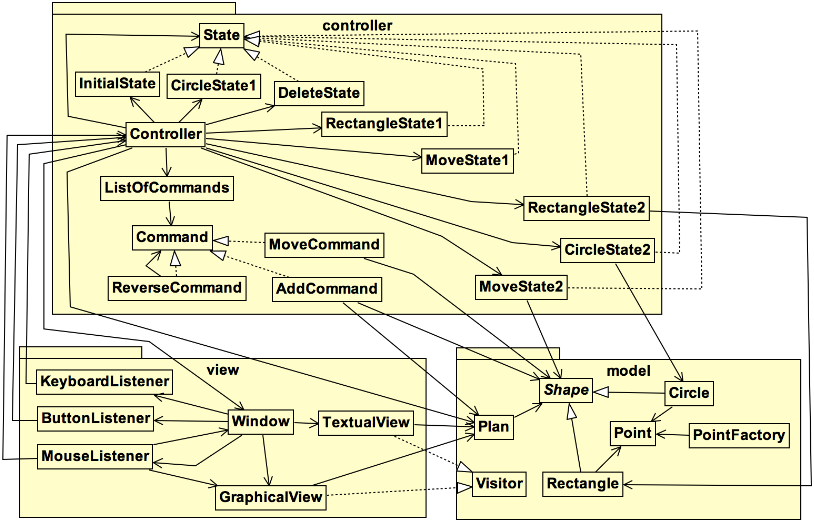

```java
public class MouseListener extends MouseAdapter {
    public void mouseMoved(MouseEvent evt) {
        Point p = coordinates(evt);
        if (p != null) controller.mouseMoved(p);
    }
    private Point coordinates(MouseEvent evt){
        MouseEvent e = SwingUtilities.convertMouseEvent(w, evt, g);
        int x = Math.round((float)e.getX()/(float)g.getScale());
        int y = Math.round((float)e.getY()/(float)g.getScale());
        return PointFactory.createPoint(x, y);
    }
```

Current Architecture of PlaCo



Some Use Cases are still missing!

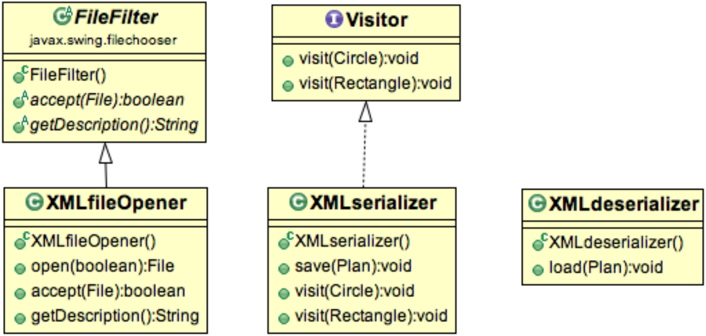
Current Architecture of PlaCo



Some Use Cases are still missing!

⇝ Load/Save a plan from/to an XML file

Class Diagram of the *xml* package



How to send messages to *XMLfileOpener* from any class of *xml*?

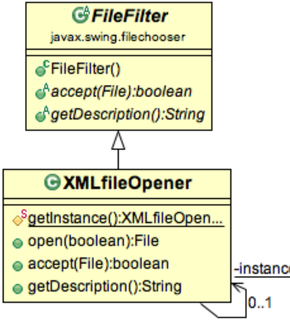
- Transform methods of *XMLfileOpener* into static methods?
  ⤳ Not possible if *XMLfileOpener* extends *fileFilter*

```
public class XMLfileOpener extends FileFilter {
    public File open(boolean read) throws ExceptionXML{
        JFileChooser jFileChooserXML = new JFileChooser();
        jFileChooserXML.setFileFilter(this);
```

- Use a Singleton

GoF Pattern: Singleton

```
public class XMLfileOpener extends FileFilter {
    private static XMLfileOpener instance = null;
    private XMLfileOpener(){}
    protected static XMLfileOpener getInstance(){
        if (instance == null) instance = new XMLfileOpener();
        return instance;
    }
    public File open(boolean read) throws ExceptionXML{
    public boolean accept(File f) {
    public String getDescription() {
    private String getExtension(File f) {
}
```



XMLfileOpener can have only one instance, and this instance is visible by all classes of the package
$\rightsquigarrow$ *XMLfileOpener.getInstance()*

Warning:

May be considered as an anti-pattern... To be used with moderation!

