Design patterns I : Introduction

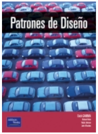
J.Serrat

Object-oriented programming, MatCAD

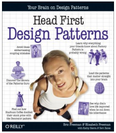
References

Books

Desing patterns: elements of reusable object oriented software. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison Wesley, 1994.



Patrones de diseno. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Pearson Educacion, 2003.



Head first design patterns. E. Freeman, E. Freeman, K. Sierra, B. Bates. O'Reilly, 2004.

References

Web sites

http://sourcemaking.com

https://en.wikipedia.org/wiki/Software_design_pattern

Index

1. Concept
2. A first pattern: *composite*
3. GoF patterns
4. Why learn design patterns?

Concept

Design pattern

Experience in object oriented design has shown that some problems appear recurrently. A design pattern

- is the proper definition of one of such problems,
- is a general, reusable, 'good' solution
- the analysis of its pros and cons

Concept

The solution

- is *not* a finished design that can be transformed directly into code
- is a description and a template for how to solve the problem
- therefore, can be applied in many different situations
- is expressed in terms of classes, interfaces, methods and their collaborations
- good = most general, complete, flexible, consensus from many experts, because it follows the object-oriented principles

A first pattern: *composite*.

A first pattern: *composite*

Remember what can we do with typical graphical editors ?

A first pattern: *composite*

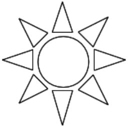
Remember what can we do with typical graphical editors ?

A first pattern: *composite*

Remember what can we do with typical graphical editors ?

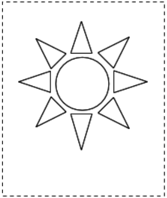
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



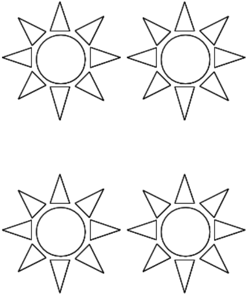
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



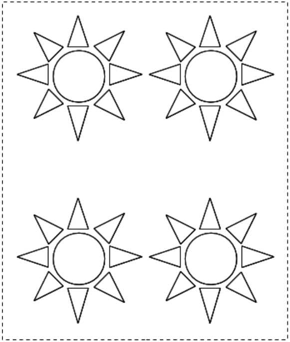
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



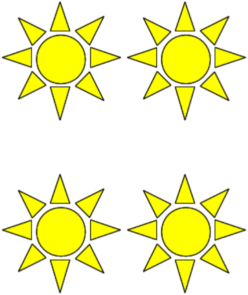
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



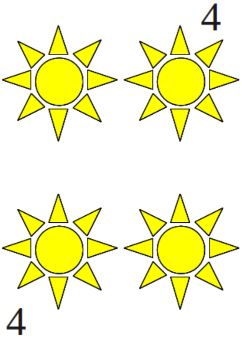
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



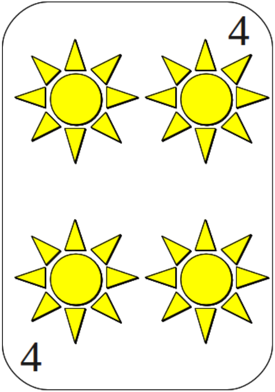
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



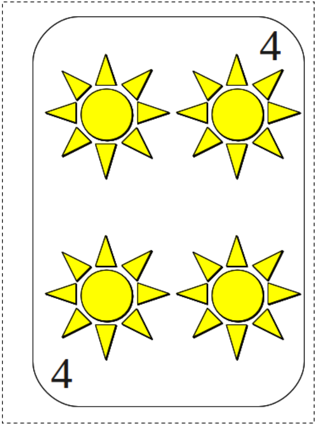
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



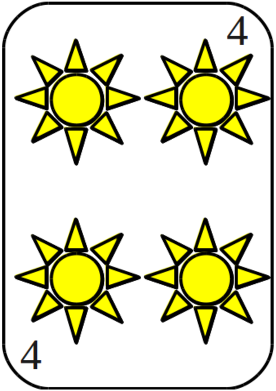
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



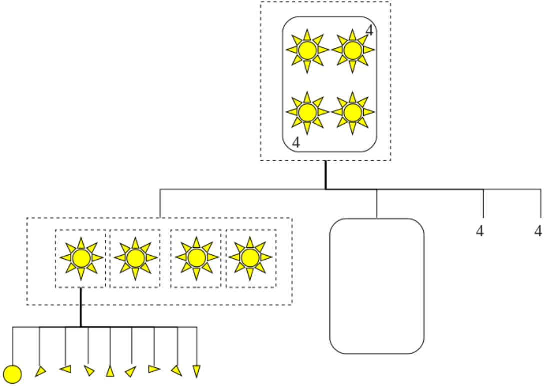
A first pattern: *composite*

Remember what can we do with typical graphical editors ?



A first pattern: *composite*

- we have grouped several objects into one so that are treated as a single object when we move, copy, change color or line width
- there is an unlimited hierarchy of groupings



A first pattern: *composite*

Which classes support these requirements ? Define

- classes for graphical primitives such as Text, Line, Circle, Triangle, plus
- a class that act as container for these primitives

A first pattern: *composite*

Which classes support these requirements ? Define

- classes for graphical primitives such as Text, Line, Circle, Triangle, plus
- a class that act as container for these primitives

All of them can (know how to) move, scale, rotate and draw themselves. These methods are different, however: rotate line, triangle, square . . . means to rotate its vertices, to rotate a circle do nothing.

A first pattern: *composite*

Which classes support these requirements ? Define

- classes for graphical primitives such as Text, Line, Circle, Triangle, plus
- a class that act as container for these primitives

All of them can (know how to) move, scale, rotate and draw themselves. These methods are different, however: rotate line, triangle, square . . . means to rotate its vertices, to rotate a circle do nothing.

Other methods are common: `setColor(Color c)`, `setLineWidth(int w)` just change the value of an attribute.

A first pattern: *composite*

Which classes support these requirements ? Define

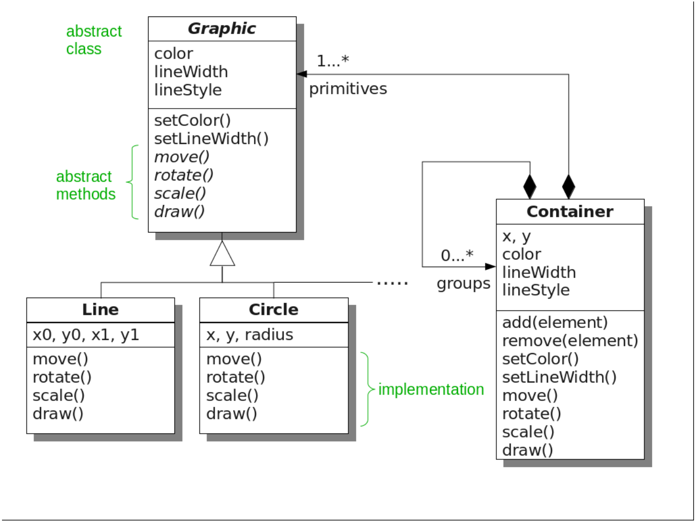
- classes for graphical primitives such as Text, Line, Circle, Triangle, plus
- a class that act as container for these primitives

All of them can (know how to) move, scale, rotate and draw themselves. These methods are different, however: rotate line, triangle, square . . . means to rotate its vertices, to rotate a circle do nothing.

Other methods are common: `setColor(Color c)`, `setLineWidth(int w)` just change the value of an attribute.

Program to a interface, not to an implementation

A first pattern: *composite*



A first pattern: *composite*

Client code must differentiate primitive from container objects, even if most of the time makes the same things, making the application more complex.

```
void print(Object ob) {
    if (ob instanceof Graphic) {
        ((Graphic) ob).printIt();
    } else if (ob instanceof Container) {
        ((Container) ob).printIt();
    }
}
```

A first pattern: *composite*

```
// make some shapes
Triangle t1 = new Triangle();
Triangle t2 = new Triangle();
Circle c1 = new Circle();
Graphic g[] = new Graphics[3]; //vector of primitives
g[0] = t1; g[1] = t2; g[2] = c1;
for (int i=0 ; i<g.length ; i++) { g.setColor(10); }
// make new shapes and group them
Triangle t3 = new Triangle();
Circle c2 = new Circle();
Container con = new Container();
con.add(t3);
con.add(c2);
con.setColor(10);
// Can not change again the color to all of them with
// a single method call: con is not a Graphic object
```

A first pattern: *composite*

Solution:

- make `Container` inherit from `Graphic`

A first pattern: *composite*

Solution:

- make `Container` inherit from `Graphic`
- `Graphic` is an abstract super class of both primitives and groups: the common interface to deal with any kind of graphics

A first pattern: *composite*

Solution:

- make `Container` inherit from `Graphic`
- `Graphic` is an abstract super class of both primitives and groups: the common interface to deal with any kind of graphics
- recursive composition : a container may contain primitives and/or other containers

A first pattern: *composite*

Solution:

- make `Container` inherit from `Graphic`
- `Graphic` is an abstract super class of both primitives and groups: the common interface to deal with any kind of graphics
- recursive composition : a container may contain primitives and/or other containers

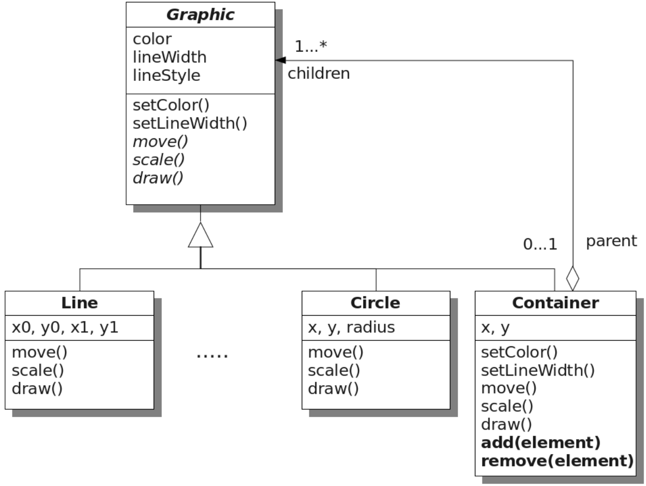
A first pattern: *composite*

Solution:

- make `Container` inherit from `Graphic`
- `Graphic` is an abstract super class of both primitives and groups: the common interface to deal with any kind of graphics
- recursive composition : a container may contain primitives and/or other containers

```
void print(Graphic ob) {
    ob.printIt();
}
```

A first pattern: *composite*

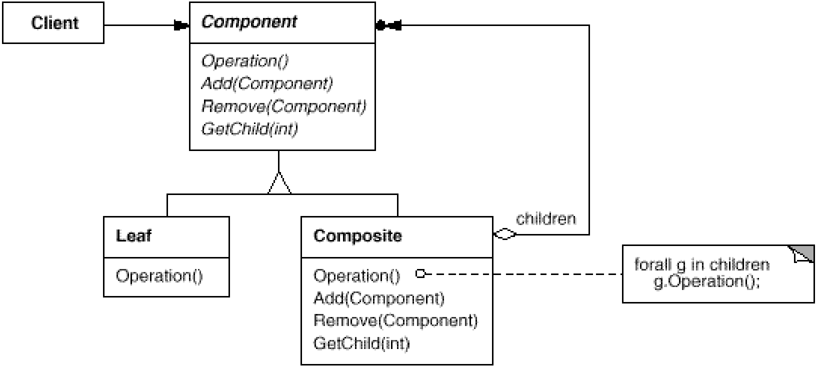


A first pattern: *composite*

```
// make some shapes
Triangle t1 = new Triangle();
Triangle t2 = new Triangle();
Circle c1 = new Circle();
// make new shapes and group them
Triangle t3 = new Triangle();
Circle c2 = new Circle();
Container con = new Container();
con.add(t3);
con.add(c2);
// make a group for the whole drawing
Graphic drawing = new Container();
drawing.add(t1); drawing.add(t2); drawing.add(con);
// change color of all drawring with a single call
drawing.setColor(10);
```

A first pattern: *composite*

Template of the composite pattern



A first pattern: *composite*

A second example: logic circuit design and simulation:

- a program must allow to design digital circuits based on logical gates

We'll come back to this problem and solve it in detail (with source code).

A first pattern: *composite*

A second example: logic circuit design and simulation:

- a program must allow to design digital circuits based on logical gates

- and simulate it (compute the output given a certain input)

We'll come back to this problem and solve it in detail (with source code).

A first pattern: *composite*

A second example: logic circuit design and simulation:

- a program must allow to design digital circuits based on logical gates

- and simulate it (compute the output given a certain input)
- obviously, we want to be able to build complex circuits by connecting simpler sub-circuits and/or logical gates

We'll come back to this problem and solve it in detail (with source code).

A first pattern: *composite*

A second example: logic circuit design and simulation:

- a program must allow to design digital circuits based on logical gates

- and simulate it (compute the output given a certain input)
- obviously, we want to be able to build complex circuits by connecting simpler sub-circuits and/or logical gates
- there's no constraint on the level of nesting circuits and gates

We'll come back to this problem and solve it in detail (with source code).

A first pattern: *composite*

A second example: logic circuit design and simulation:

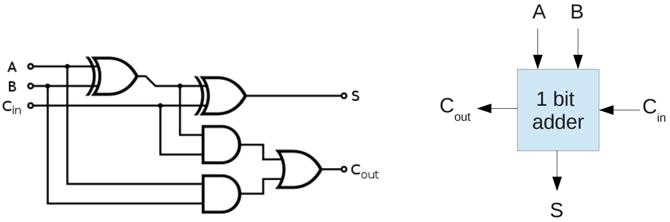
- a program must allow to design digital circuits based on logical gates

- and simulate it (compute the output given a certain input)
- obviously, we want to be able to build complex circuits by connecting simpler sub-circuits and/or logical gates
- there's no constraint on the level of nesting circuits and gates
- any logical gate and circuit can be simulated through a `process()` method

We'll come back to this problem and solve it in detail (with source code).

A first pattern: *composite*

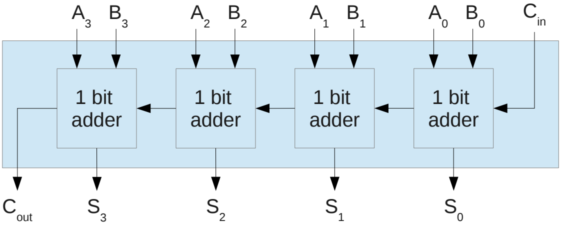
A second example: logic circuit design and simulation



One bit adder with carry

A first pattern: *composite*

A second example: logic circuit design and simulation



Four bits adder with carry, made of four one bit adders

A first pattern: *composite*

A third example : a simple maze game, from Gamma et al. book (introduction to creational patterns).

- build a maze for a computer game, where the player has to find the way out

A first pattern: *composite*

A third example : a simple maze game, from Gamma et al. book (introduction to creational patterns).

- build a maze for a computer game, where the player has to find the way out
- focus on how mazes get created, not on players, graphics etc.

A first pattern: *composite*

A third example : a simple maze game, from Gamma et al. book (introduction to creational patterns).

- build a maze for a computer game, where the player has to find the way out
- focus on how mazes get created, not on players, graphics etc.
- useful as starting point to introduce other patterns later

A first pattern: *composite*

A third example : a simple maze game, from Gamma et al. book (introduction to creational patterns).

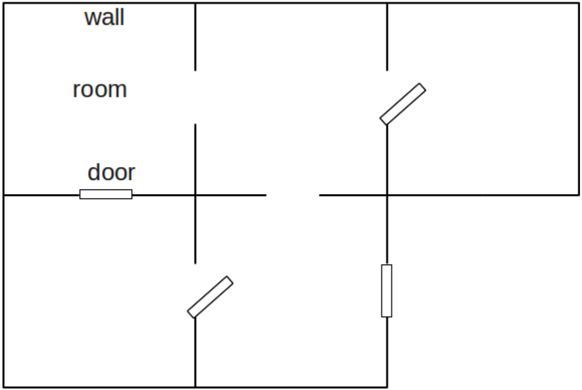
- build a maze for a computer game, where the player has to find the way out
- focus on how mazes get created, not on players, graphics etc.
- useful as starting point to introduce other patterns later
- maze is made of rooms, a room knows its neighbours: another room, a wall, or a door to another room

A first pattern: *composite*

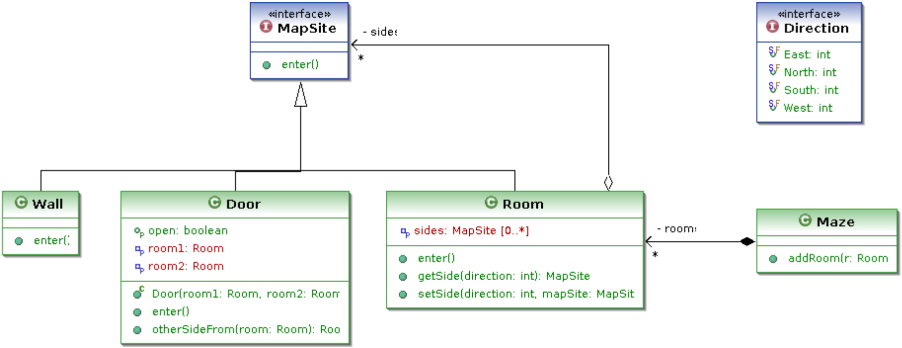
A third example : a simple maze game, from Gamma et al. book (introduction to creational patterns).

- build a maze for a computer game, where the player has to find the way out
- focus on how mazes get created, not on players, graphics etc.
- useful as starting point to introduce other patterns later
- maze is made of rooms, a room knows its neighbours: another room, a wall, or a door to another room
- you can enter a room, door (changes room if open) and even a wall (if given the power to do so, change room)

A first pattern: *composite*



A first pattern: *composite*



Exercise: write the Java code to create the former maze in the `main` method of a `Client` class.

A first pattern: *composite*

Eric Gamma and colleagues published in 1995 the influential book *Design patterns: Elements of Reusable Object-Oriented Software*. Has a catalogue of 23 patterns. For each one, a template is followed:

- Name
- Intent : what it does and advantages $1-2$ sentences
- Motivation : example
- Structure : template class diagram
- Applicability : when to use it
- Consequences : advantages and shortcomings
- Implementation discussion, C++ sample code

A first pattern: *composite*

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

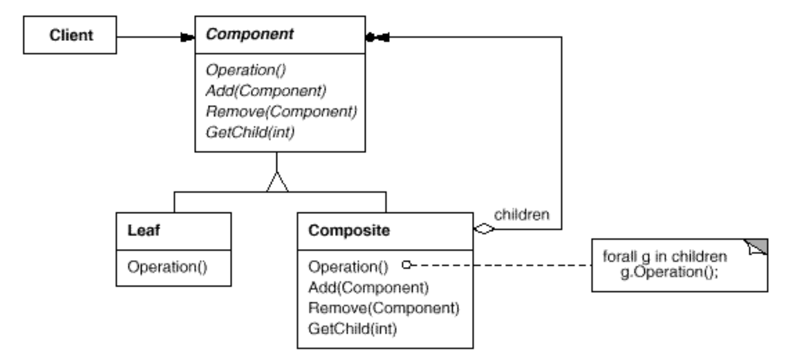
Applicability

Use the Composite pattern when you want

- represent part-whole hierarchies of objects
- want clients to ignore the difference between compositions of objects and individual objects

A first pattern: *composite*

Structure



A first pattern: *composite*

Consequences

- primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively

A first pattern: *composite*

Consequences

- primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively
- makes the client simpler : don't need to know whether they're dealing with a leaf or a composite component

A first pattern: *composite*

Consequences

- primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively
- makes the client simpler : don't need to know whether they're dealing with a leaf or a composite component
- wherever client code expects a primitive object, it can also take a composite object

A first pattern: *composite*

Consequences

- primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively
- makes the client simpler : don't need to know whether they're dealing with a leaf or a composite component
- wherever client code expects a primitive object, it can also take a composite object
- makes it easier to add new kinds of components: clients don't have to be changed

A first pattern: *composite*

Consequences

- primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively
- makes the client simpler : don't need to know whether they're dealing with a leaf or a composite component
- wherever client code expects a primitive object, it can also take a composite object
- makes it easier to add new kinds of components: clients don't have to be changed
- can make your design overly general: sometimes you want a composite to have only certain types of components but you can not restrict the components of a composite.

A first pattern: *composite*

Underlying OO principles:

Program to an interface, not to an implementation

The declared types of variables and parameters should be a supertype (abstract class or interface in Java) so that the referenced objects can be any concrete implementation (non-abstract subclass) of it.

Encapsulate what varies

Identify the aspects of your application that vary and encapsulate (separate, isolate) them, so that they can change without affecting those that don't change.

GoF patterns

Gamma et al. classify patterns into 3 groups:

- **Creational** patterns concern the process of object creation
- **Structural** patterns deal with the composition of classes or objects
- **Behavioral** patterns characterize the ways in which classes or objects interact and distribute responsibilities

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

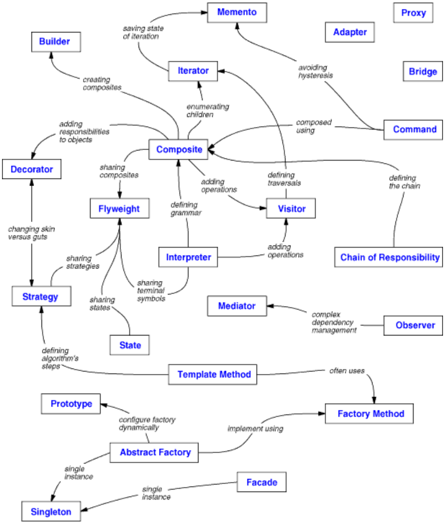
GoF patterns

Design patterns are related in different ways :

- combined, used together: Composite is often used with Iterator or Visitor
- alternatives: Prototype is often an alternative to Abstract Factory
- have similar structure but different intent: Composite and Decorator

GoF patterns

Pattern relationships



Why patterns ?

Why learn design patterns ?

- do not reinvent the wheel, reuse good solutions

Why patterns ?

Why learn design patterns ?

- do not reinvent the wheel, reuse good solutions
- "learn from others' successes instead of your own failures"

Why patterns ?

Why learn design patterns ?

- do not reinvent the wheel, reuse good solutions
- "learn from others' successes instead of your own failures"
- robust, flexible designs, prepared for change $\Rightarrow$ maintainable

Why patterns ?

Why learn design patterns ?

- do not reinvent the wheel, reuse good solutions
- “learn from others’ successes instead of your own failures”
- robust, flexible designs, prepared for change $\Rightarrow$ maintainable
- know common language: *composite*, *observer*, *strategy* . . . to communicate complex design ideas in a compact way

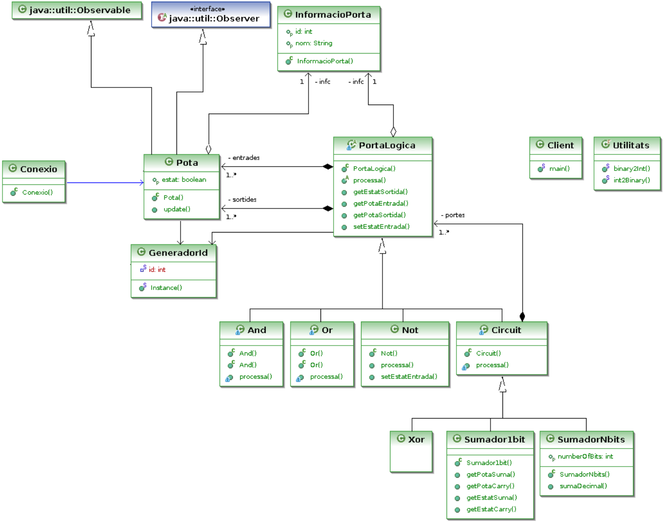
Why patterns ?

Why learn design patterns ?

- do not reinvent the wheel, reuse good solutions
- "learn from others' successes instead of your own failures"
- robust, flexible designs, prepared for change $\Rightarrow$ maintainable
- know common language: *composite*, *observer*, *strategy* . . . to communicate complex design ideas in a compact way
- high level view of the design

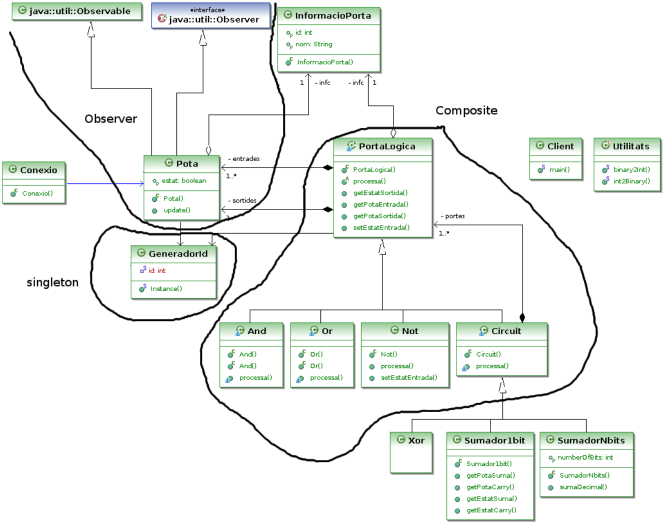
Why patterns ?

What's this ??



Why patterns ?

What's this ??



Why patterns ?

A way to apply object oriented design principles in your designs

- favour composition over inheritance (see *strategy*, *bridge*)
- strive for loosely coupled classes (see *observer*, *mediator*)
- program to an interface, not an implementation (see *composite*, *observer*)
- classes should be open for extension but closed for modification (see *decorator*, *chain of responsabability*)
- a class should have just one one reason to change
- identify the aspects that vary and separate them from those that stay the same

GoF patterns

Encapsulate what varies is a most important and recurrent principle of design patterns

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Why patterns ?

A novice chess player knows

A good player knows

A novice OO designer must know

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules

A good player knows

A novice OO designer must know

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

A good player knows

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining ...

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining . . .
- strategies : x-rays, immobilize, win with only two bishops . . .

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining . . .
- strategies : x-rays, immobilize, win with only two bishops . . .
- apertures, famous matches

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

- inheritance, encapsulation, data abstraction . . .

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining . . .
- strategies : x-rays, immobilize, win with only two bishops . . .
- apertures, famous matches

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

- inheritance, encapsulation, data abstraction . . .
- UML notation

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining . . .
- strategies : x-rays, immobilize, win with only two bishops . . .
- apertures, famous matches

An expert designer knows

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

- inheritance, encapsulation, data abstraction . . .
- UML notation

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining . . .
- strategies : x-rays, immobilize, win with only two bishops . . .
- apertures, famous matches

An expert designer knows

- object oriented principles

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

- inheritance, encapsulation, data abstraction . . .
- UML notation

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining . . .
- strategies : x-rays, immobilize, win with only two bishops . . .
- apertures, famous matches

An expert designer knows

- object oriented principles
- examples of good designs

Why patterns ?

A novice chess player knows

- the game rules
- the value of all pieces

A novice OO designer must know

- inheritance, encapsulation, data abstraction . . .
- UML notation

A good player knows

- tactics : occupy central cells, do not move the same piece twice at begining . . .
- strategies : x-rays, immobilize, win with only two bishops . . .
- apertures, famous matches

An expert designer knows

- object oriented principles
- examples of good designs
- design patterns

You should know

- what's a design pattern
- which OO principle drives them
- what's composite, when to apply it
- why design patterns are important