Design patterns in videogames

Videogames Technology
Asignatura transversal

Departamento de Automática

Universidad de Alcalá
ISG

Objectives

- Understand the need of design patterns
- Distinguish the main design patterns categories
- Apply the main patterns to problems in videogames

Bibliography

1. *Desarrollo de Videojuegos, Arquitectura del Motor de Vieojuegos. Capítulo 4. UCLM.*
2. Wikipedia

Table of Contents

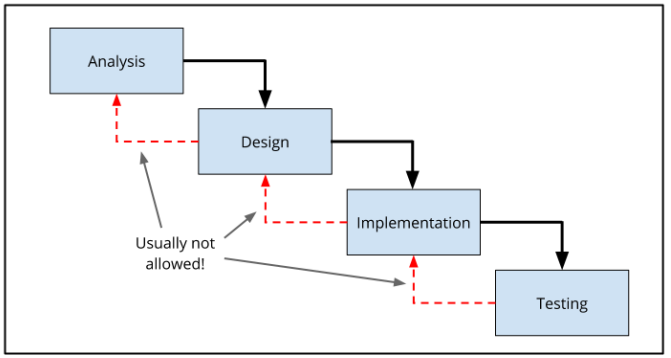
Software Engineering in videogames (I)

Game programming is a complex task

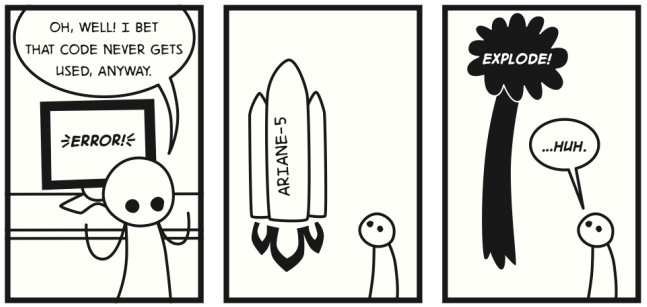
- Rarely done by a single person
- Development team ⇒ Software Engineering

Classic development process (**software lifecycle**)

1. *Analysis*: What do I need?
2. *Design*: How do it?
3. *Implementation*: Do it
4. *Testing*: Does it work?



The waterfall process



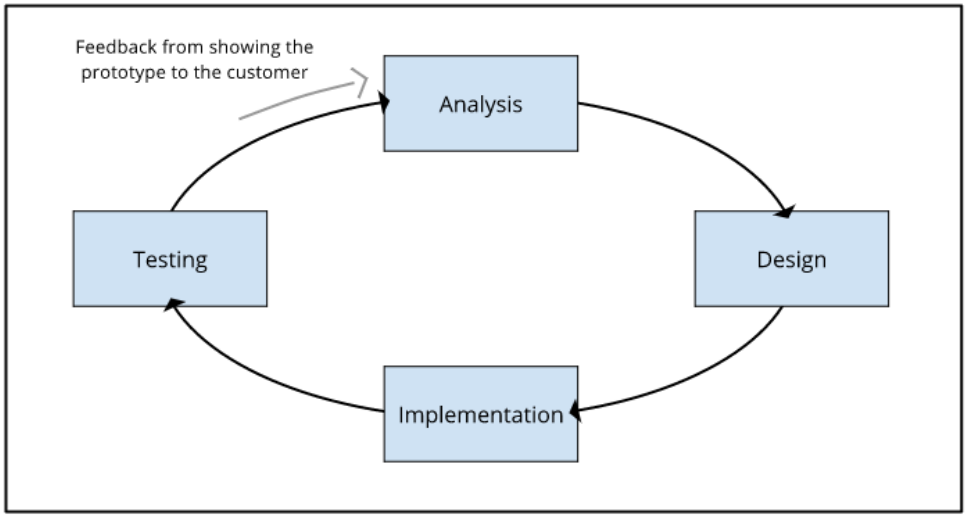
(Source)

More: `http://en.wikipedia.org/wiki/Iterative_and_incremental_development`

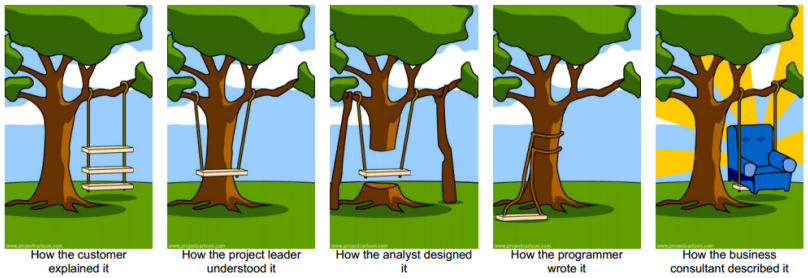
Software Engineering in videogames (II)

Many development processes

- Usually, game development is **iterative**



Iterative software development



Design pattern definition

Concept (I)

Some problems happen frequently

- Experience is a valuable asset, but it is not enough
- A design pattern stores knowledge on successful designs

Design pattern

It is the description of the communication among objects and classes customized to solve a generic design problem under a given context

Design Patterns. Elements of Reusable Object-Oriented Software Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (GoF- Gang of Four), 2008

Design pattern definition

Concept (II)

Informal definition: A design pattern is a solution to a design problem

- Its utility has been verified by experience
- It must be reusable

More: `http://en.wikipedia.org/wiki/Software_design_pattern`

Design pattern definition

Concept (III)

Design patterns goals

- Provide a portfolio of reusable elements in software design
- Avoid loose time searching solutions to already solved problems
- Formalize a shared vocabulary
- Standarize designs
- Ease learning

Design pattern do not want to

- Impose some design alternatives
- Remove designer creativity

Design pattern definition

Design pattern structure

Four components:

1. **Name**. Short name that identifies the pattern
2. **Problem and context**. Problem that the pattern solves, context where it takes sense and list of *preconditions*
3. **Solution**. General solution not tied to any programming language. Usually described with UML diagrams.
4. **Advantages/drawbacks**.

Additionally:

- Classification, applicability, structure, roles, colaborators, implementation, example code, related patterns, ...

Design pattern definition

Types of design patterns

Three great groups:

1. **Creational patterns**. Objects and data structures creation
   - Singlenton, factory, abstract factory, ...
2. **Structural patterns**. Class hierarchy, relation and composition of objects
   - Model-View-Controller (MVC), adapter, façade, proxy, ...
3. **Behavioral patterns**. Objects message passing (communication)
   - Observer, chain of responsability, command, iterator, state, strategy, ...

Additional domain patterns

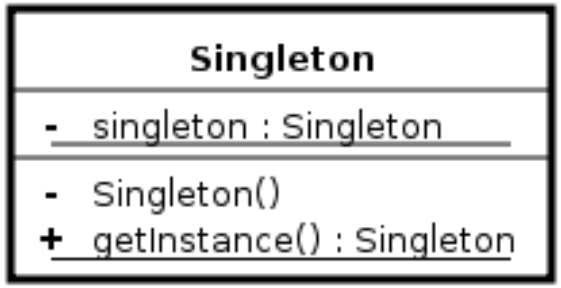
- Web development, GUIs, business, ...

Creational patterns

Singlenton

Singlenton

Problem: Guarantee only one instance of a class
Solution: Private constructor, instanciate the class through a public method
Example: We need only one game instance



Code example

```
public class Singleton {
  private static Singleton INSTANCE = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() { return INSTANCE; }
}
```

Creational

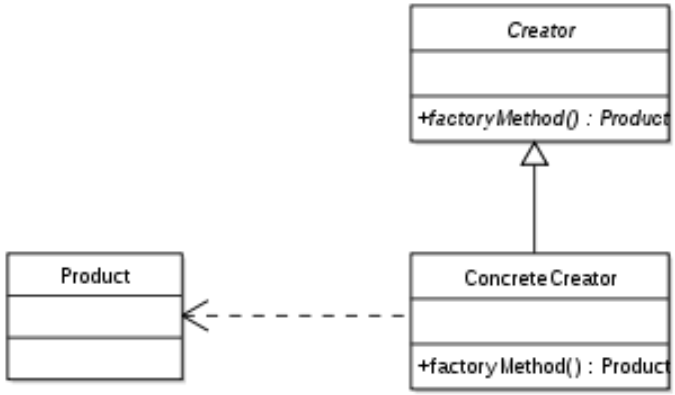
Factory

Factory

Problem: Create new object

Solution: Group object creation login in a factory class

Example: Create warriors and rogues in a RPG game



Factory code example

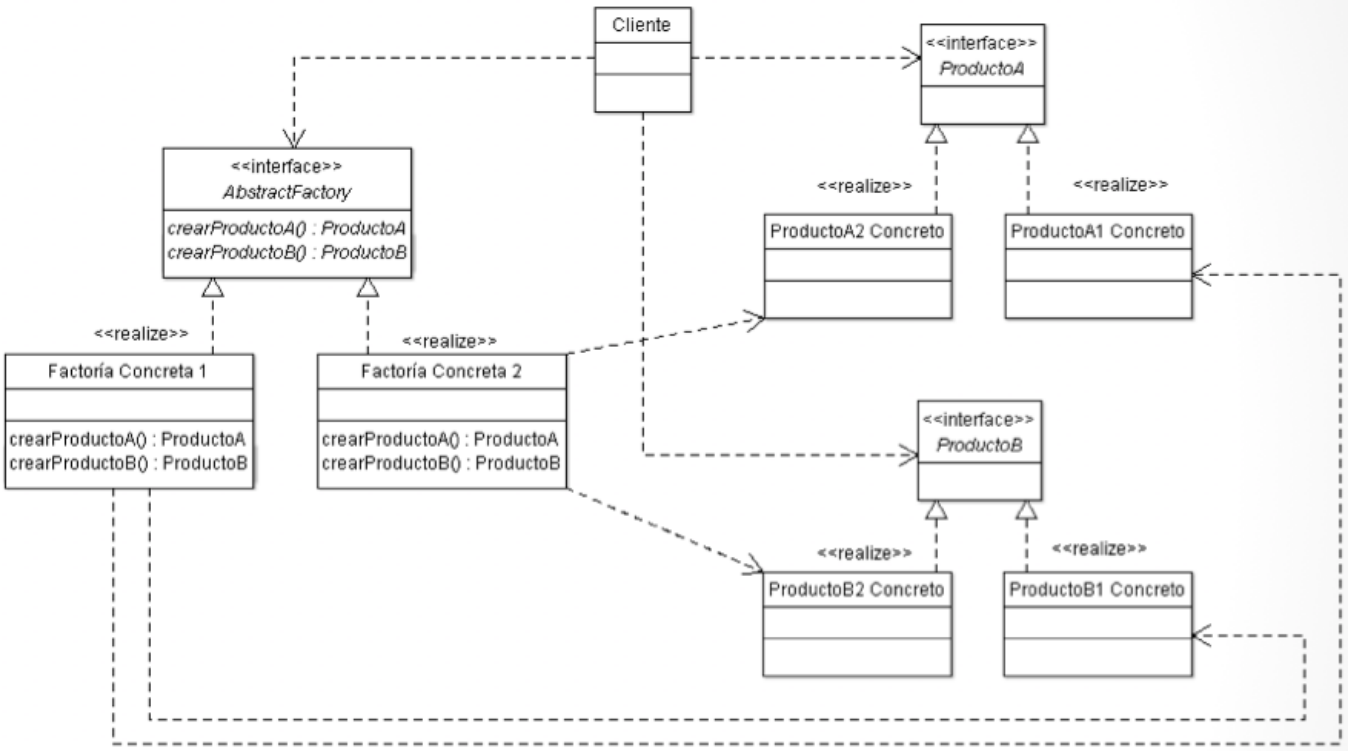
```
public class CarFactory {
  public static Car buildCar(String model) {
    switch (model) {
      case "small":
        return new SmallCar();
      case "sedan":
        return new SedanCar();
      case "luxury":
        return new LuxuryCar();
      }
    }
}
```

Creational patterns

Abstract Factory (I)

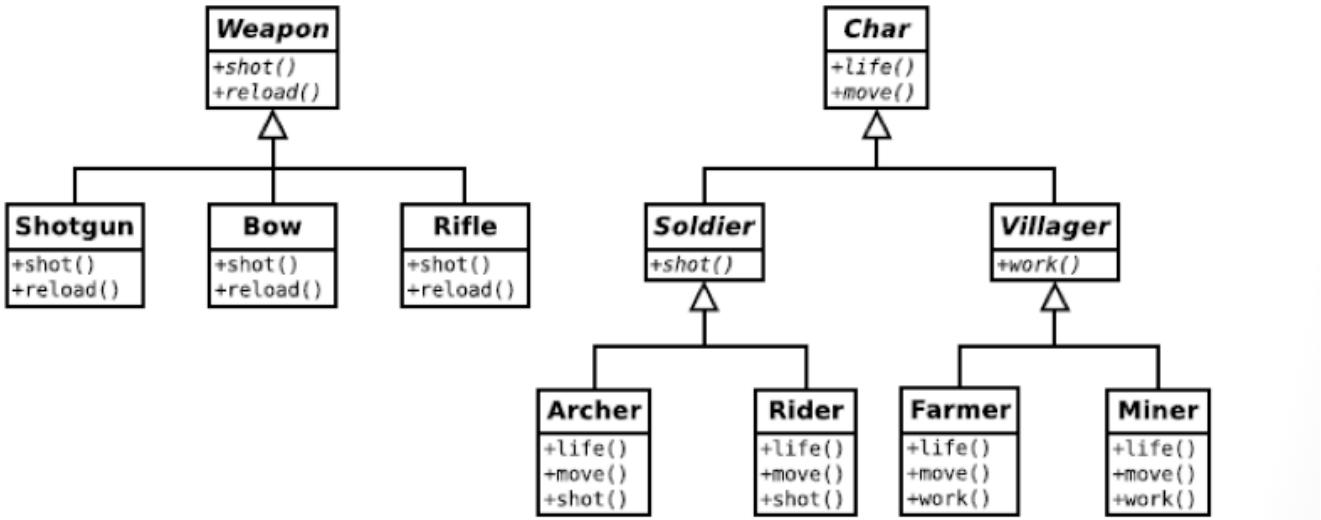
Abstract Factory

Problem: Create families of new objects
Solution: Create a hierarchy of factories
Example: Create human or orc warriors in a RPG game



Creational patterns

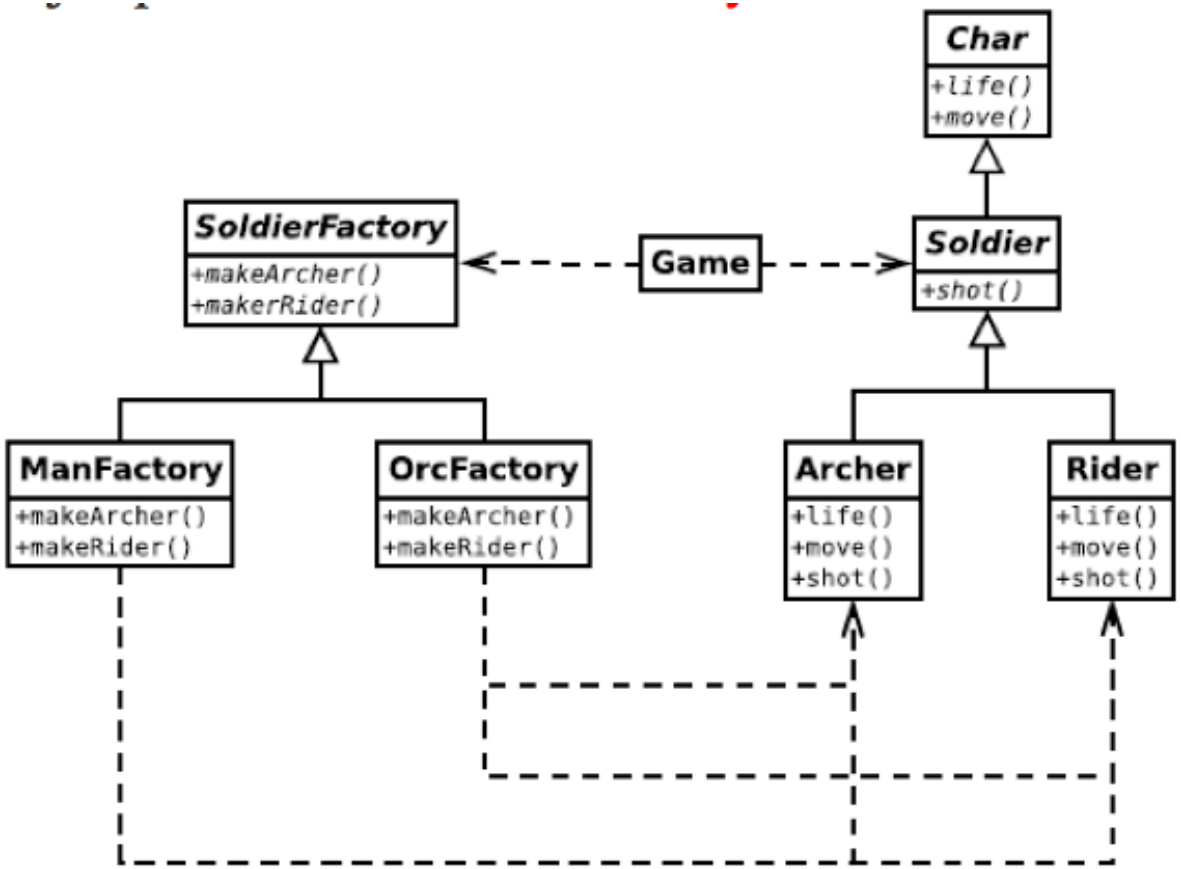
Abstract Factory (II)



RTS game class hierarchy

Creational patterns

Abstract Factory (III)



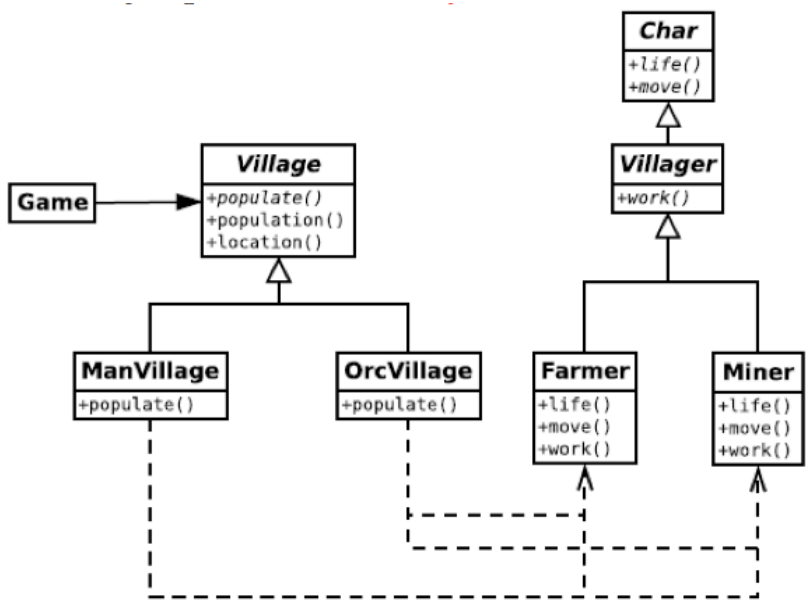
Example of *abstract factory* applied to a RTS game

Creational patterns

Factory Method

Factory Method

Problem: Create new objects
Solution: Method that instanciates objects
Example: Populate a village with characters

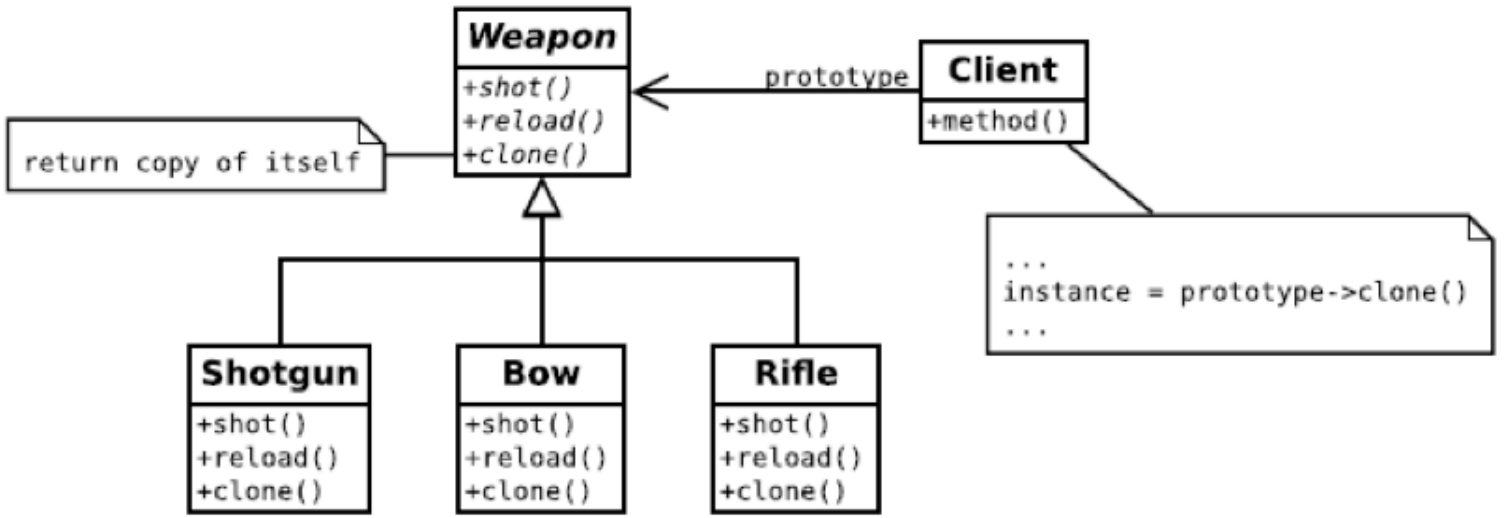


Design patterns

Creational patterns: Prototype

Prototype

Problem: Create a large number of objects whose instantiation is heavy
Solution: Clone objects
Example: Instanciate a large number of weapon objects

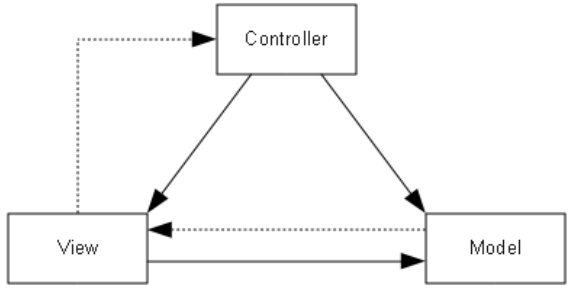


Structural patterns

MVC (I)

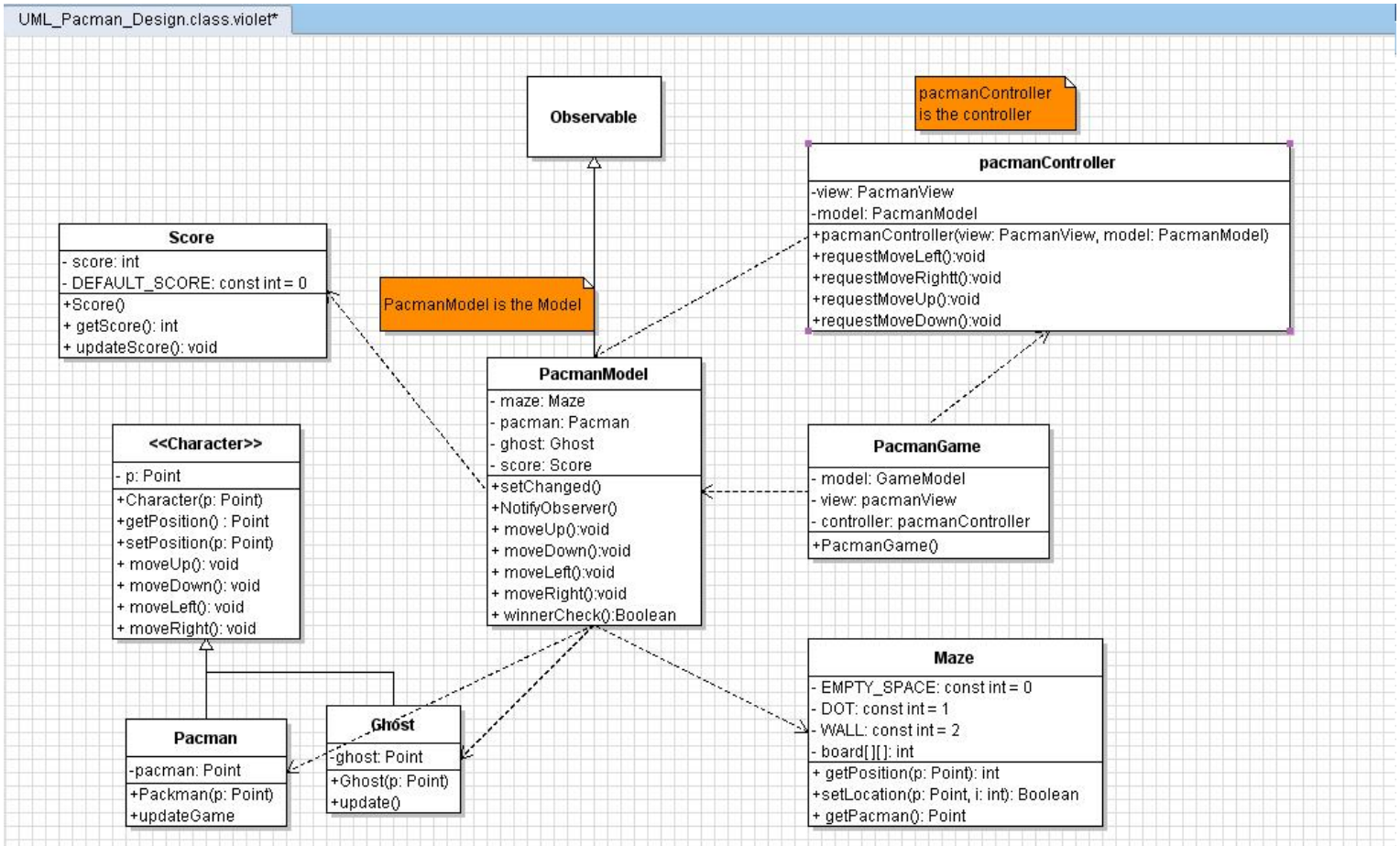
Model-View-Controller (MVC)

Problem: Decouple logic, data and visualization
Solution: Use different classes to contain data, its visualization and the game control
Example: Any game or graphical application



Design patterns

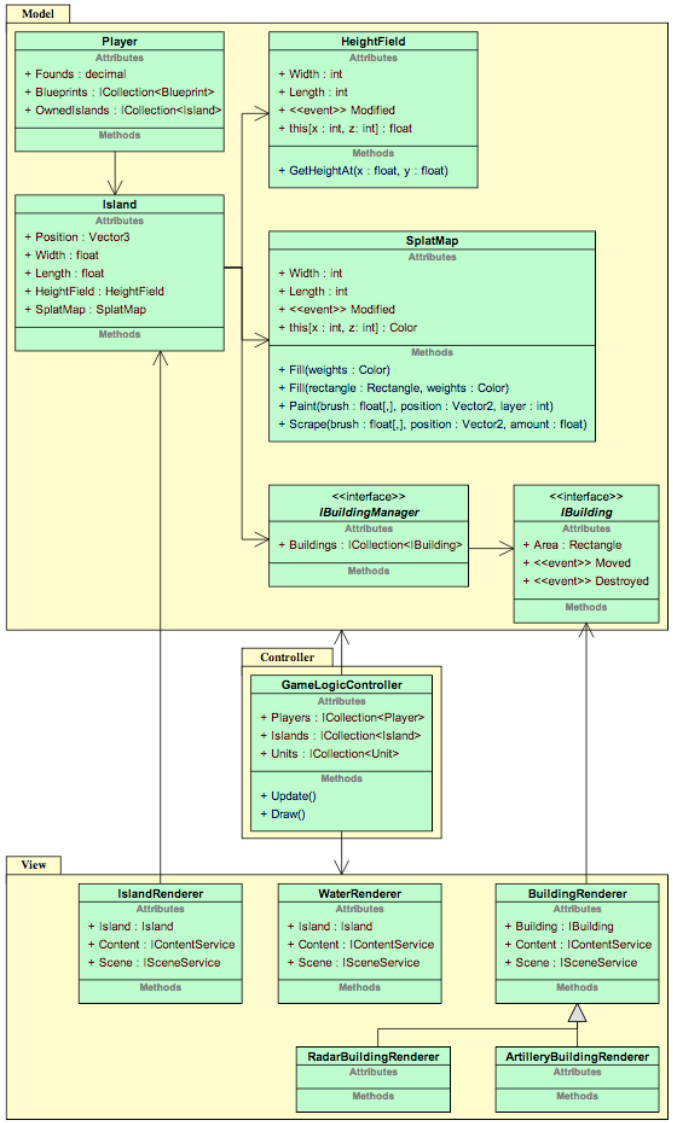
Structural patterns: MVC (II)



Source: `https://code.google.com/p/pacpounder/downloads/list`

Design patterns

Structural patterns: MVC (III)



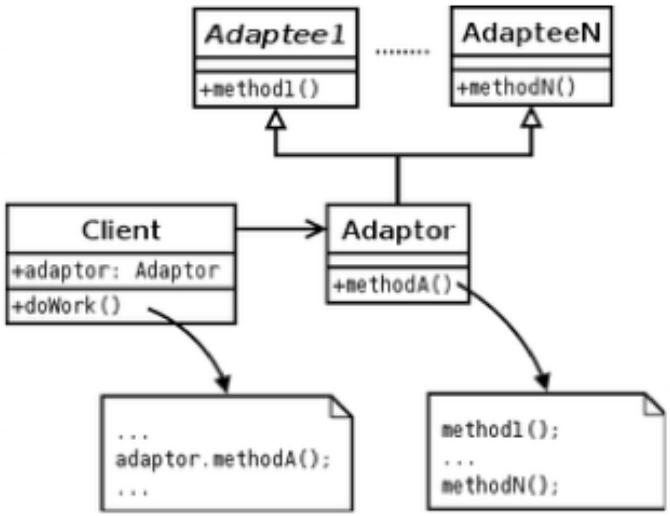
Source: `http://blog.nuclex-games.com/2010/09/mvc-in-games/`

Structural patterns

Adapter

Adapter

Problem: One class needs to invoke a method in another class, but it cannot
Solution: Use an intermediate class with a new interface
Example: Incompatible third-party library



Structural patterns

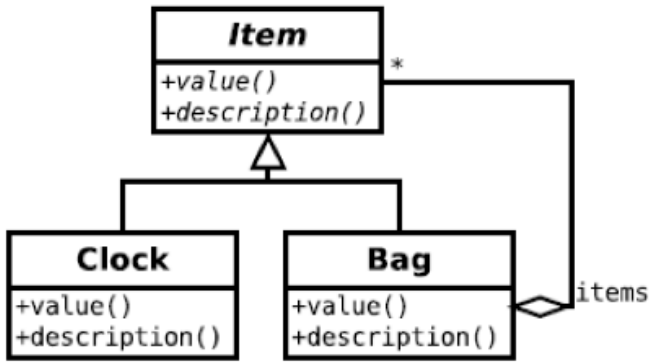
Composite

Composite

Problem: Store objects that might contain other objects

Solution: Objects composition

Example: Game whose player keeps an inventory whose items might contain other items



Structural patterns

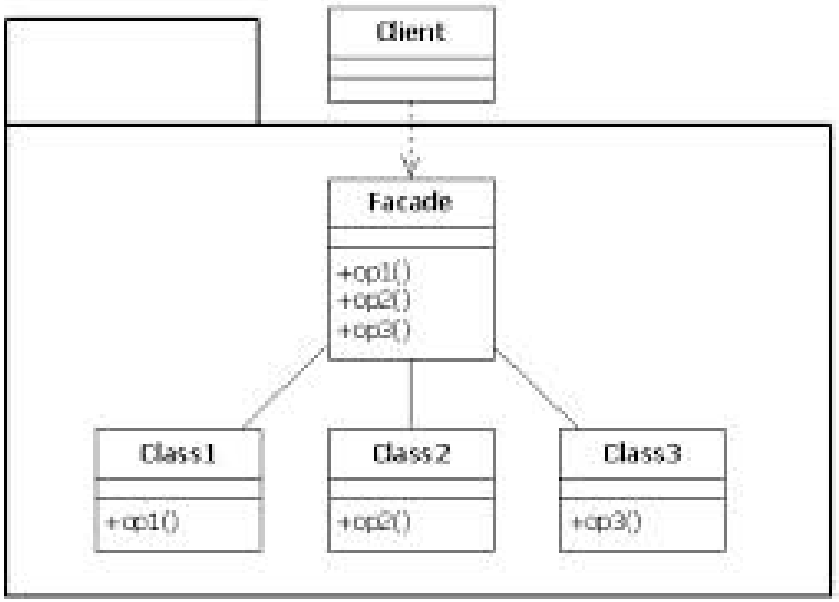
Façade

Façade

Problem: Complex interface to a set of classes

Solution: Create an intermediate class that simplifies the interface

Example: Graphical library with several operation modes

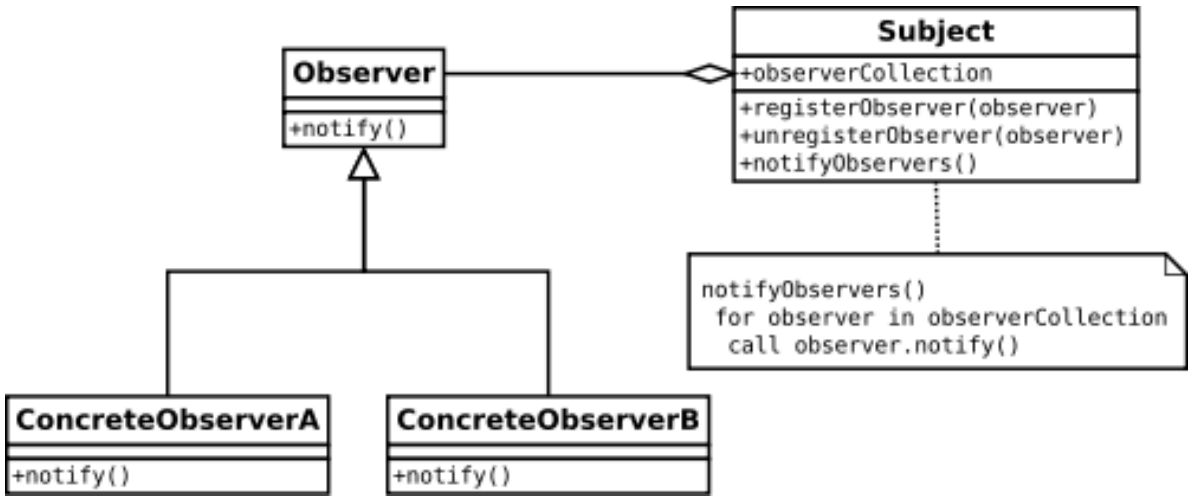


Behavioral patterns

Observer (I)

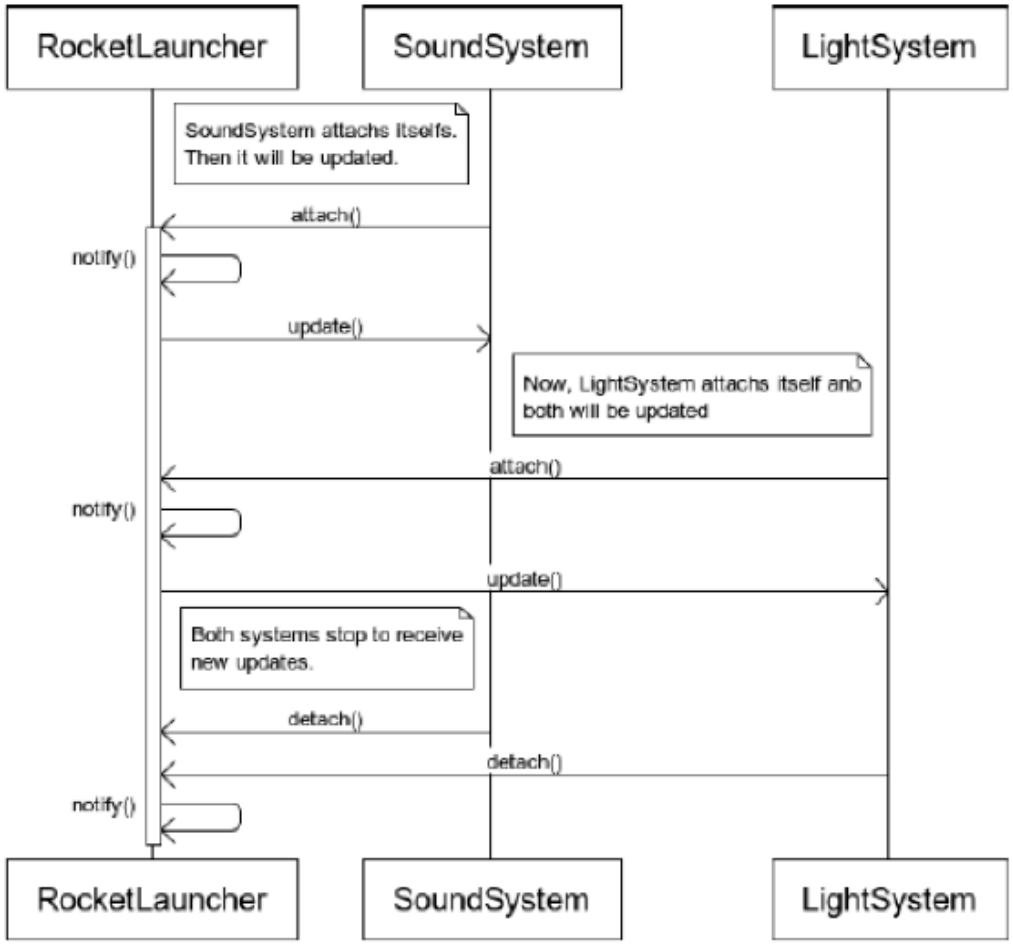
Observer

Problem: Notify a set of objects when another object changes
Solution: Link a set of observers to an observed object
Example: A view that has to know when the model changes



Behavioral patterns

Observer (II)



Behavioral patterns

Observer (III)

DataStore.java

```
public class DataStore extends Observable {
  private String data;

  public String getData() { return data; }

  public void setData(String data) {
    this.data = data;
    setChanged();
    notifyObservers();
  }
}
```

Screen.java

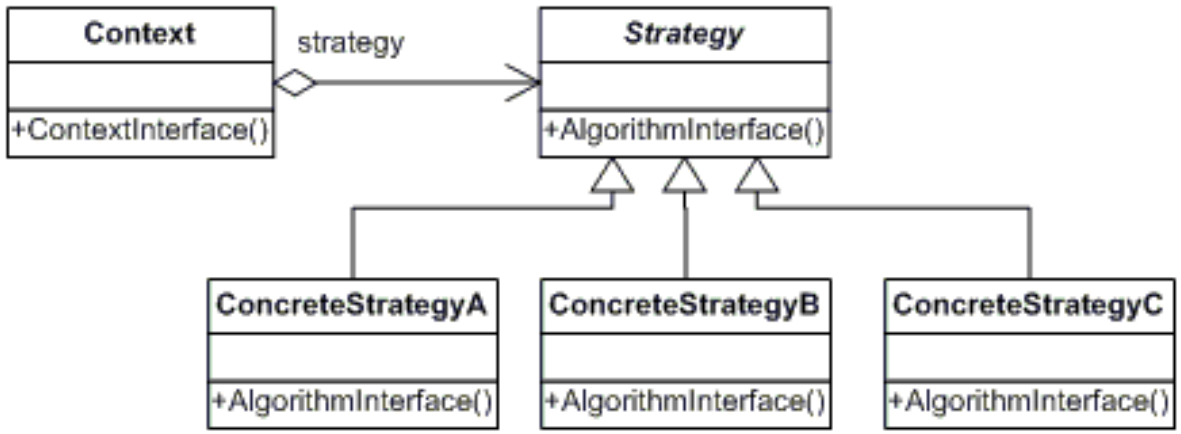
```
public class Screen implements Observer {
  @Override
  public void update(Observable ob, Object arg) {
    // Do something
  }

  public static void main(String args[]) {
    Screen screen = new Screen();
    DataStore datastore = new DataStore();
```

Behavioral patterns

Strategy (I)

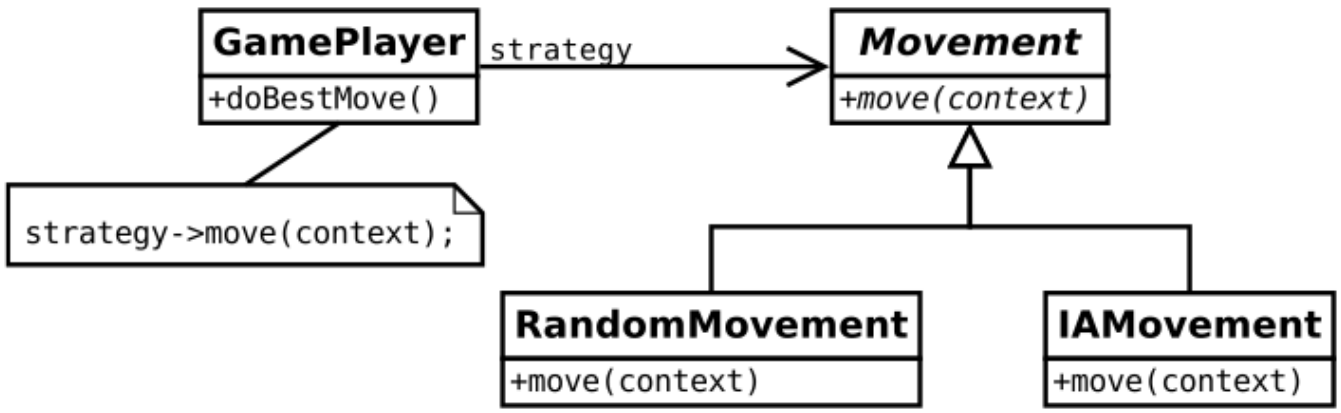
Observer

Problem: Choose in execution time which method use from several ones
Solution: Encapsulate the method in a class
Example: A fighter with several fighting styles



Design patterns

Behavioral patterns: Stategy (II)

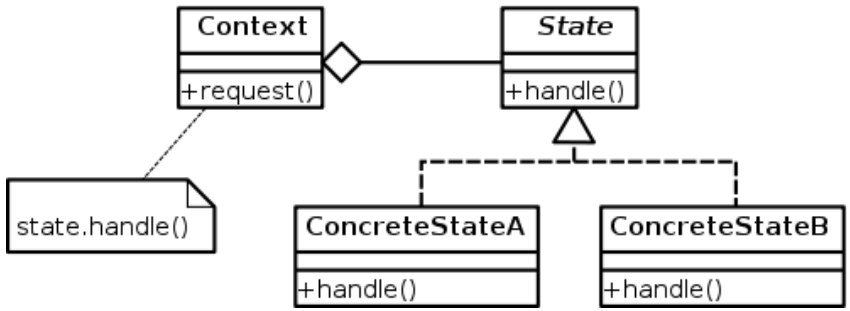


Behavioral patterns

State (I)

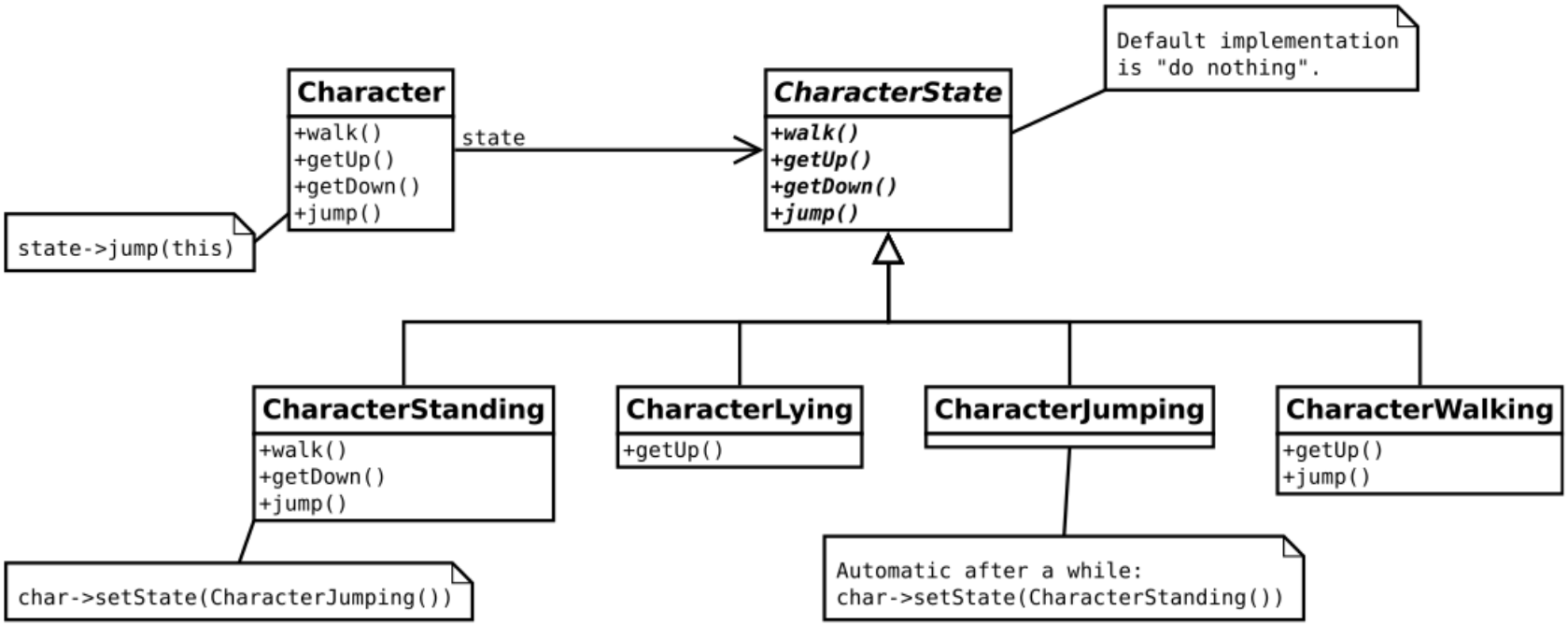
State

Problem: Implement a state machine
Solution: Encapsulate state transitions
Example: NPC behavior



Design patterns

Behavioral patterns: State (II)

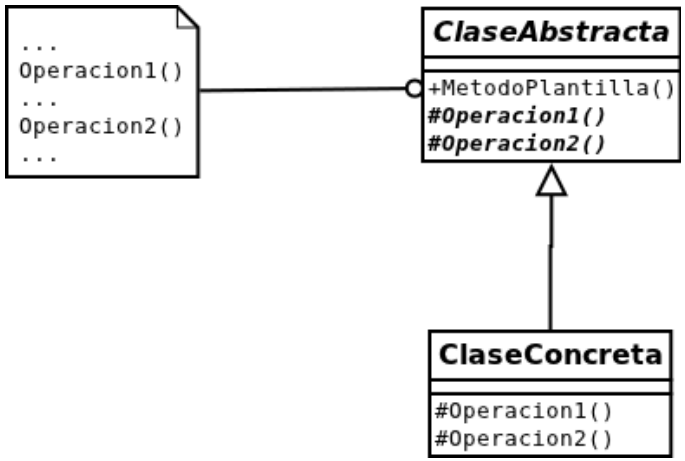


Behavioral patterns

Template Method (I)

Template Method

Problem: Customize an algorithm
Solution: Divide the algorithm in methods that can be overriden
Example: Chess and checkers games



Behavioral patterns

Template method (II)