Overview

23 Patterns of the Gang of Four (GoF)

[E. Gamma, R. Helm, R. Johnson, J. Vlissides]

Patterns illustrated with PlaCo:

- Creation: Factory, Singleton
- Behaviour: Iterator, State, Observer, Command, Visitor
- Structure: FlyWeight

Patterns introduced in the next slides:

- Creation: Abstract factory
- Behaviour : Strategy
- Structure: Decorator, Adaptator, Facade, Composite

Pattern introduced for the project:

- Behaviour: Template

Patterns that won't be studied in this course:

- Creation: Prototype, Builder
- Behaviour: Chain of resp., Interpretor, Mediator, Memento
- Structure: Bridge, Proxy

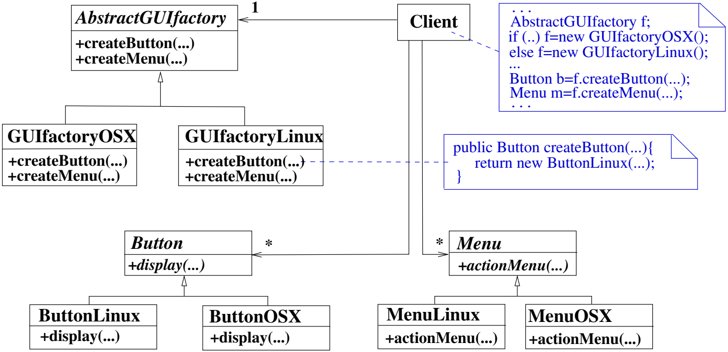
GoF Pattern: Abstract factory (1/2)

Problem:

Create a family of objects without specifying their concrete classes

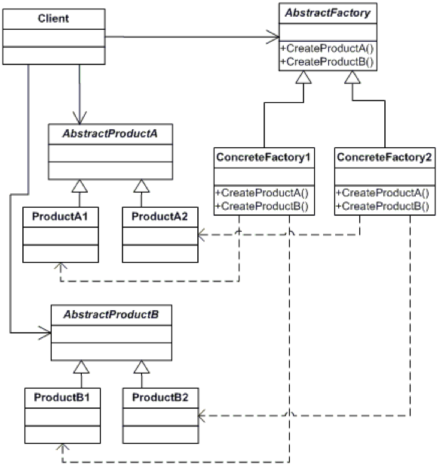
Illustration on an example:

- Create a GUI with widgets (buttons, menus, ...)
- Point of variation: OS (Linux or OSX)



GoF Pattern: Abstract factory(2/2)

Generic Solution [Wikipedia]:



Remarks:

- AbstractFactory and AbstractProduct usually are interfaces
  $\rightsquigarrow$ Programming for interfaces
- *createProductX()* methods are factory methods

Advantages of the pattern:

- Indirection: Isolate Client from product implementations
- Protected variations: Make it easy to change product families
- Consistency is automatically ensured

But adding new products is more tedious

GoF Pattern: Strategy (1/3)

Problem:

How to dynamically change the behaviour of an object?

Illustration on an example:

- In a video game, characters fight monsters...
  $\leadsto$ method *fight(Monster m)* of class *Character*
  ...and the code of *fight* may be different from a character to an other one
  - Sol. 1: *fight* contains a case for each kind of fight
  - Sol. 2: The class *Character* is specialised in subclasses that override *fight*
- Represent these solutions with UML. Can we easily:
  - Add a new kind of fight?
  - Change the kind of fight of a character?

GoF Pattern: Strategy (1/3)

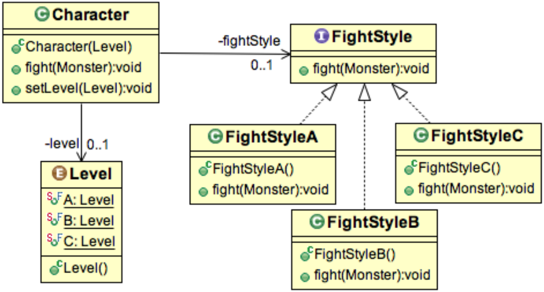
Problem:

How to dynamically change the behaviour of an object?

Illustration on an example:

- In a video game, characters fight monsters...
  $\rightsquigarrow$ method *fight(Monster m)* of class *Character*
  ...and the code of *fight* may be different from a character to an other one
  - Sol. 1: *fight* contains a case for each kind of fight
  - Sol. 2: The class *Character* is specialised in subclasses that override *fight*
  - Sol. 3: Strategy pattern = *Character* delegates fight to classes that encapsulate fight code and all realise a same interface
- Represent these solutions with UML. Can we easily:
  - Add a new kind of fight?
  - Change the kind of fight of a character?

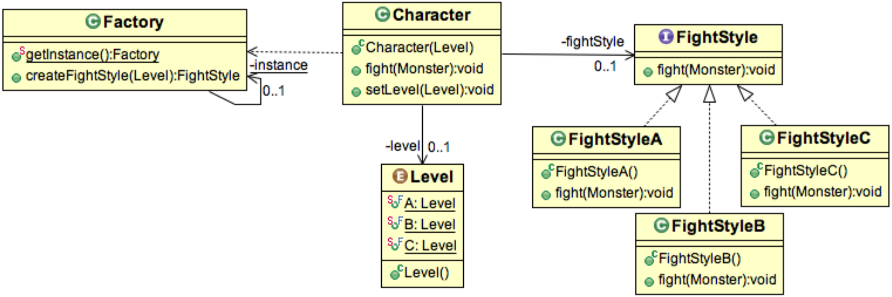
GoF Pattern: Strategy (2/3)




```
public class Character{
    private Level level;
    private FightStyle fightStyle;
    public Character(Level l){
        level = l;
        fightStyle =
    }
    public void fight(Monster m){
        fightStyle.fight(m);
    }
    public void setLevel(Level l){
        level = l;
        fightStyle =
    }
    // ... other methods of Character
}
```

How to create the instance of FightStyle corresponding to level?

GoF Pattern: Strategy (2/3)

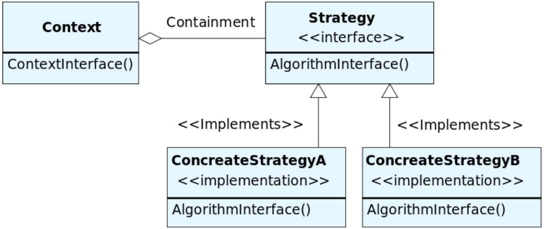



```java
public class Character{
    private Level level;
    private FightStyle fightStyle;
    public Character(Level l){
        level = l;
        fightStyle = Factory.getInstance().createFightStyle(l);
    }
    public void fight(Monster m){
        fightStyle.fight(m);
    }
    public void setLevel(Level l){
        level = l;
        fightStyle = Factory.getInstance().createFightStyle(l);
    }
    // ... other methods of Character
```

GoF Pattern: Strategy (3/3)

Generic Solution:

[Wikipedia]



Remarks:

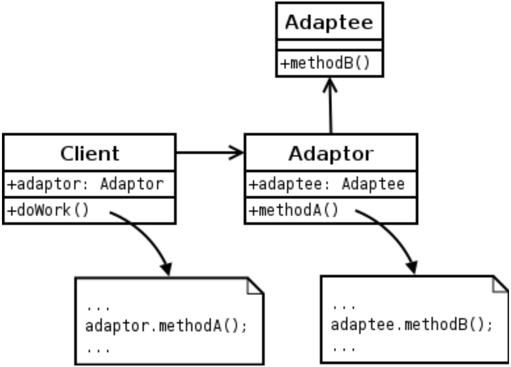
- Principles implemented:
  - Indirection: *Context* is isolated from *Strategy* implementations $\leadsto$ Protected variations
  - Compose rather than inherit to dynamically change strategies
- How to transfer information from *Context* to *Strategy*?
  - Push: Use parameters of *AlgorithmInterface()*
  - Pull: Use getters of the context (and pass the context as a parameter of *AlgorithmInterface()*

GoF Pattern: Adapter

Problem:

How to provide a stable interface (Adaptor) to a component whose interface may change (Adaptee)

Generic Solution [Wikipedia]:



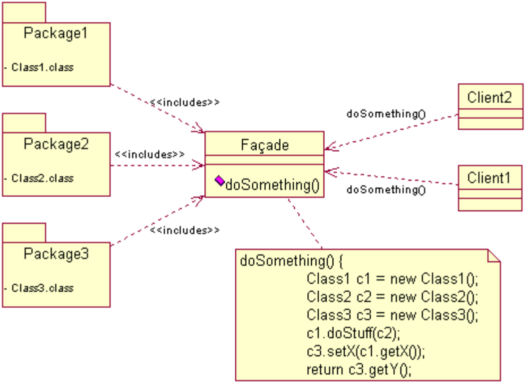
Principles implemented = indirection and protected variations

GoF Pattern: Facade

Problem:

Provide a simplified interface (Facade)

Generic Solution [Wikipedia]:



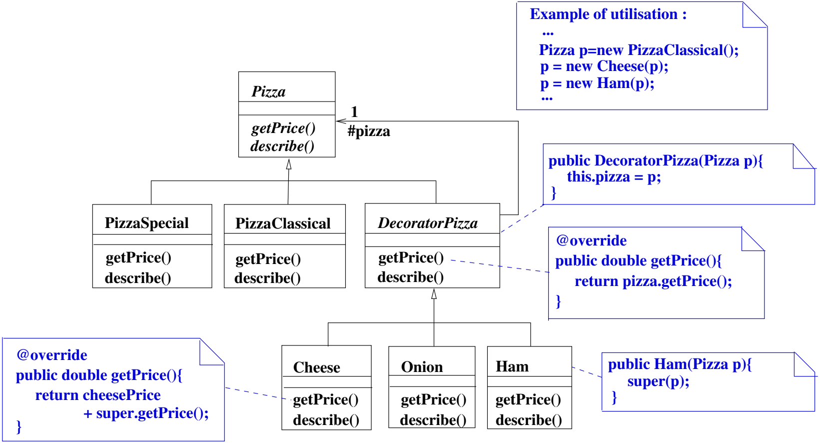
Principles implemented = indirection and protected variations

GoF Pattern: Decorator (1/2)

Problème:

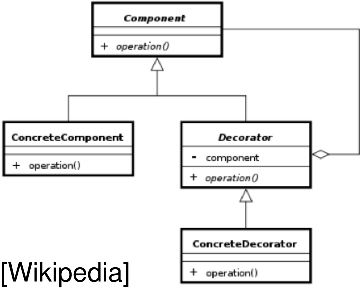
Dynamically add new responsibilities to an object

Illustration on an example:



GoF Pattern: Decorator (2/2)

Generic Solution:



[Wikipedia]

Remarks:

- Compose rather than inherit: Dynamically add responsibilities to ConcreteComponent without modifying its code
- n decors $\Rightarrow 2^n$ combinations
- **Drawback**: May generate a lot of wrapper objects

Utilisation for extending input/output Java Classes:

- Component: InputStream, OutputStream
- ConcreteComponent: FileInputStream, ByteArrayInputStream, ...
- Decorator: FilterInputStream, FilterOutputStream
- ConcreteDecorator: BufferedInputStream, CheckedInputStream, ...

Adapter, Facade and Decorator

Common points:

- Indirection $\rightsquigarrow$ Wrapper
- Protected variations

Differences:

- Adapter: Convert an interface into an other one (needed by a Client)
- Facade: Provide a simplified interface
- Decorator: Dynamically add responsibilities to methods of a class without modifying its code

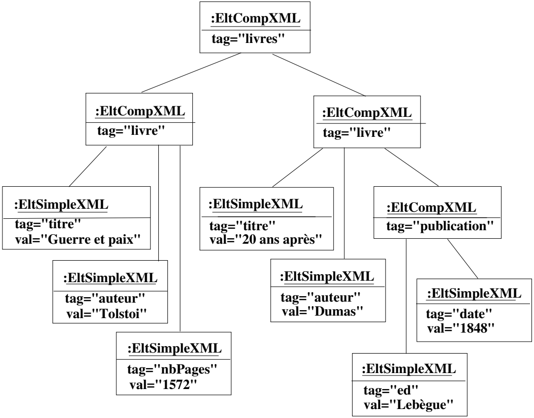
GoF Pattern: Composite (1/2)

Problème:

Represent hierarchies and uniformly treat component and compound objects

Illustration on an example:

```
<?xml version="1.0"?>
<livres>
    <livre>
        <titre>Guerre et paix</titre>
        <auteur>Tolstoï</auteur>
        <nbPages>1572</nbPages>
    </livre>
    <livre>
        <titre>20 ans après</titre>
        <auteur>Dumas</auteur>
        <publication>
            <ed>Lebègue</ed>
            <date>1848</date>
        </publication>
    </livre>
</livres>
```



How to count the number of tags?

GoF Pattern: Composite (2/2)

Generic Solution [Wikipedia]: