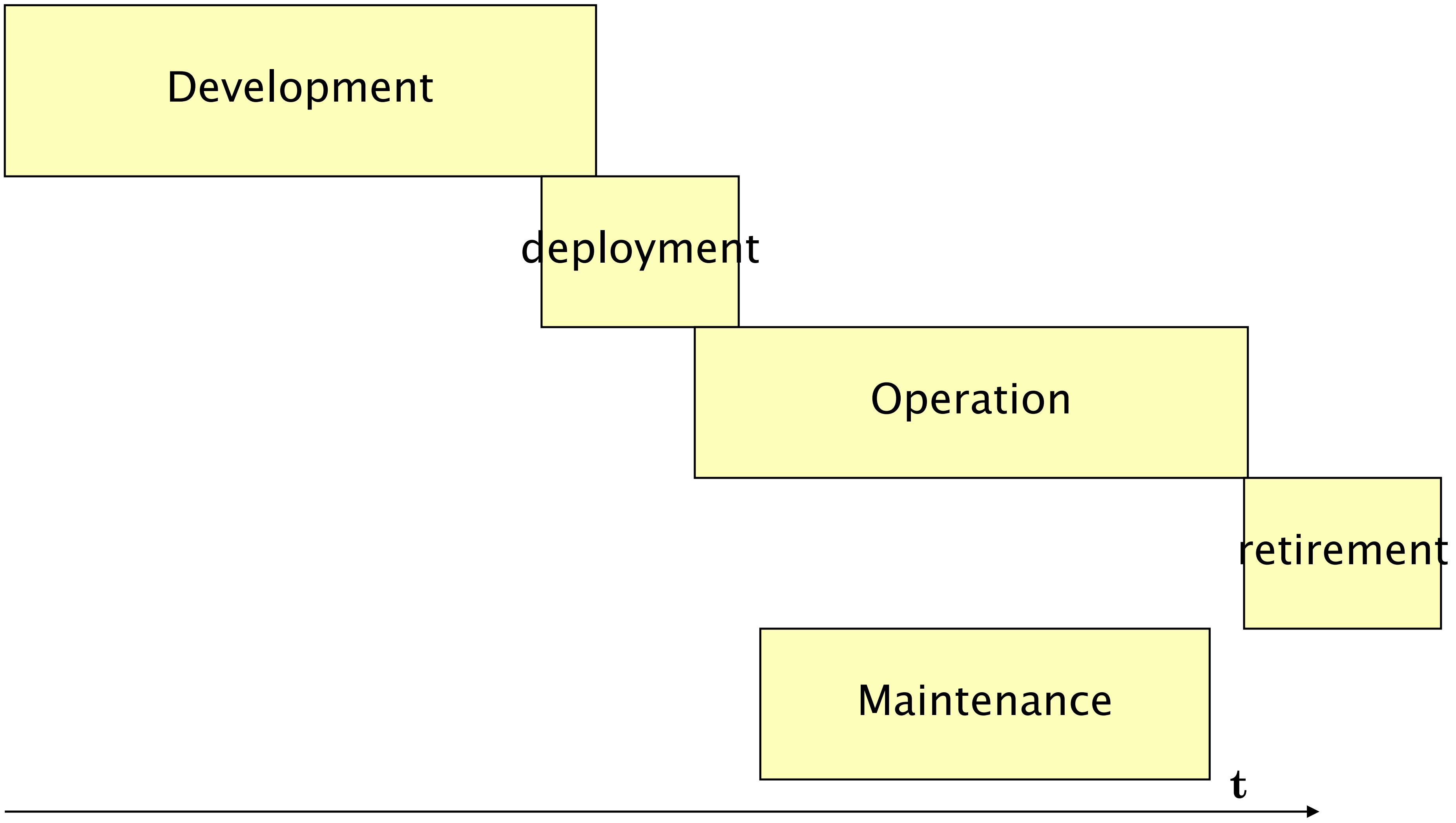
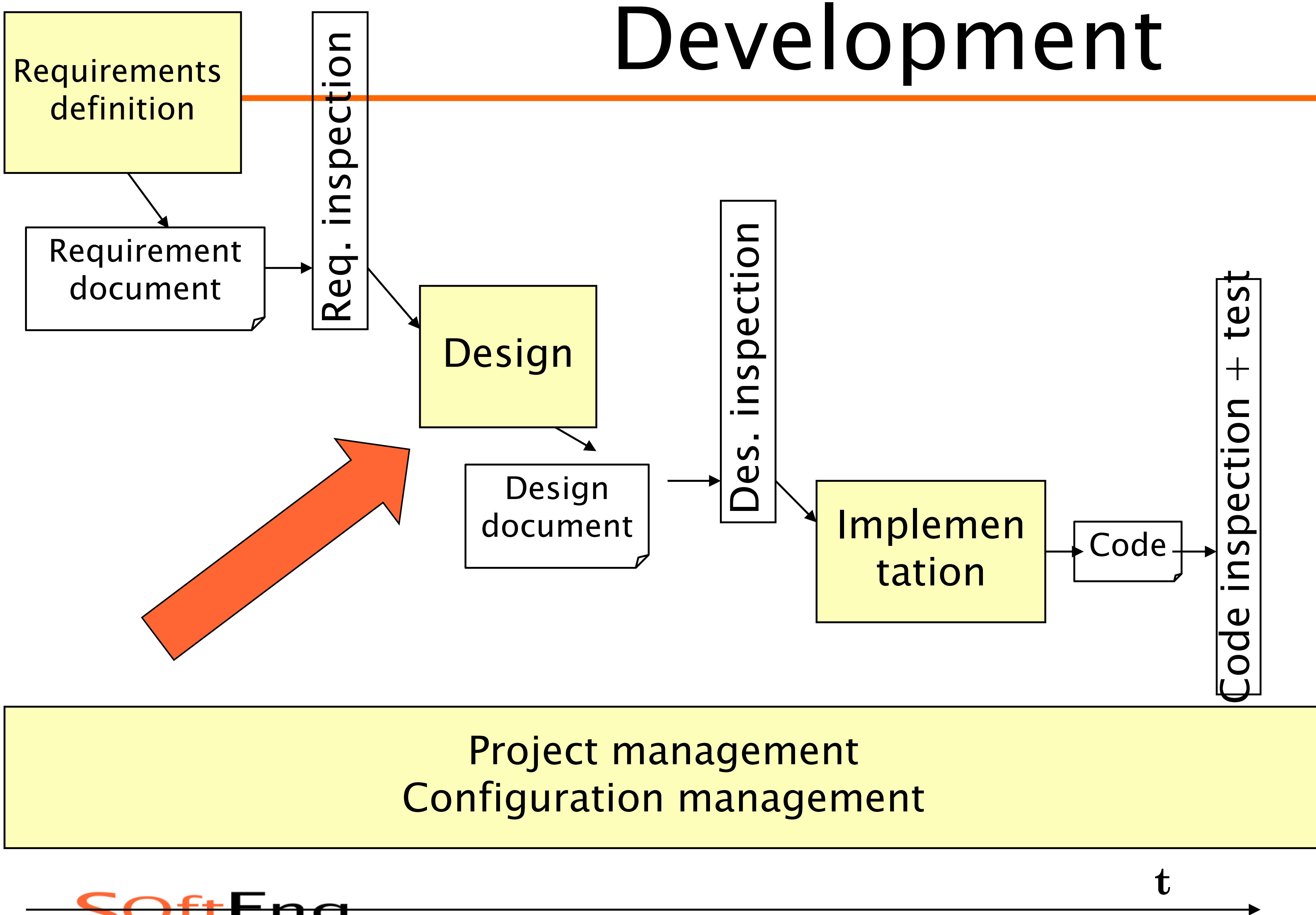
Architecture and Design

SoftEng
http://softeng.polito.it

Main Phases


Development
deployment
Operation
retirement
Maintenance
t

Development



Outline

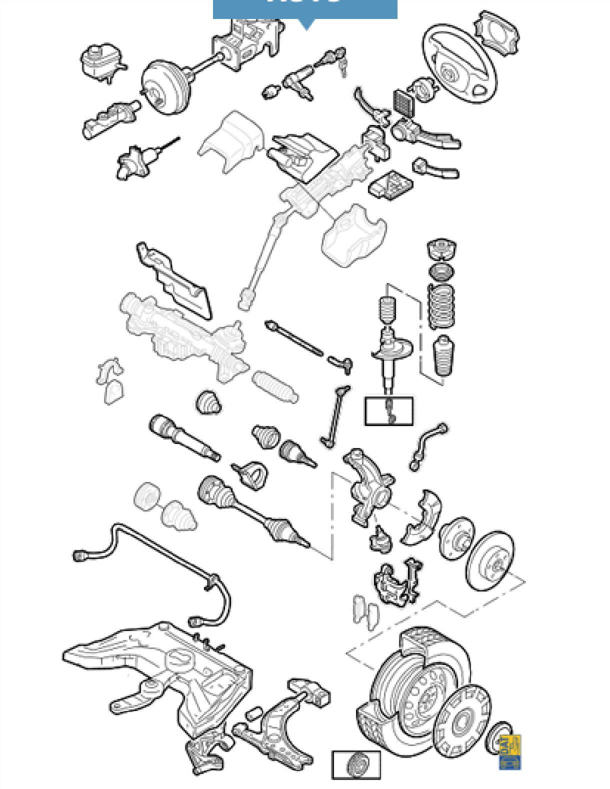
- Process
- Properties
- Notations
- Patterns
  - Architectural patterns /styles
  - Design patterns

Architecture

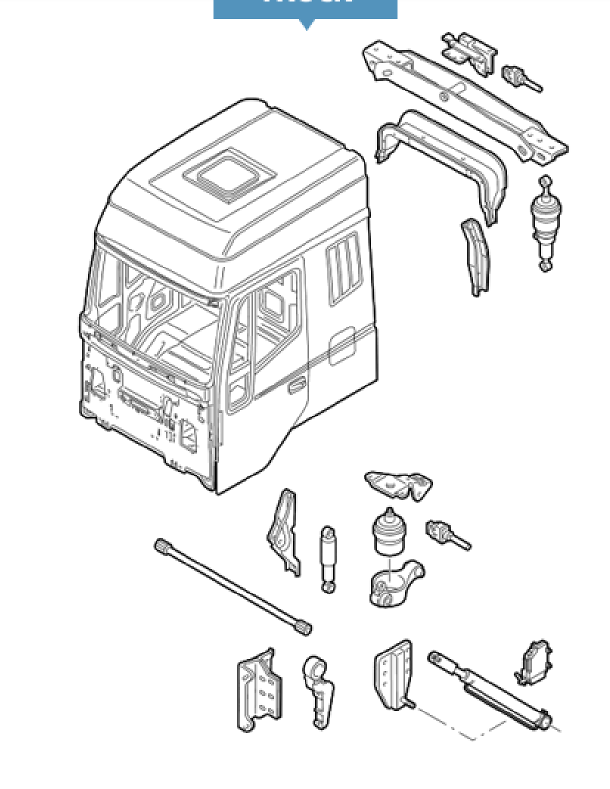
- Requirements: **what** the system should do
- Architecture, design: **how** the system should be built
  - Architecture, design: same flavour but
    - Architecture: high level, decide major components and their control and communication framework
    - Design: lower level, decide internals of each component

Vehicles

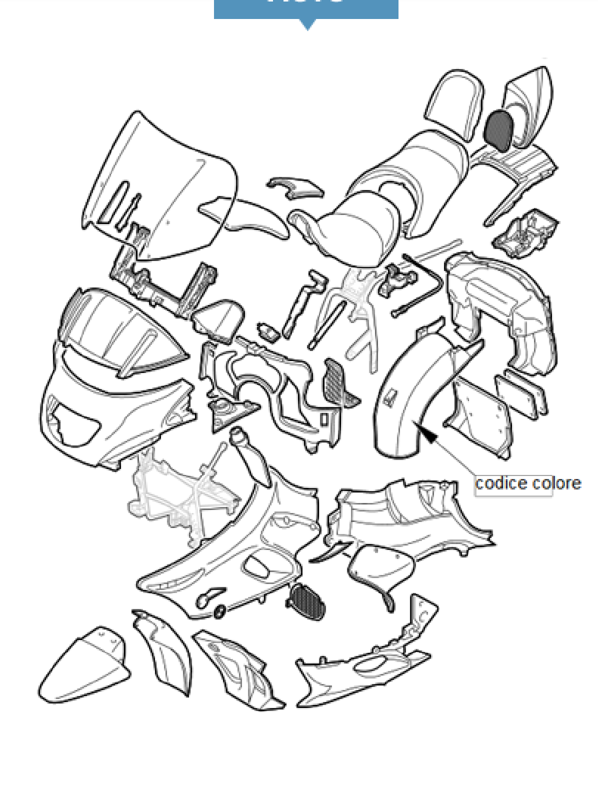
AUTO



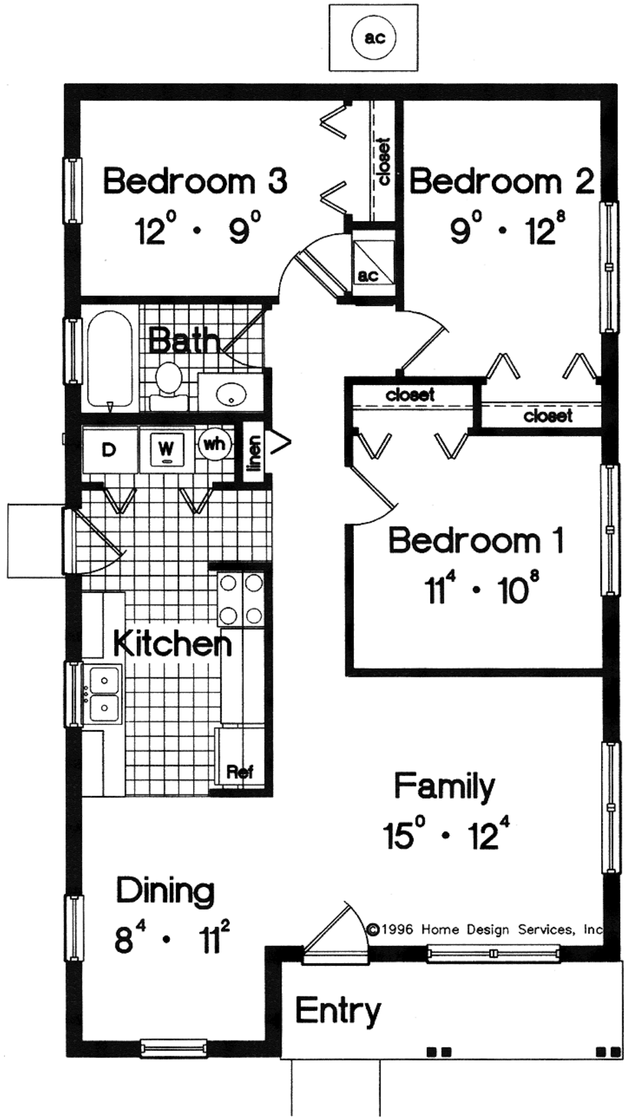
TRUCK

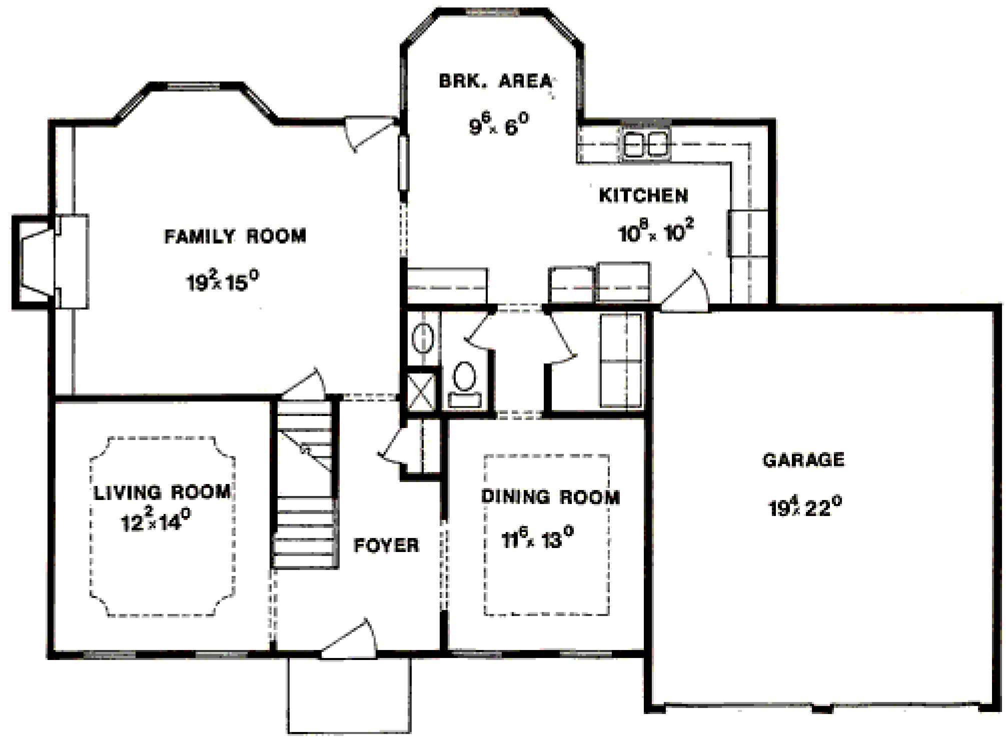


MOTO



Houses


ac
Bedroom 3
12° · 9°
closet
Bedroom 2
9° · 12⁸
ac
Bath
closet
closet
D
W
wh
linen
Bedroom 1
11⁴ · 10⁸
Kitchen
Ref
Family
15° · 12⁴
Dining
8⁴ · 11²
©1996 Home Design Services, Inc
Entry


BRK. AREA
9⁶ x 6⁰
KITCHEN
10⁸ x 10²
FAMILY ROOM
19² x 15⁰
LIVING ROOM
12² x 14⁰
FOYER
DINING ROOM
11⁶ x 13⁰
GARAGE
19⁴ x 22⁰

Architecture, design, why

- Most defects come from requirements and design
- Essential to define, analyze and evaluate design choices *early*
- If no design is defined, but code is developed immediately, design choices are made implicitly and evaluated *late*
- Doing design allows to make design choices *explicit* , document and evaluate them

Requirements to design

- Given one set of requirements
- In general many different designs are possible (*design choices*)
  - Cfr. Requirement: mid sized car in price range 10 to 20k
  - Designs: hundreds of models on the market,
    - High level design choices
      - diesel or gas or electric engine
      - front or rear or all wheel drive
    - Low level design choices
      - Color, outer details
      - With ABS, ESP, .. or not
- But not all designs are equal

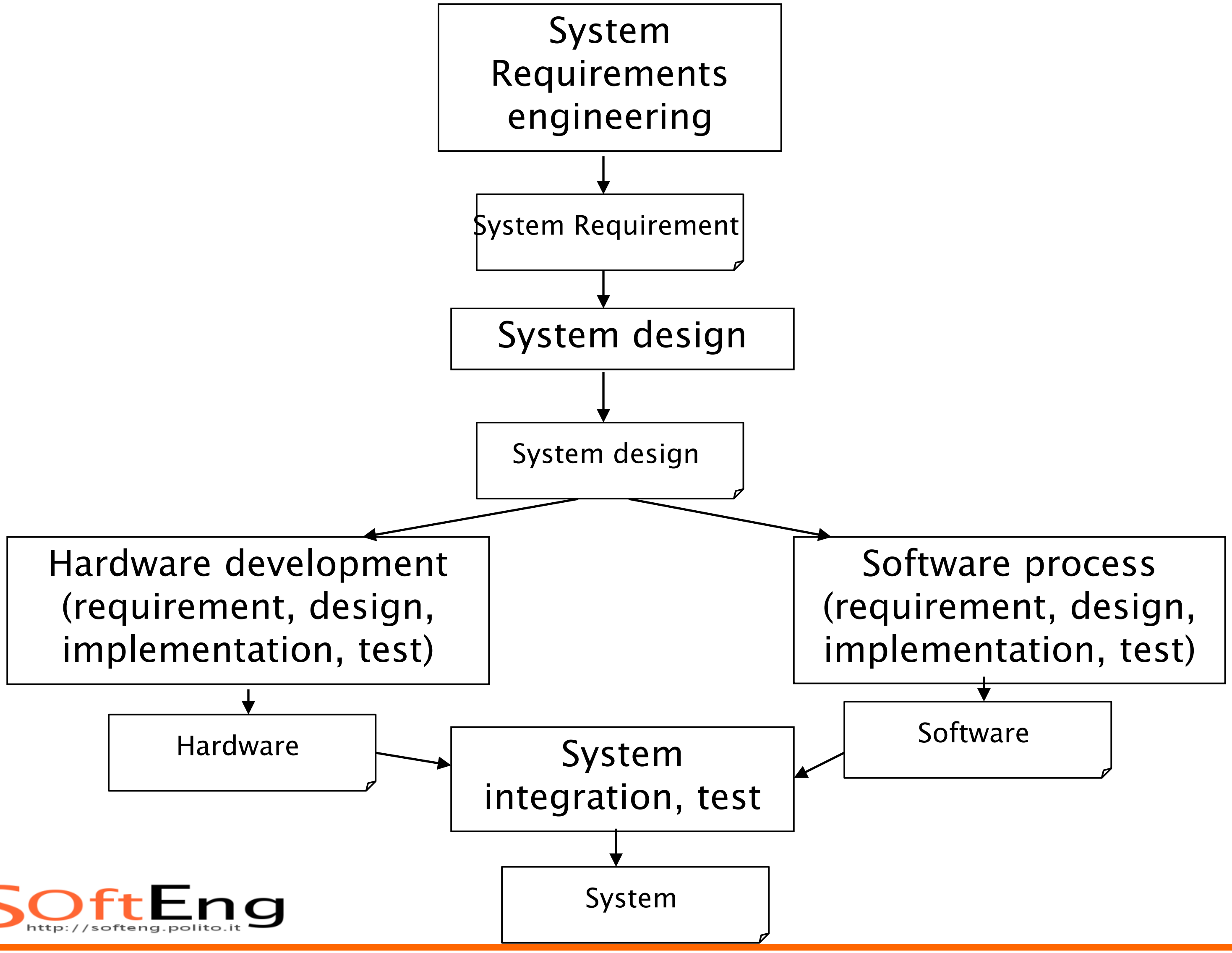
Requirements to design

- A creative process
- Driven by skill and experience
- Experience formalized in semi formal guidelines
  - Architectural styles (patterns)
  - Design patterns

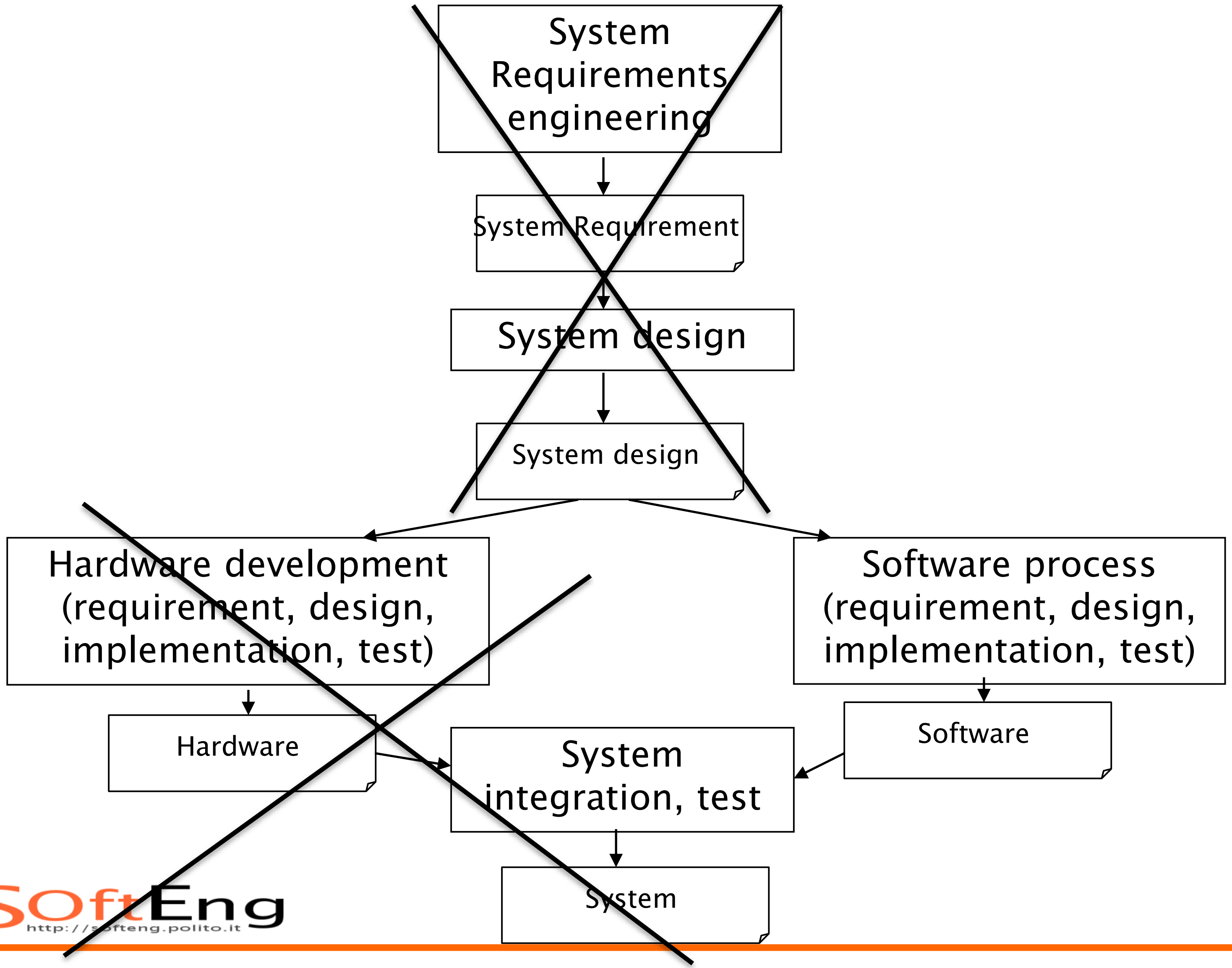
Two process variants

Software only process	System process
	▪ System requirements ▪ System design
▪ (Software) Requirements	▪ Software requirements
▪ (Software) Design	▪ Software design

System process



Software process



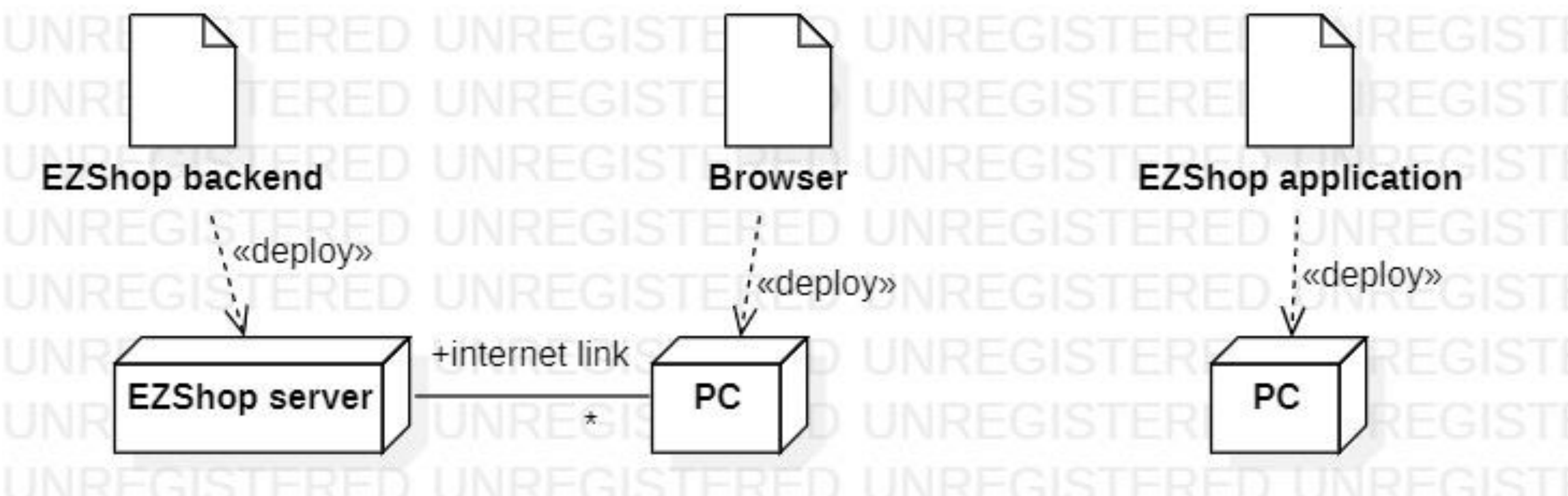
Hw – sw interaction

- Software process includes also a part about hardware components and software – hardware allocation (UML Deployment diagram)

Example

- Two options of hw –sw allocation for Ezshop

client server single app



Software design

- Having defined the hardware context, software design is about:

Defining software modules (functions, classes, packages, modules, ..) and their interactions

so that they satisfy
functional +
non functional requirements

Process

Process

- Analysis
  - Architecture
  - High level design
  - Low level design
- Formalization
  - Text, diagrams (UML)
- Verification

Process

- Input
  - Requirement document
    (functional requirements
    non functional requirements)
- Output
  - Design document
    - Components + connections
    - Capable of satisfying functional + non functional requirements

- 1 Architecture

  (about the whole system)

  - Define high level components and their interactions
  - Select communication and coordination model
    - Processes, threads
    - Messages, (remote) procedure calls, broadcast, blackboard
  - Use architectural style(s) /pattern(s)

- 2 Design
  - 2.1 High level (about many classes)
    - Define classes and their interactions
    - Use design patterns
  - 2.2 Low level (about one class)

Properties

Properties of a design

- Functional properties
  - Does the design support the functional requirements?
    - Functional requirements (requirements document)

      vs.
    - functional properties (design)
- Non functional properties
  - Does the design support the non functional requirements?
    - Non functional requirements (requirements document)

      vs.
    - Non functional properties (design)

Non functional properties

- Reliability
- Efficiency/performance
- Usability
- Maintainability
- Portability
- Safety
- Security

Non functional properties

- More specific to design
  - Testability
    - Observability
    - controllability
  - Monitorability
  - Interoperability
  - Scalability
  - Deployability
  - Mobility

Non functional properties

- Complexity
  - Number of components
  - Number of interactions
  - KISS: keep it simple, stupid
- Coupling (or decoupling)
  - Degree of dependence between two components
- Cohesion
  - Degree of consistence of functions of a component

Coupling

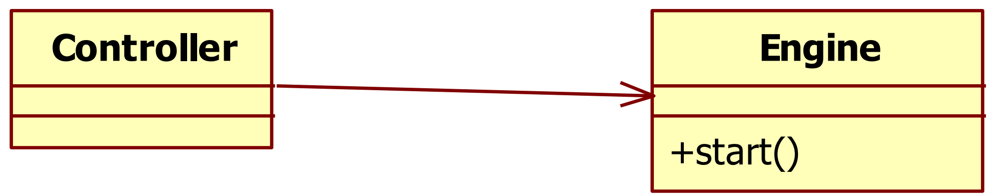
Walls vs plumbing system

lower

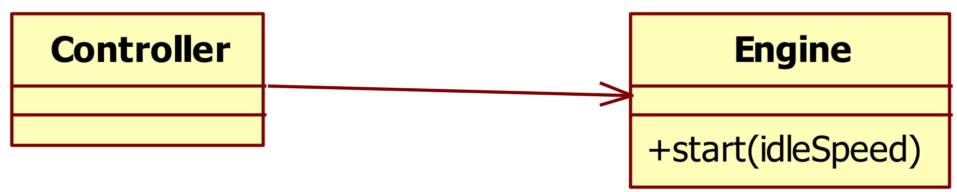
higher

Coupling

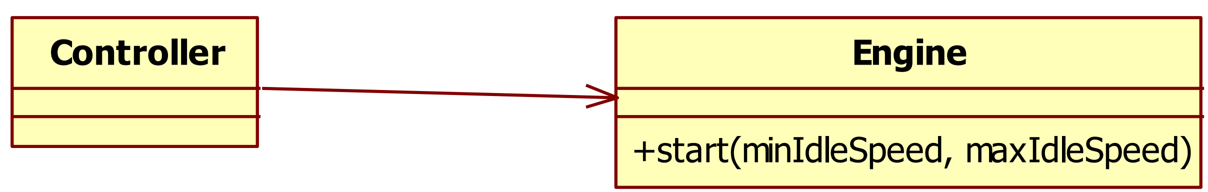
- Controller vs engine
  - lowest



  - intermediate

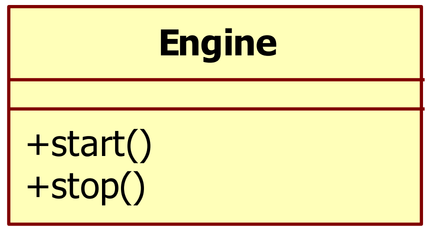


  - highest

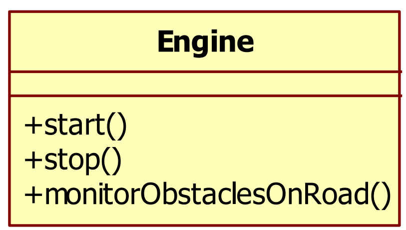


Cohesion

- Higher



- lower



Non functional properties

- Cost
- Schedule
- Staff skills

Properties – what to do

- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localise safety-critical features in a small number of sub-systems.
- Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.

Properties

- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localising safety-related features usually means more communication so degraded performance

Properties, trade offs

- Not all properties can be satisfied
- Design is also about deciding tradeoffs
  - Ex security (add layers) vs. speed (avoid layers)
  - Ex. changeability (add abstraction layer to insulate from hardware change) vs. speed (avoid layers)
- Possibly, trade offs are decided at requirement time
  - Ex: requirement: security prevails on speed

Notations for formalization of architecture

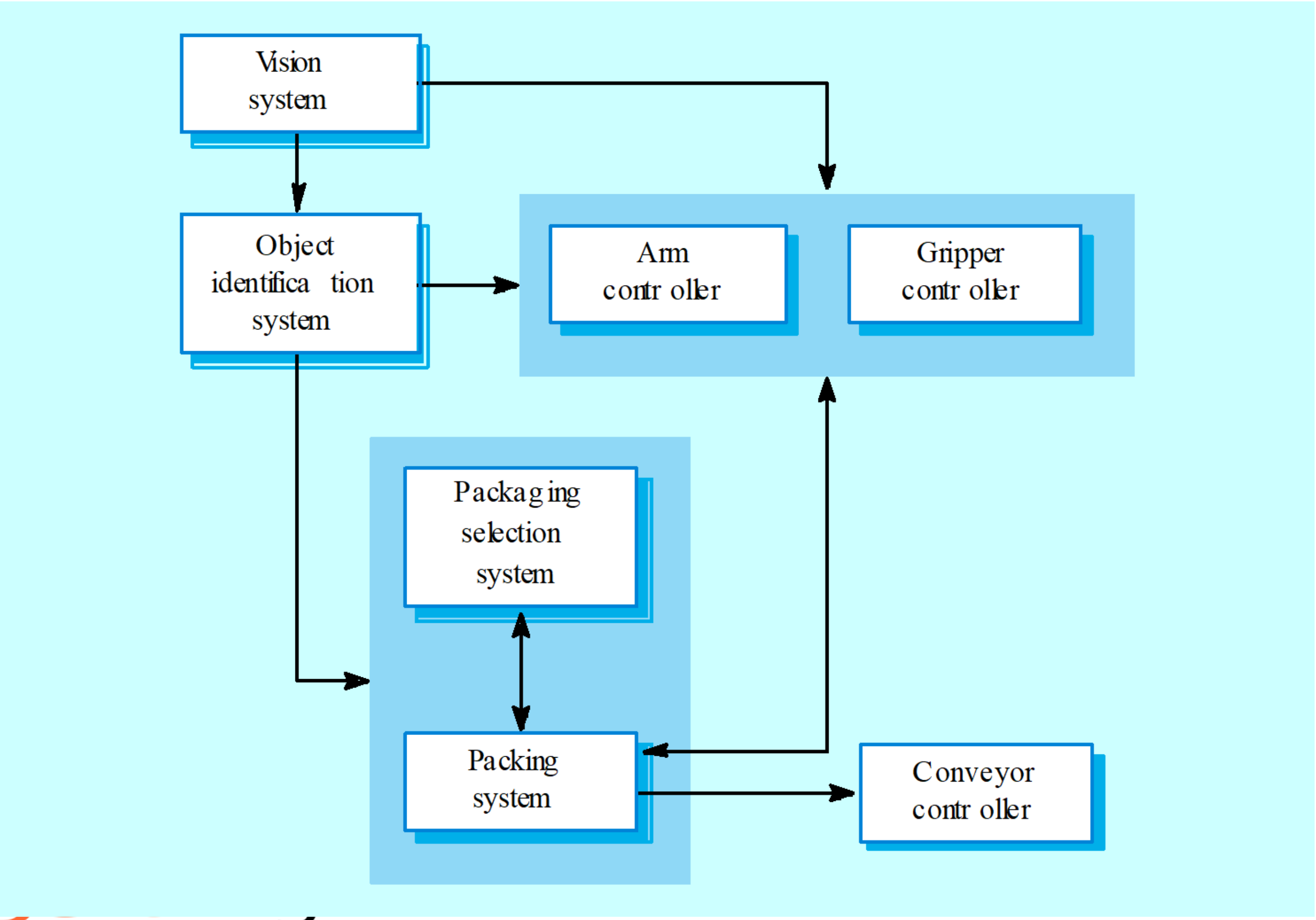
Formalizing the architecture

- Informal
  - box and lines
- Semiformal
  - UML diagrams
    - Structural views
      - Component, package diagrams
      - Class diagrams
      - Deployment diagram
    - Dynamic views
      - Sequence diagrams
      - State charts
- Formal ADL (Architecture description languages)
  - Many, ex C2 (component Connector)

Box and line diagrams

- Very abstract – they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Packing robot control system



UML diagrams

- Structural view
  - Component or package diagram for high level view
  - Class diagram (inside each package or component)
  - Class description (for each class)

Heating control system

- Goal: control temp in house, using sensors in each room, and actuators (open close heating in each room)
- Choices – high level
  - One CPU, one process (no distribution, no concurrency)
  - Communication and control: procedure call
  - Layered style (at least partially)
- Choices – low level
  - Observer pattern

Heating control system

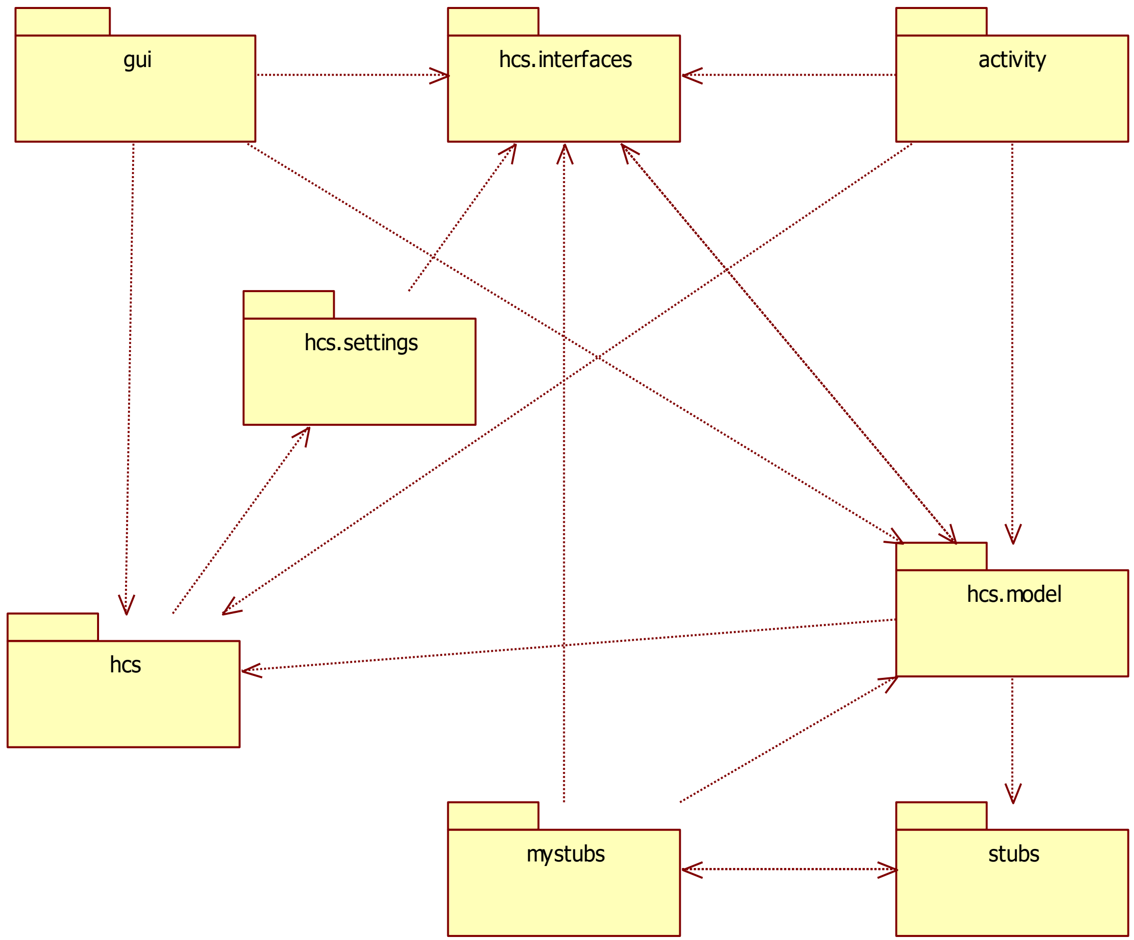
- Option 1
- High level:
    - One CPU, one process (no distribution, no concurrency)
    - Communication and control: procedure call
    - Layered style (at least partially)
    -

Heating control system

- Option 2
- High level:
  - One CPU per room, one CPU per house (distribution, concurrency)
  - Communication and control: http calls (house controller calls rooms)

UML – structural

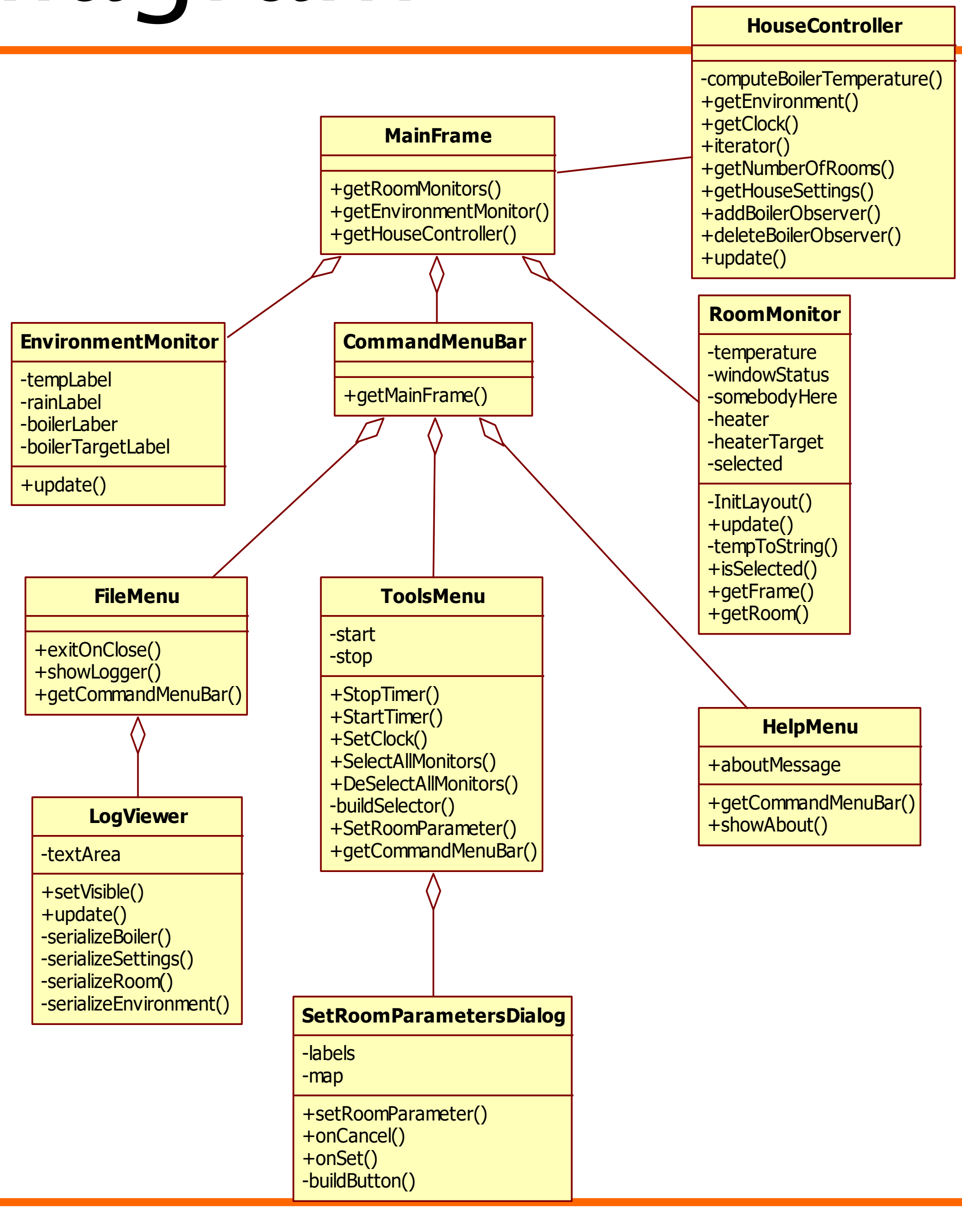
UML – package diagram



SoftEng http://softeng.polito.it

UML – class diagram

- Package GUI



Class (HouseController)

- The main class in the heating control system, it integrates the logical model of the various parts of the house and performs the high-level activities.
- computeBoilerTemperature()
  - Computes the desired water temperature in the boiler
- getEnvironment()
  - Navigates to the logical model of the environment
- getClock()
  - Navigates to the Clock
- iterator()
  - Returns an iterator to the contained Rooms
- getNumberOfRooms()
  - Returns the number of rooms
- getHouseSettings()
  - Navigates to the current global settings
- update()
  - Computes the next logical state of the system
- addBoilerObserver()
  - Adds an observer to the Boiler
- deleteBoilerObserver()
  - Removes an object from the list of Boiler observers

Structure and hierarchy

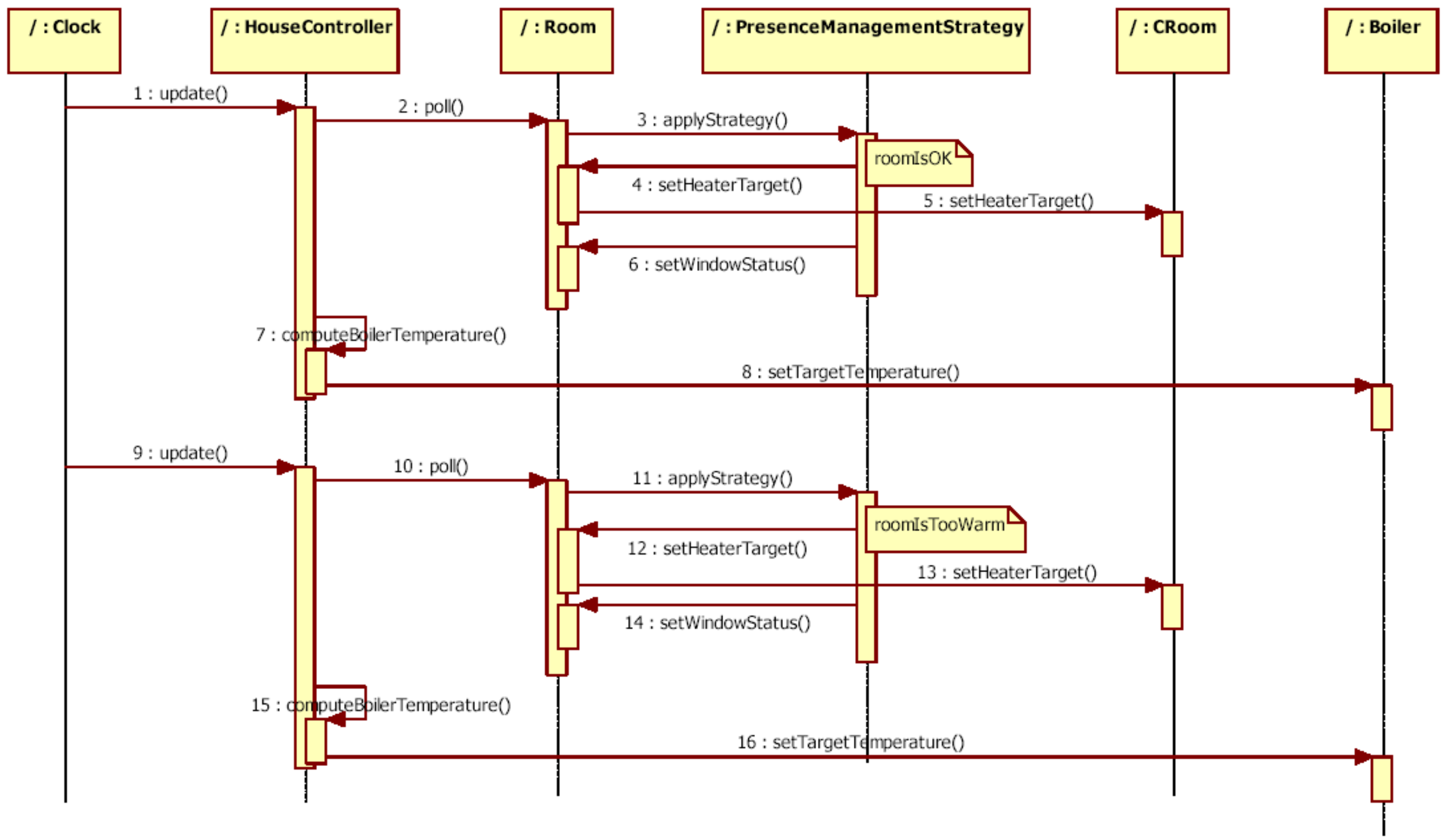
- UML helps in presenting structure in an organized (hierarchical) way
  - Packages in system
  - Classes in package
  - Attributes and methods in class
- Presentation is sequential, but the definition of such a structure requires several iterations

UML – dynamic

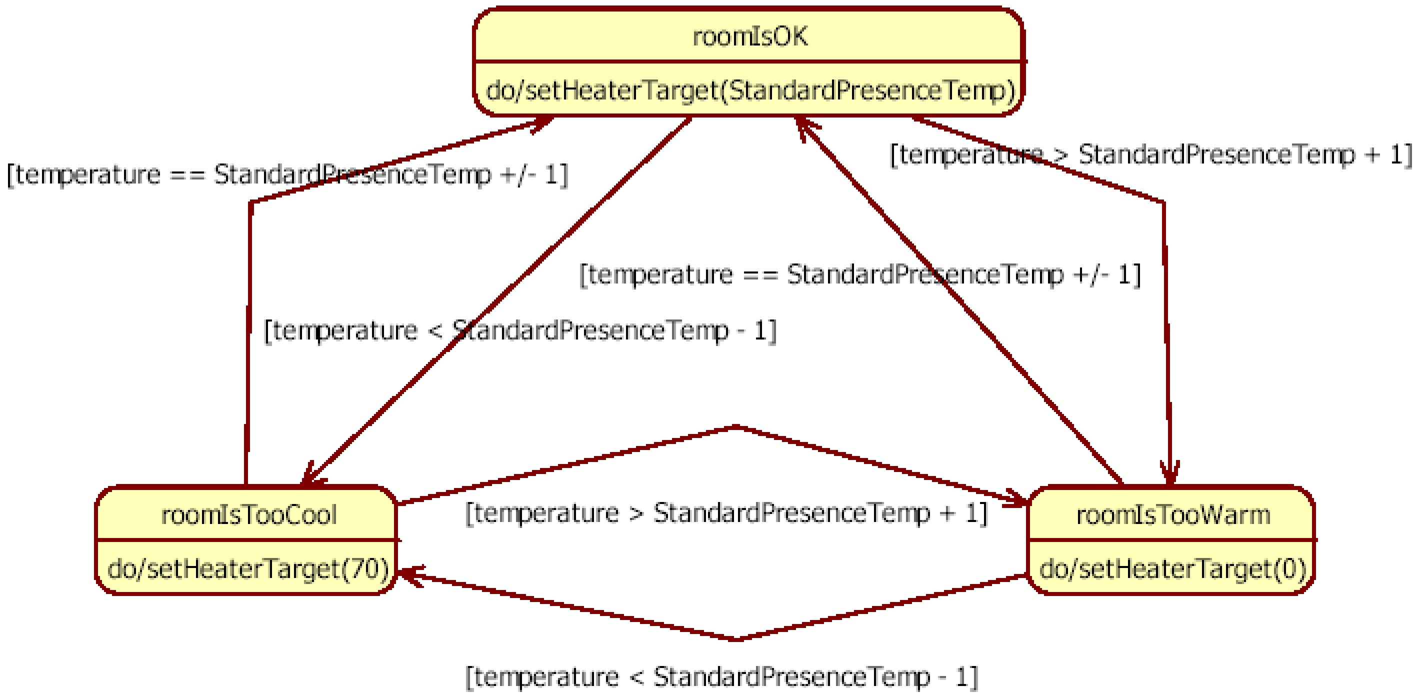
- State charts
- Sequence diagrams

Sequence

Sequence diagram for scenario 11:



State chart


roomIsOK
do/setHeaterTarget(StandardPresenceTemp)
[temperature == StandardPresenceTemp +/- 1]
[temperature > StandardPresenceTemp + 1]
[temperature == StandardPresenceTemp +/- 1]
[temperature < StandardPresenceTemp - 1]
roomIsTooCool
do/setHeaterTarget(70)
[temperature > StandardPresenceTemp + 1]
roomIsTooWarm
do/setHeaterTarget(0)
[temperature < StandardPresenceTemp - 1]

Patterns

Patterns

- Reusable solutions
- To recurring problems
- In a defined context

- Cfr also dominant design in technology management area

Patterns

- Known, working ways of solving a problem

History

- Initially proposed by Christopher Alexander
- He described patterns for architecture (of buildings)
  - *The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing and when we create it. It is both a process and a thing …*

Types of Pattern

- Architectural Patterns (or styles)
  - Address system wide structures
- Design Patterns
  - Leverage higher level mechanisms
- Idioms
  - Leverage language specific features

Patterns vs. process

- 1 Architecture
- 2 Design
  - 2.1 High level
  - 2.2 Low level

Architectural patterns

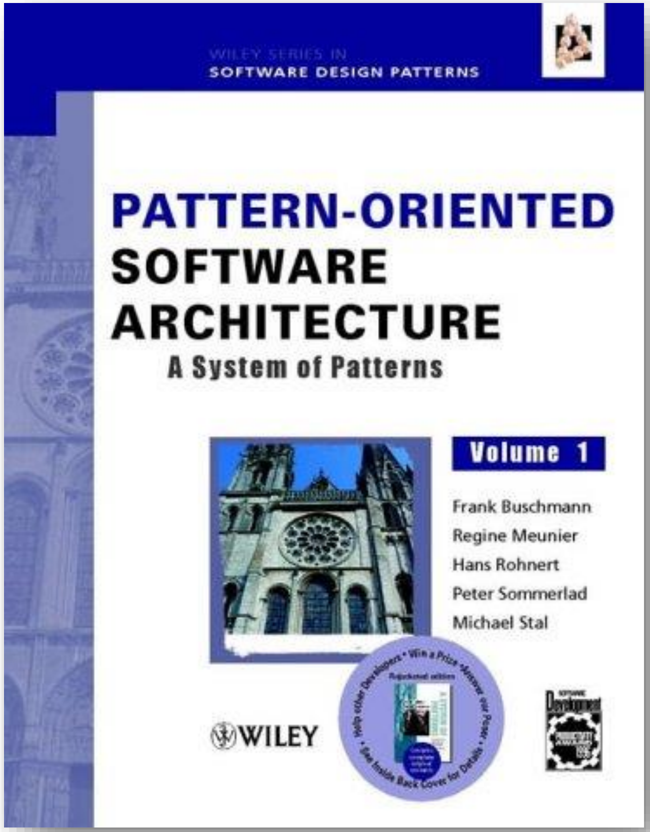
Design patterns

Architecture

Process

- 1 Architecture
- 2 Design
  - 2.1 High level
  - 2.2 Low level

Architectural patterns



Architectural Patterns

- Layers
- Pipes and filters
- Repository
- Client server
- Broker
- MVC
- Microkernel
- Microservices

- A real system is usually influenced by many architectural patterns / styles

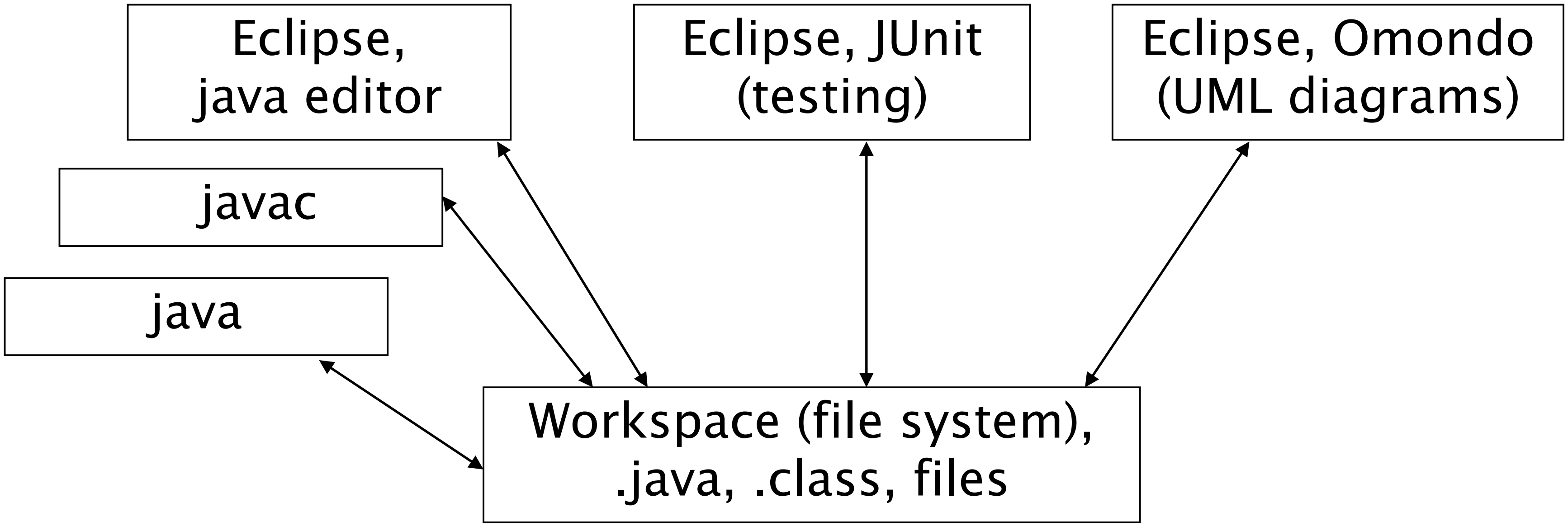
The repository style

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

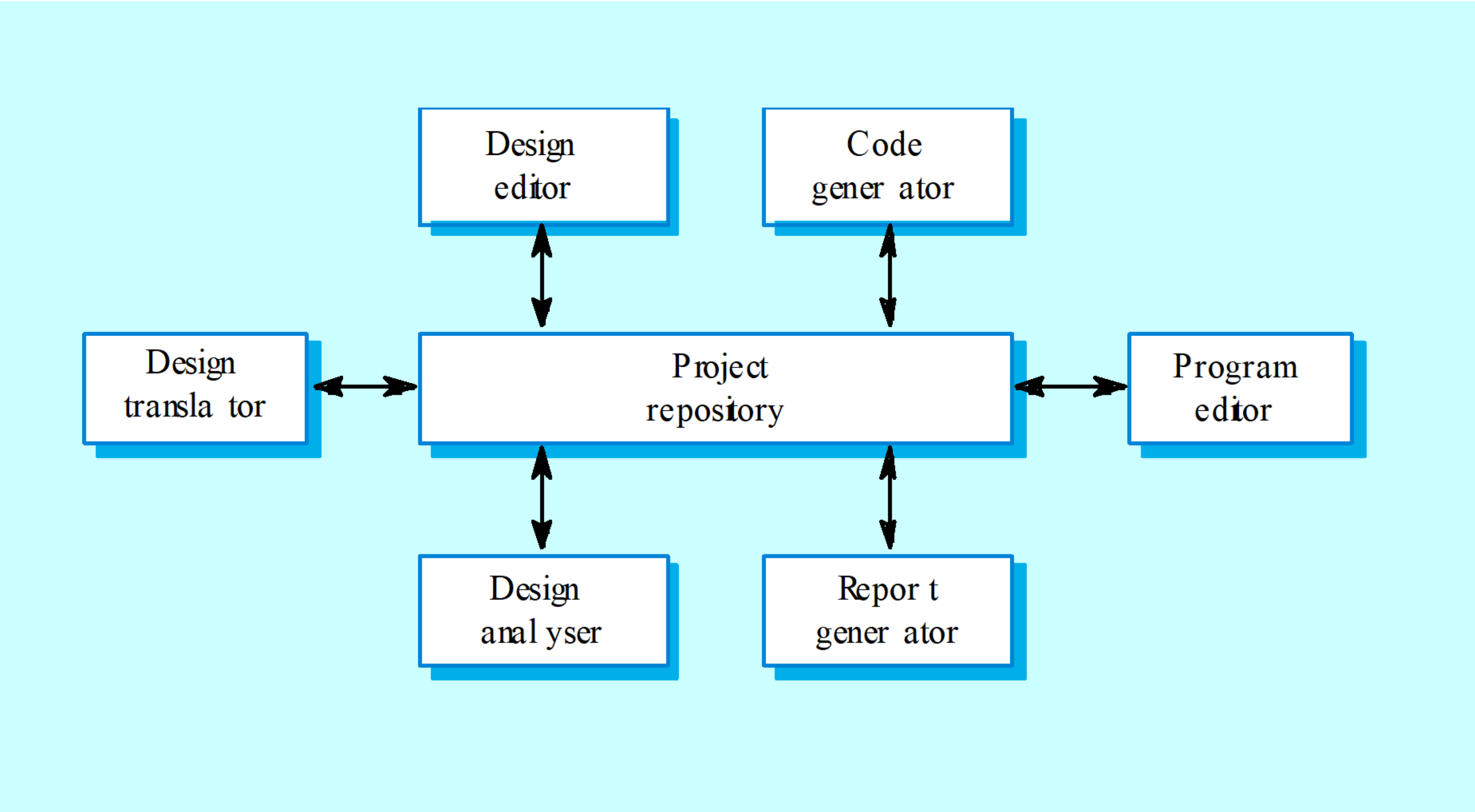
The repository style

- Subsystems exchange data only through the repository, by reading/writing files
  - No direct exchange through API
- The data model is the same for all subsystems
- The repository takes care (in the same way for all subsystems) for common services: backup, security, ..

Eclipse and plugins



IDE toolset architecture



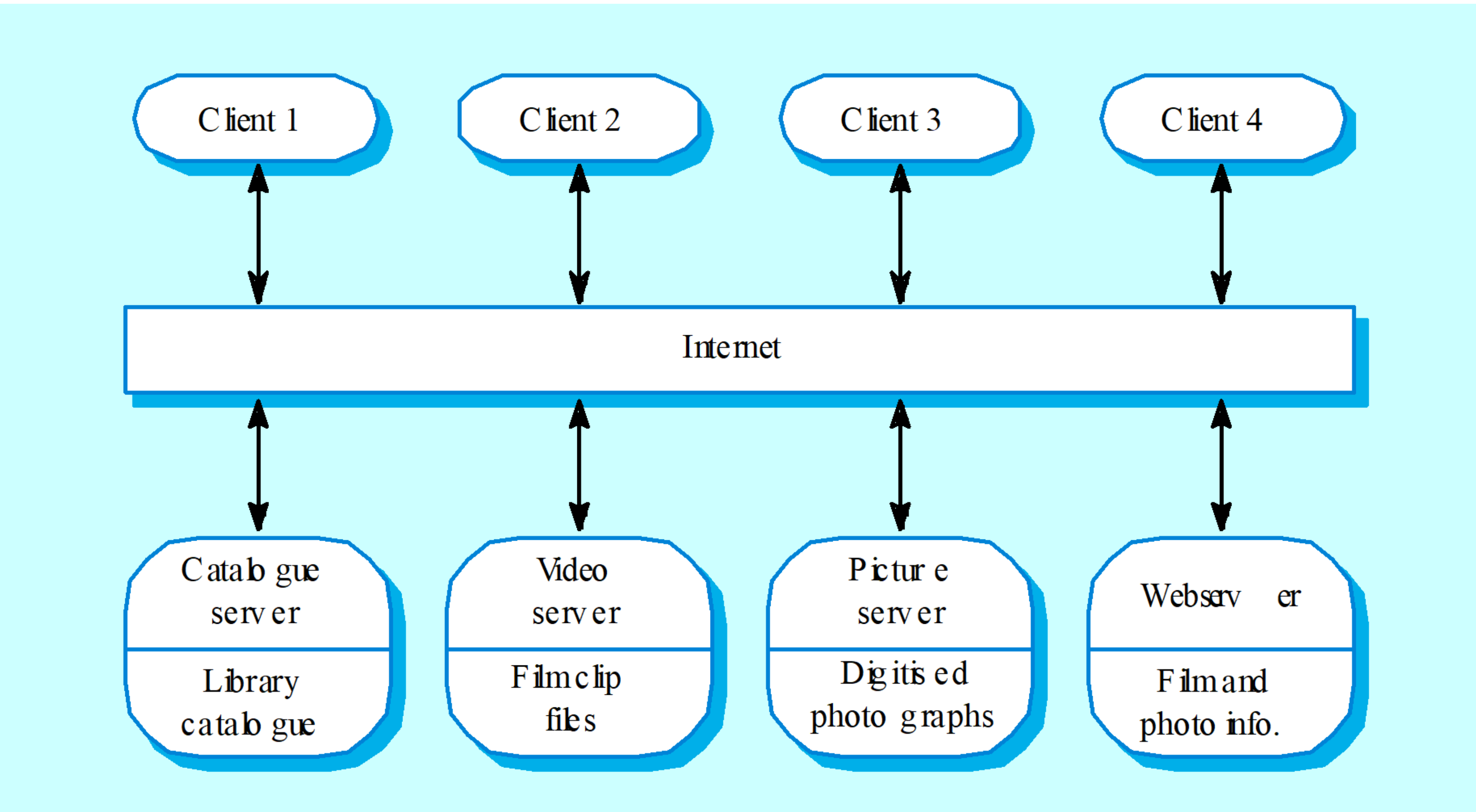
Repository style characteristics

- Advantages
  - Efficient way to share large amounts of data;
  - Sub-systems need not be concerned with how data is produced
  - Centralised management e.g. backup, security
  - Sharing model is published as the repository schema.
- Disadvantages
  - Sub-systems must agree on a repository data model. Inevitably a compromise;
  - Data evolution is difficult and expensive;
  - No scope for specific management policies;
  - Difficult to distribute efficiently.

Client-server model

- Distributed system model which shows how data and processing is distributed across a range of components.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

Film and picture library



Client–server characteristics

- Advantages
  - Distribution of data is straightforward;
  - Makes effective use of networked systems. May require cheaper hardware;
  - Easy to add new servers or upgrade existing servers.
- Disadvantages
  - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
  - Redundant management in each server;
  - No central register of names and services – it may be hard to find out what servers and services are available.

Abstract machine (layered) model

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Constraint: layer uses only services from adjacent layer
- Advantages
  - In design: each layer is about a problem (separation of concerns)
  - In evolution: when a layer interface changes, only the adjacent layer is affected.
- Problems
  - Sometimes artificial to structure systems in this way.

ISO Osi model

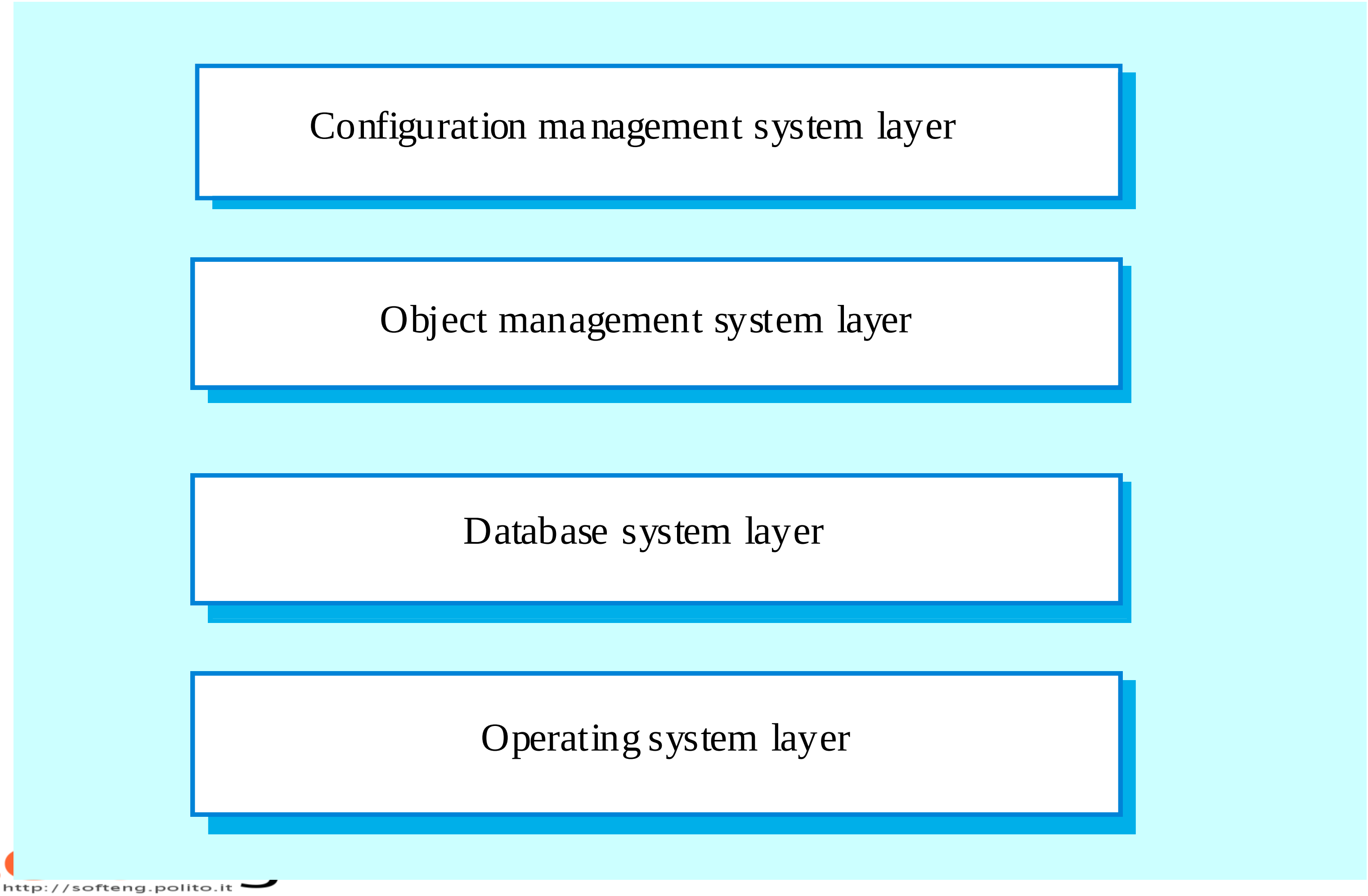
7 application
6 presentation
5 session
4 transport
3 network
2 data link
1 physical

3 tier architecture

Presentation
Application logic
Data (drivers)

Presentation
Application logic
Data (DBMS)

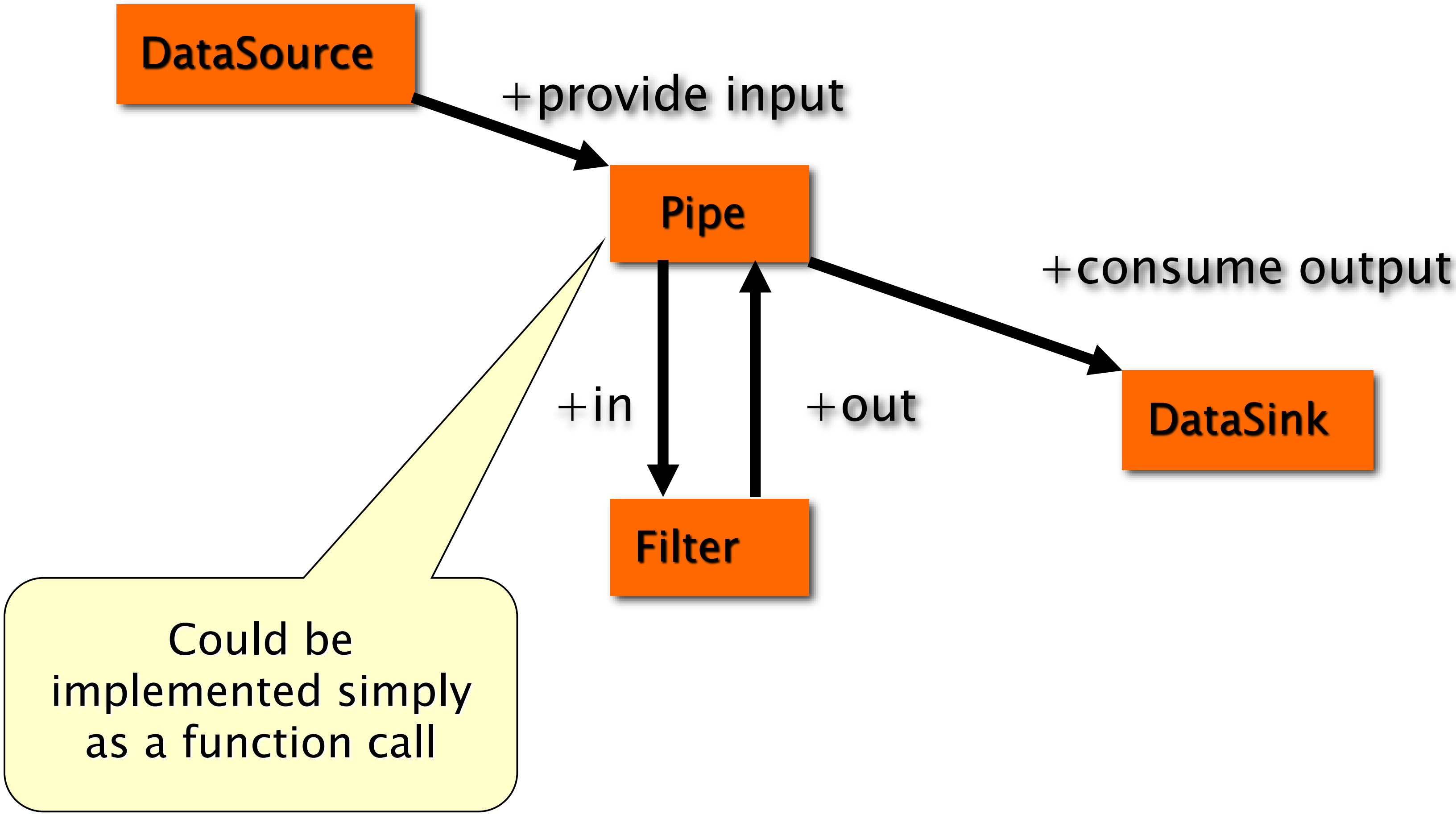
Version management system



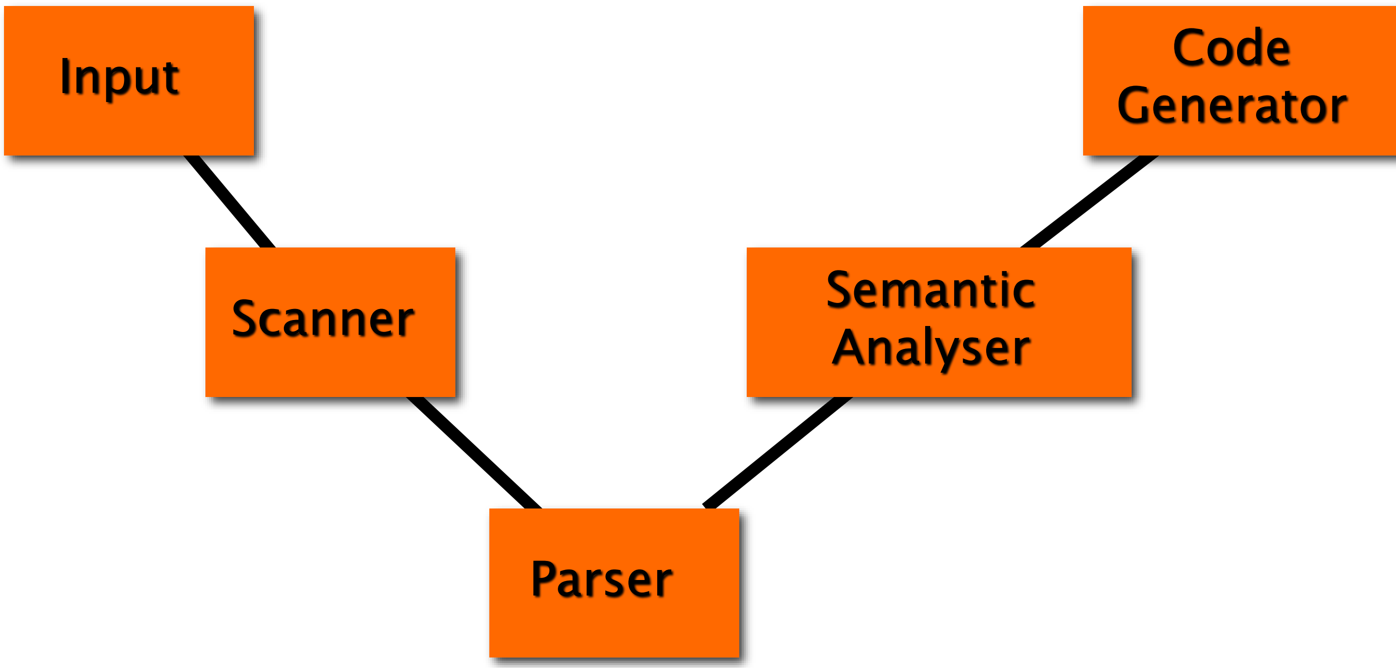
Pipes & Filters

- Context
  - We need to process data streams according to several steps
- Problem
  - Must be possible recombining steps
  - Non-adjacent steps do not share info
  - The user storing data after each step may result into errors and garbage

Pipes & Filters

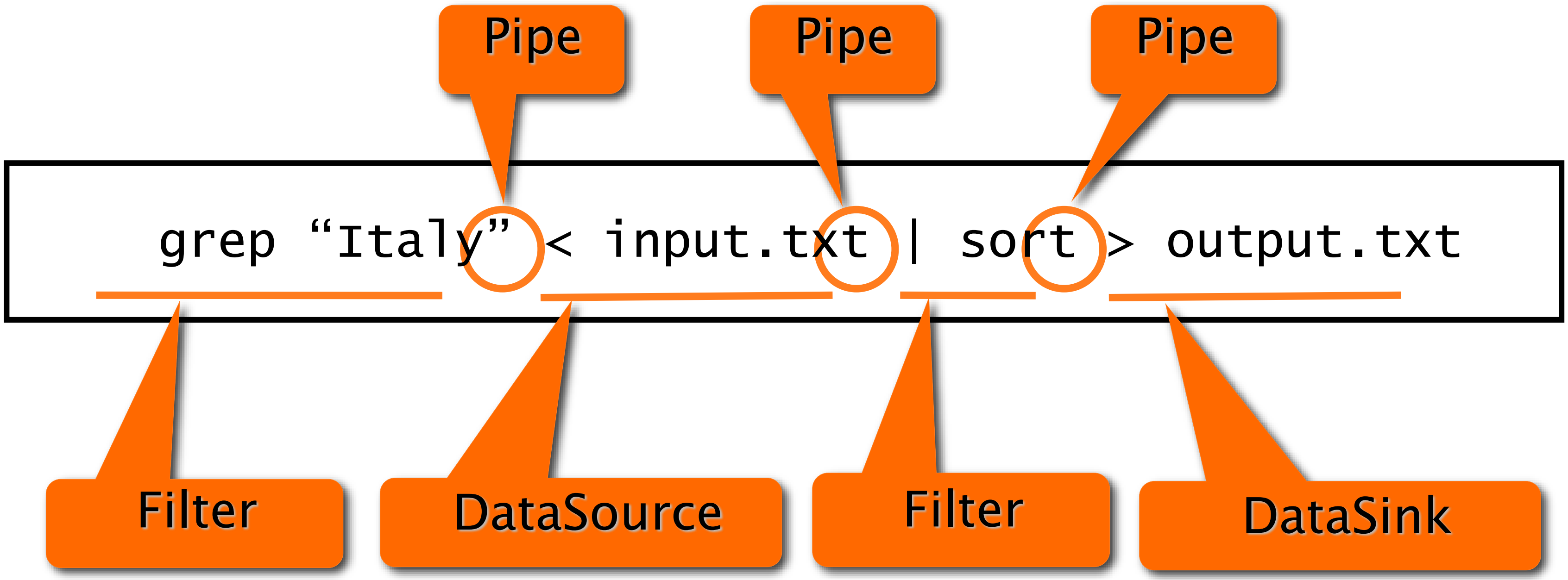


Pipes & Filters Example

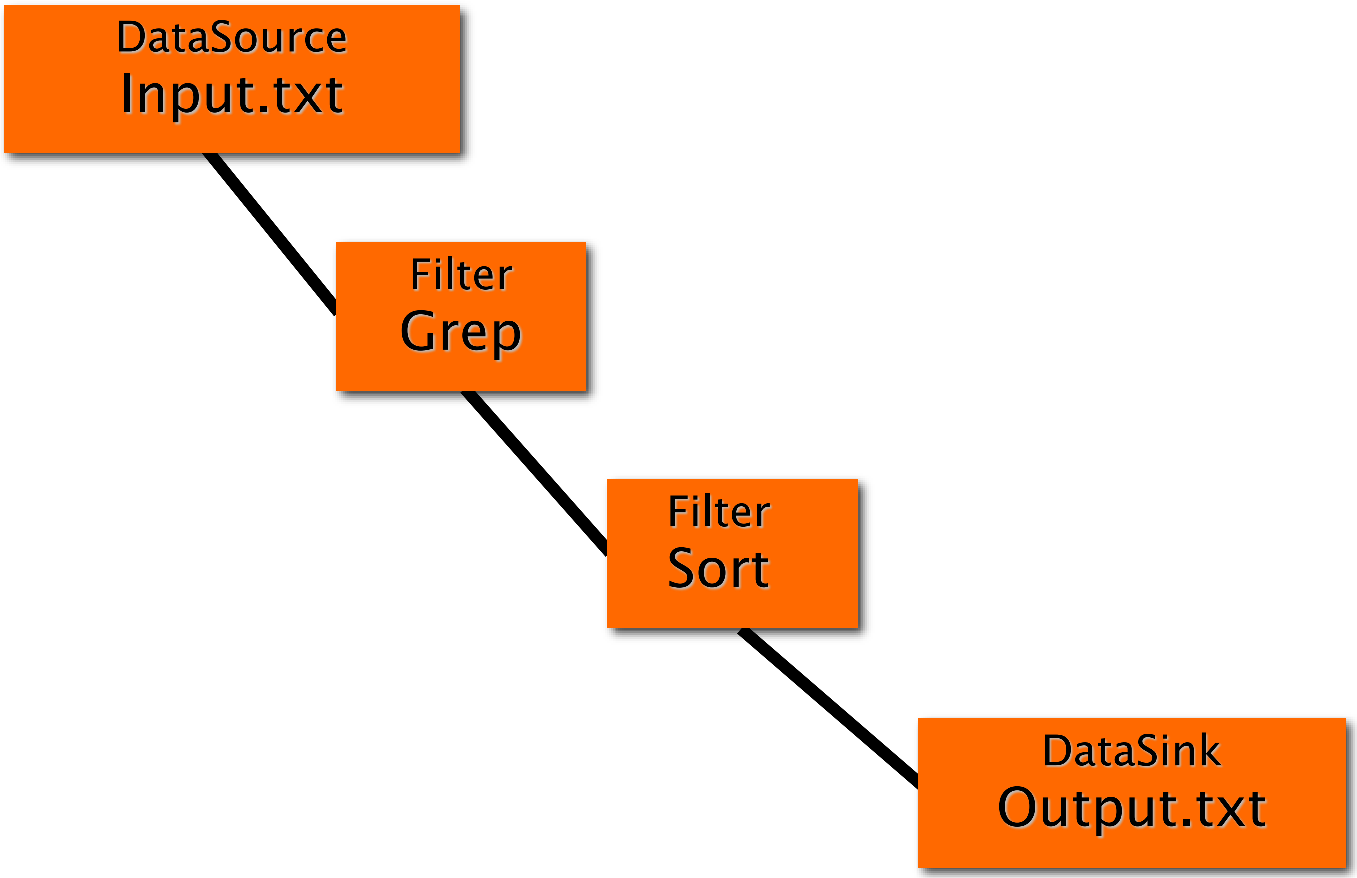


Pipes & Filter Example

Unix shell commands



Pipes & Filter Example



Pipes & Filter Example

Input.txt

```
Rome, Italy
Milan, Italy
Turin, Italy
Paris, France
Marseille, France
Brussels, Belgium
Munich, Germany
Berlin, Germany
```

Pipes & Filter Example

grep “Italy” < Input.txt

```
Rome, Italy
Milan, Italy
Turin, Italy
Paris, France
Marseille, France
Brussels, Belgium
Munich, Germany
Berlin, Germany
```

Pipes & Filter Example

| sort > output.txt

```
Rome, Italy
Milan, Italy
Turin, Italy
```

Pipes & Filter Example

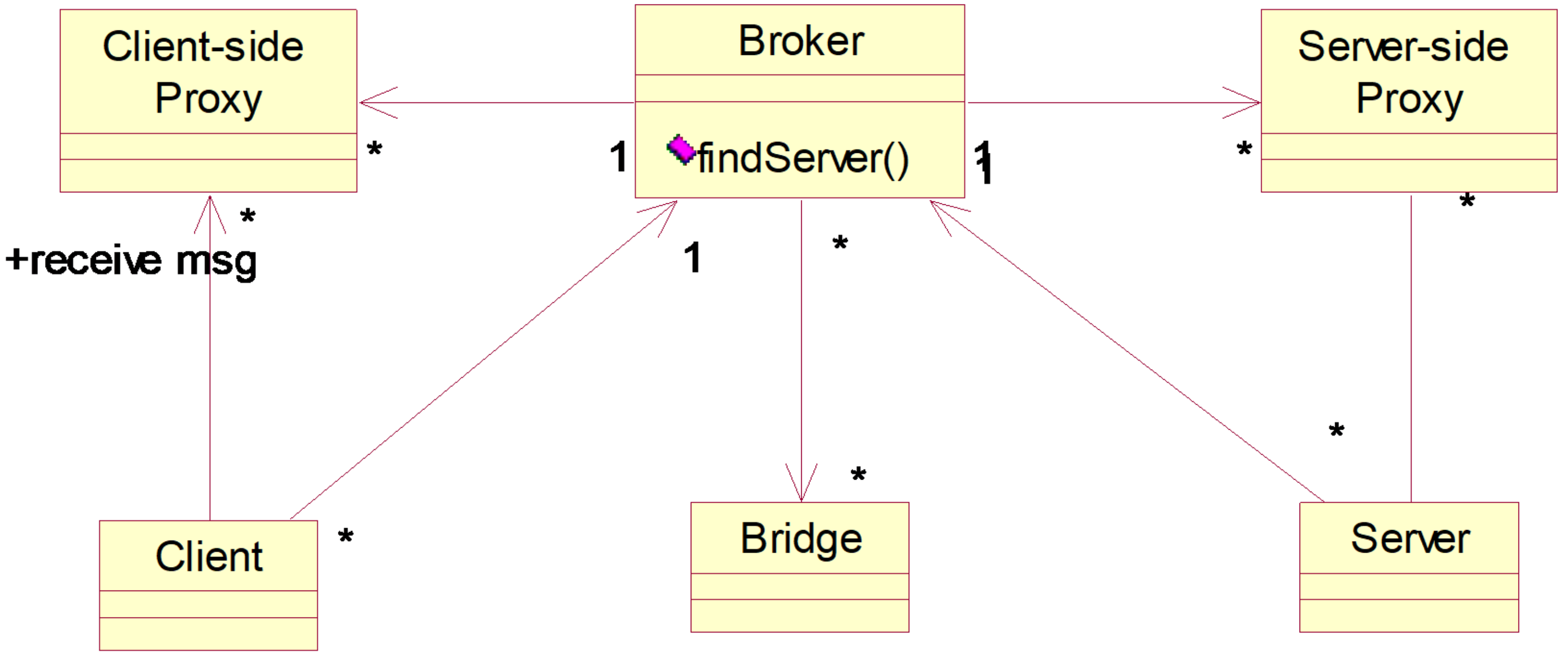
Output.txt

```
Milan, Italy
Rome, Italy
Turin, Italy
```

Broker

- Context
  - Environment with distributed and possibly heterogeneous components
- Problem
  - Components should be able to access others
    - Remotely
    - Location independently
  - Components can be changed at run-time
  - Users should not see too many details

Broker

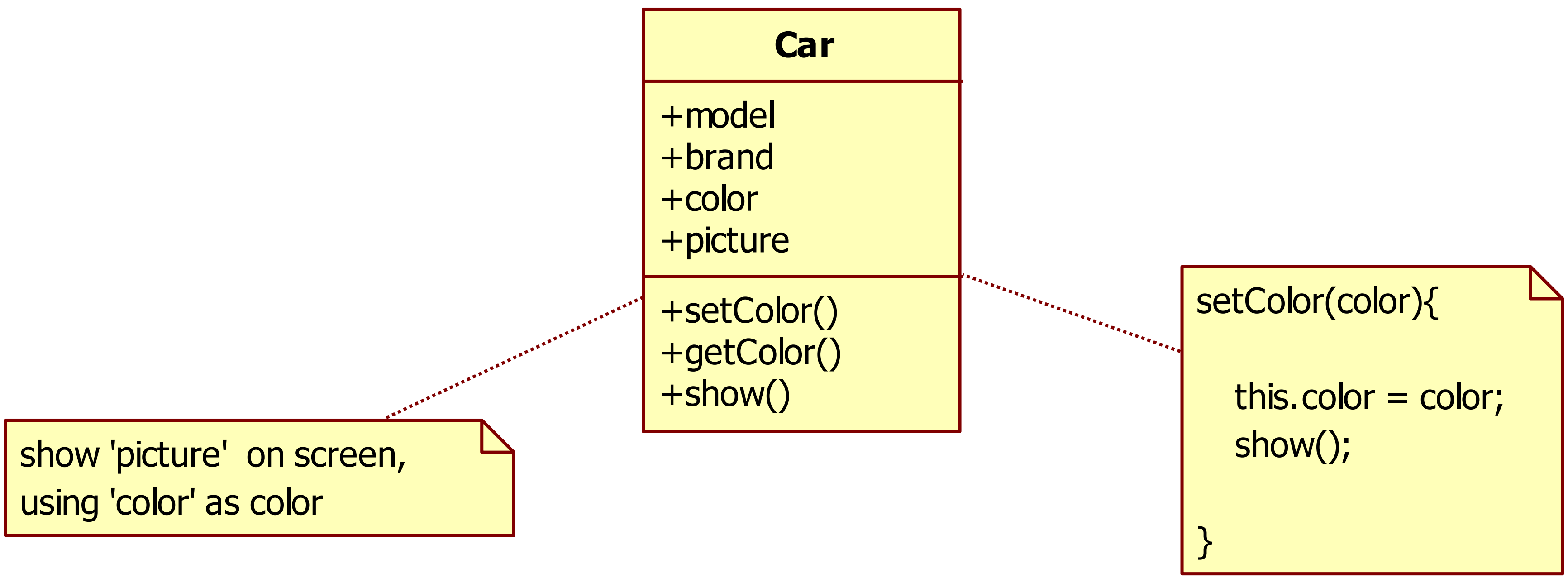


Client-side Proxy
Broker
findServer()
Server-side Proxy
+receive msg
Client
Bridge
Server
*
1
1
1
*
*
*
1
*
*
*
*
*

MVC – Problem

- Show data to user, manage changes to data
  - Option1: one class
  - Option2: MVC pattern

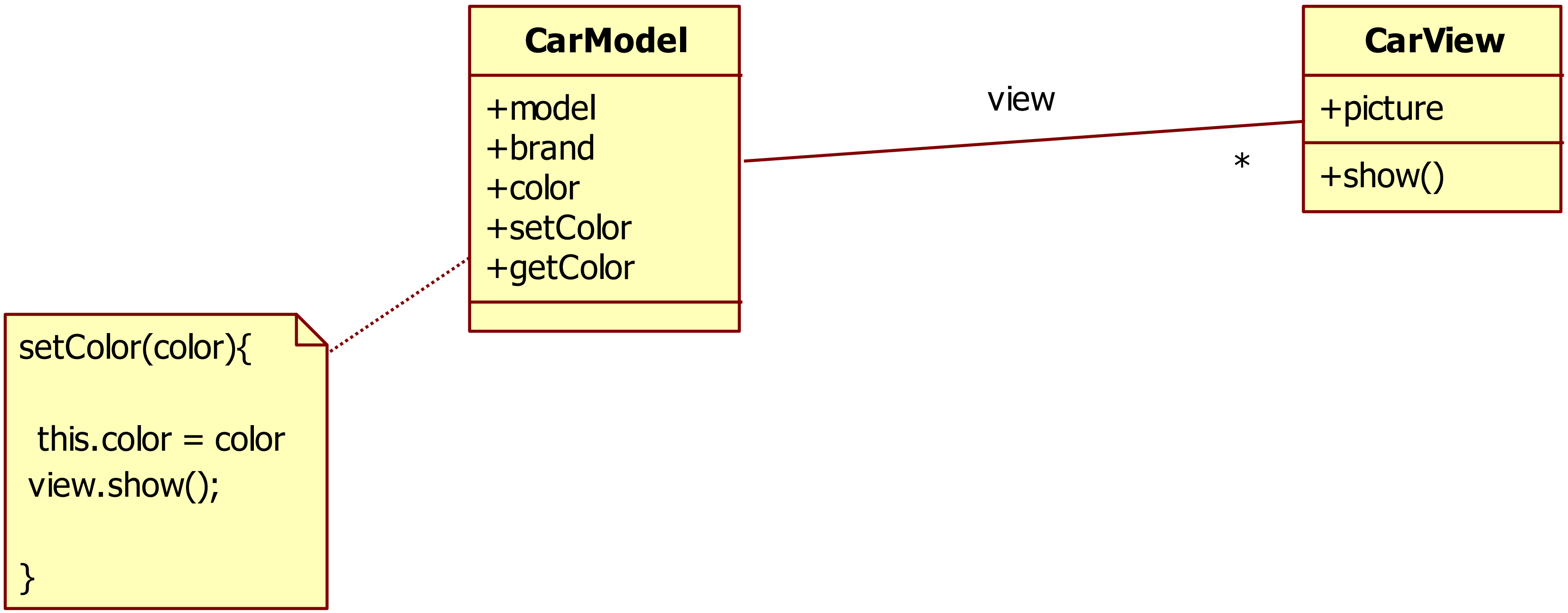
Option1



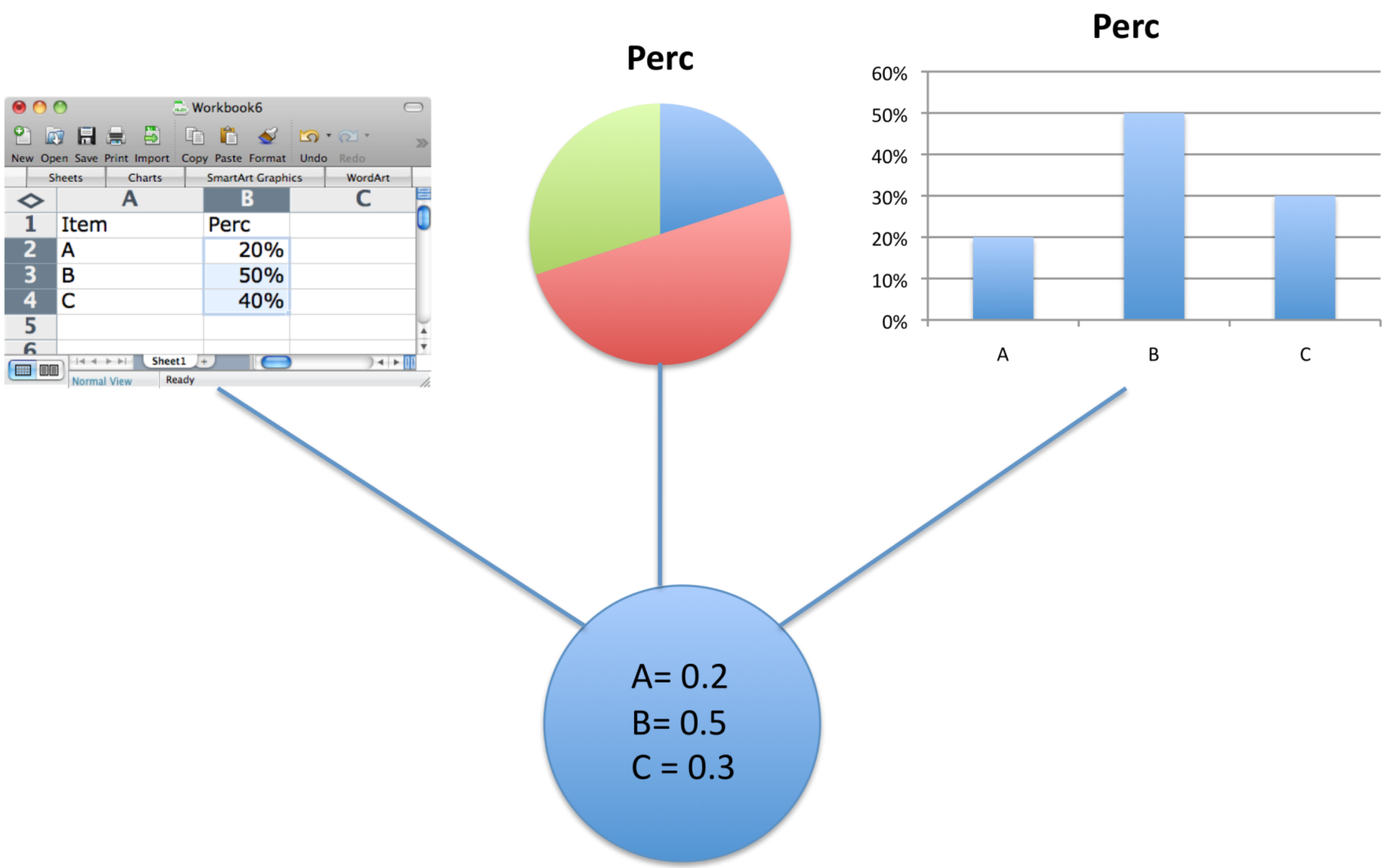
Car
+model
+brand
+color
+picture
+setColor()
+getColor()
+show()
show 'picture' on screen,
using 'color' as color
setColor(color){
this.color = color;
show();
}

Option1

- Pro
  - Easy
- Con
  - What if two (three..) pictures?


CarModel
+model
+brand
+color
+setColor
+getColor
view
*
CarView
+picture
+show()
setColor(color){
this.color = color
view.show();
}

Another case



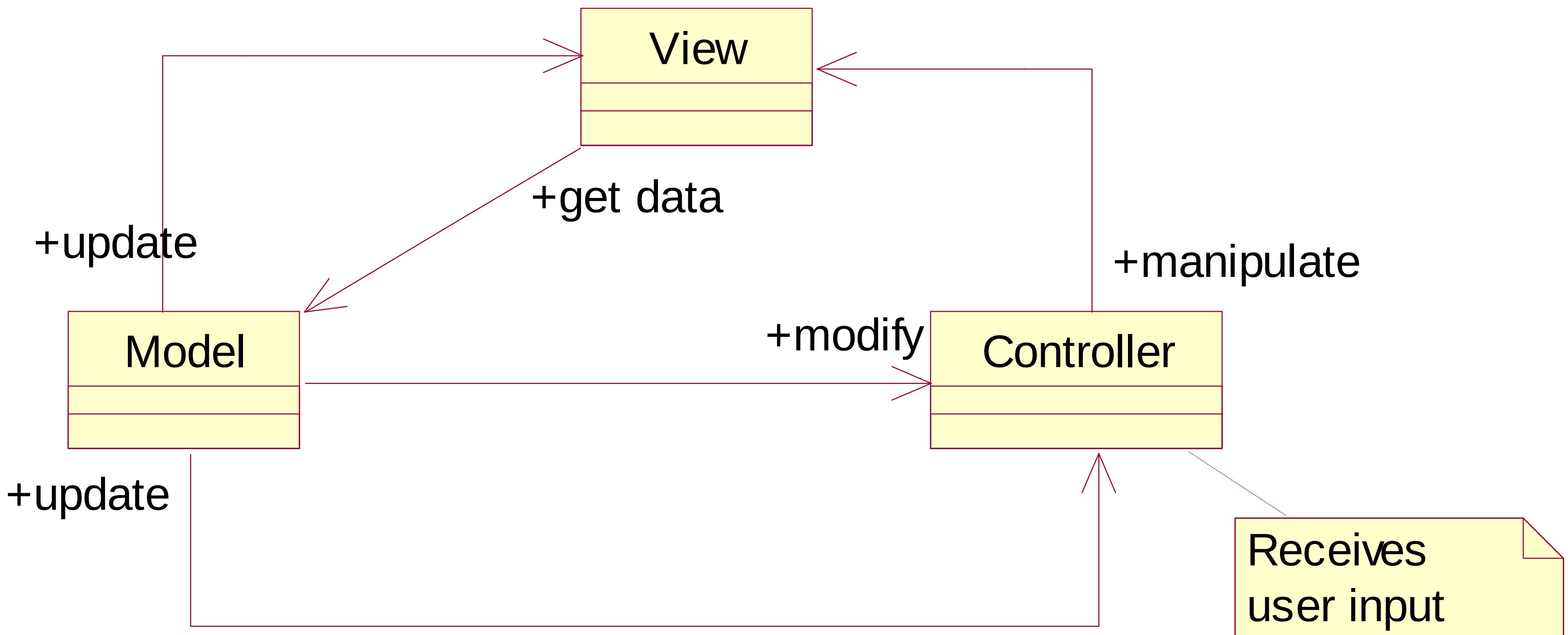
MVC

- Context
  - Interactive applications with flexible HCI
- Problem
  - The same information is presented in different ways/windows
  - Windows must present consistent data
  - Data changes
- Goal (product property)
  - Maintainability, portability

MVC

- Model
  - Responsible to manage state (interfaces with DB or file system)
- View
  - Responsible to render on UI
- Controller
  - Responsible to handle events from UI

MVC



View
+get data
+update
+manipulate
Model
+modify
Controller
+update
Receives
user input

- Pros
  - Separation of responsibilities
    - Many different views possible
    - Model and view can evolve independently (maintainability)
- Cons
  - More complexity (less performance)

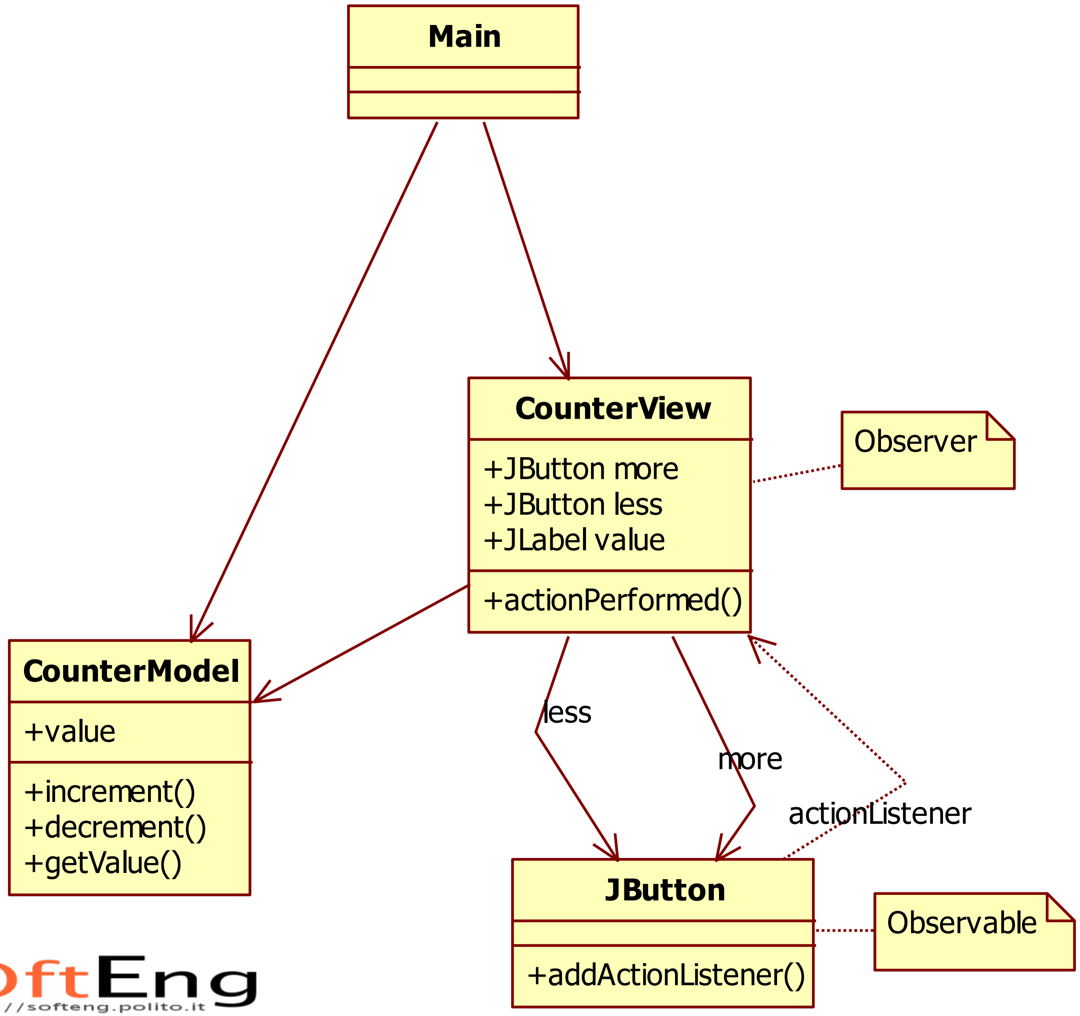
Execution flow

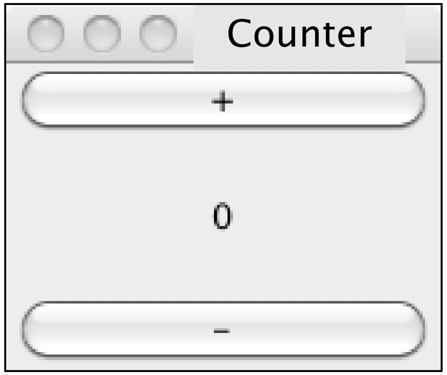
- There is no predefined order of execution
  - Operations are performed in response to external events (e.g. mouse click)
  - Event handling is serialized
  - To execute operations in parallel, threads must be used
  - Method main in GUIs has the only goal of instantiating the graphical elements

MVC implementations

- Given the high level idea
- Different implementations happen in different environments
  - Java
  - C#
  - Android
  - IoS

MVC in Java (MV)

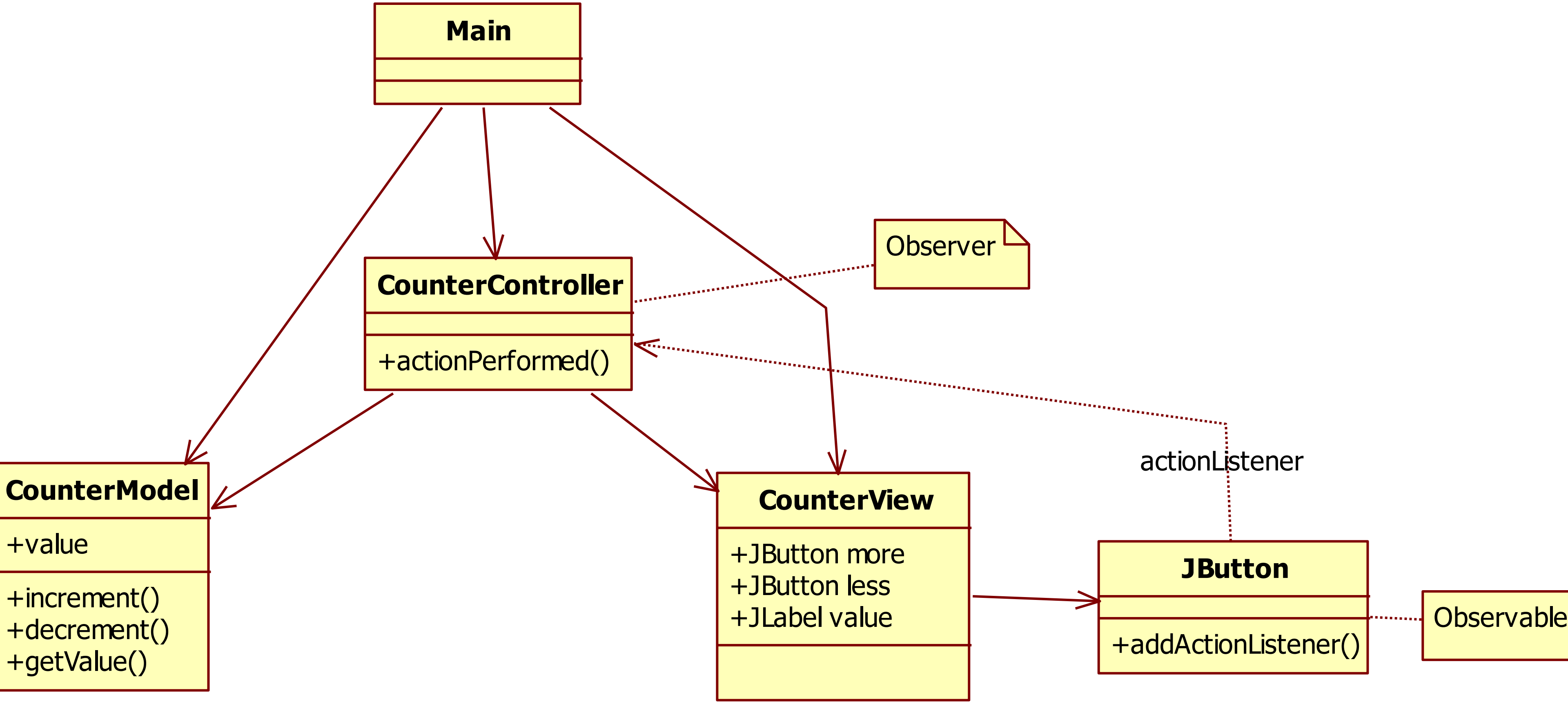




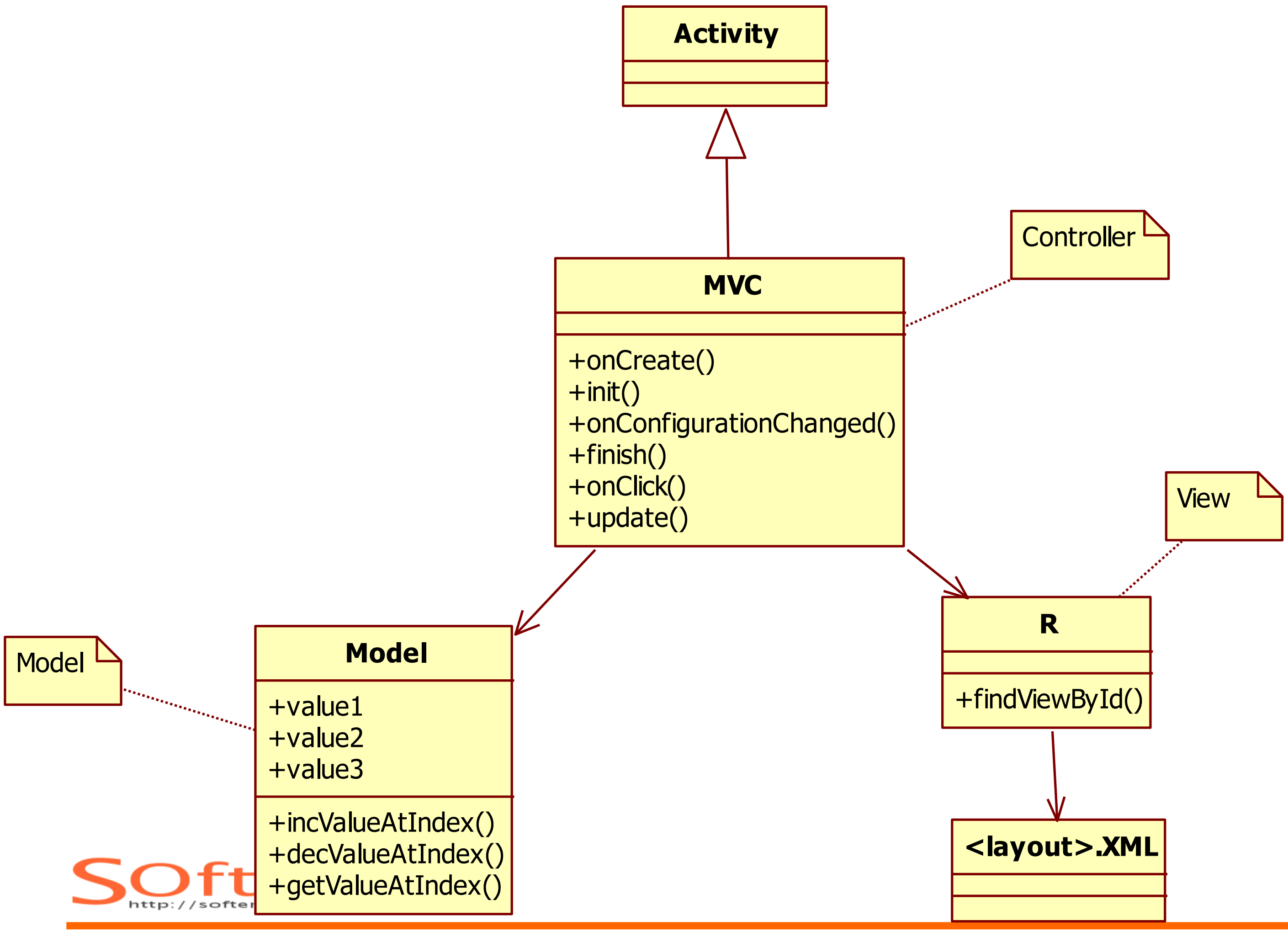
```java
class CounterView implements ActionListener {
    private CounterModel model;
    private JLabel valueLabel;
    private JButton more;
    private JButton less;

    public CounterView(CounterModel m, JPanel panel) {
    model = m;

    int value = model.getValue();
    panel.add(new JLabel("counter"));
    panel.add(valueLabel= new JLabel(Integer.toString(value)));
    more = new JButton("more");
    less = new JButton("less");
    panel.add(more);
    panel.add(less);
    more.addActionListener(this);
    less.addActionListener(this);
    }

    public void update(){
    valueLabel.setText(Integer.toString(model.getValue()));
    }

    public void actionPerformed(ActionEvent arg0) {
      Object o = arg0.getSource();
      if (o== more) model.increment();
      if (o == less) model.decrement();
      update();
    }

    }
```

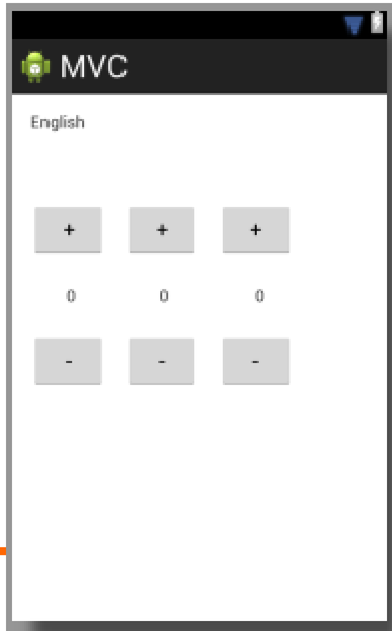
```
public class CounterModel {

  private int value;
  public void increment(){ value++;}
  public void decrement(){ value--;}
  public int getValue(){ return value;}


}
public class MainMV {

public static void main(String[] args) {


JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.add(new JLabel("here"));
frame.setContentPane(panel);
frame.setSize(300,100);
frame.setVisible(true);
frame.repaint();

CounterModel m = new CounterModel();
CounterView v = new CounterView(m, panel);
}
```

MVC in Java (MVC)

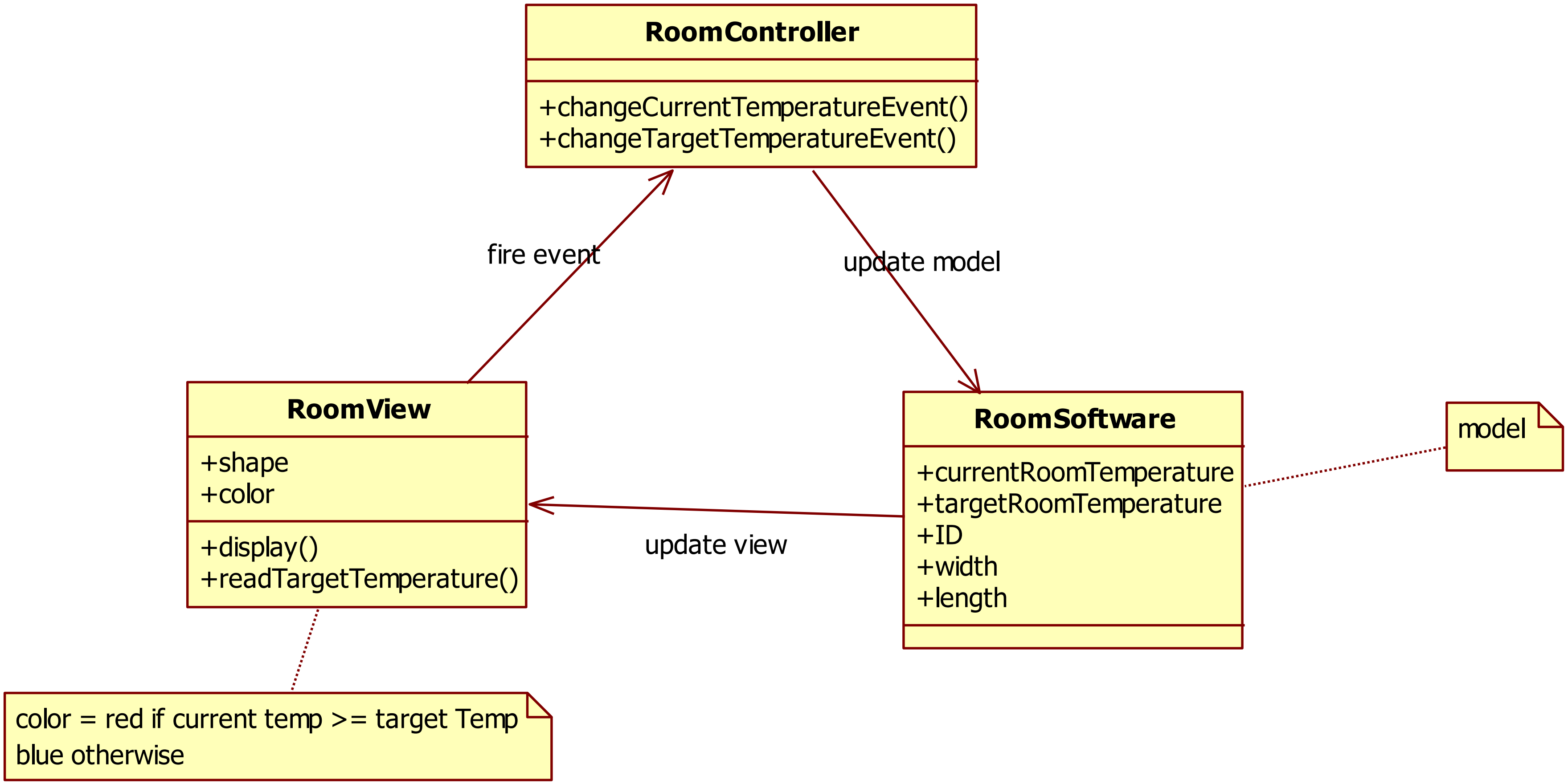


MVC in Android





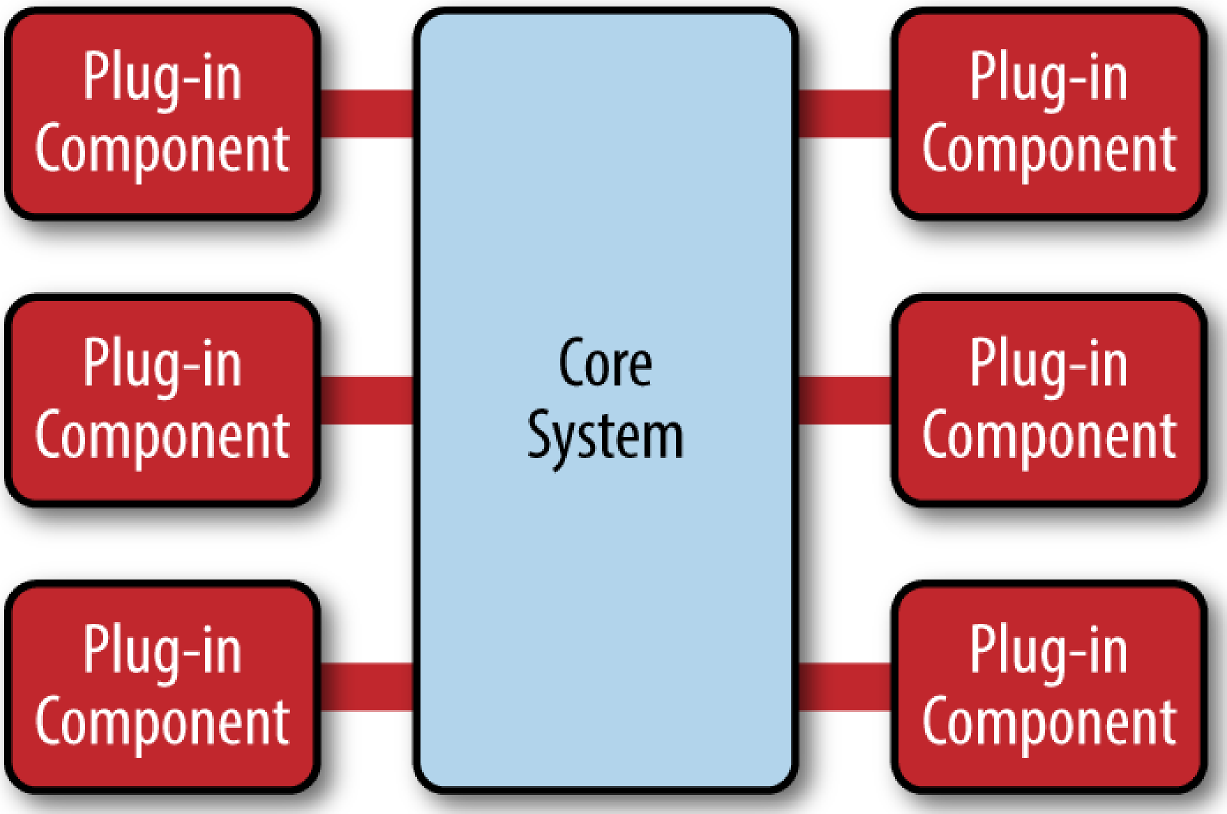
In heating control system



Microkernel

- Context
  - Several APIs insisting on a common core
- Problem
  - HW and SW evolve continuously and independently
  - The platform should be:
    - Portable
    - Extendable

Microkernel



Microservices

- Microservice = one executable running in its process (real or virtual machine)
- Microservices communicate via http calls, (RESTFul APIs)
- Application made of many communicating microservices
  - Via orchestration
  - Via choreography

Microservices

- Advantages
  - Each MS could use a different technology stack
  - Each MS can be released and deployed independently of others
  - Lower coupling between MSs
- Disadvantages
  - Added complexity
  - Possibly worse time performance

Microservices

- Example (EZShop)

- 3 MS:
  - Management of inventory
  - Management of catalogue
  - Management of sales and customers

Summary

- Architectural patterns deal with overall system structure
- They provide a unique metaphor for the system (e.g. pipe and filters)
- They address specific domains (e.g. distribution or interaction) and system evolvability

Design

Process

- 1 Architecture
- 2 Design
  - 2.1 High level
  - 2.2 Low level

2.1 Design, high level

- Definition of classes
  - From glossary: consider a class for each key entity in glossary
  - From context diagram:
    - Consider a class for each actor = physical device or subsystem
    - Define GUI for each actor = human actor
- Consider design patterns

2.2 Design, low level

- (inside a class or two)
- For each attribute, define type, privacy
- For each method, define return type, number and type of parameters, privacy
- Define setters, getters (if needed)
- For each method, choose algorithms (if needed)
- For each relationship with other class, choose implementation
  - If ‘one’ relationship: reference or key
  - If ‘many’ relationship: array, map, list

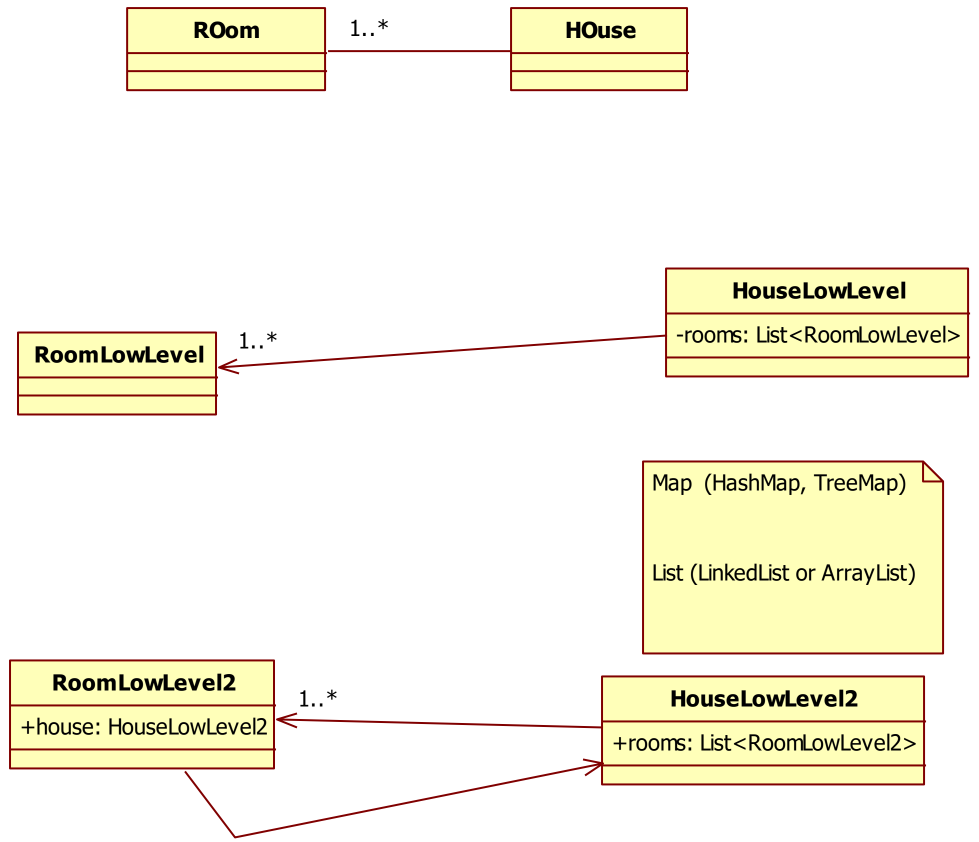
2.2 Design, low level

- (inside a class or two)
- Decide persistency
  - No persistence
  - Yes persistence
    - Serialization (to file, to network)
    - To database
      - Decide framework (hybernate, mybatis, slick …)

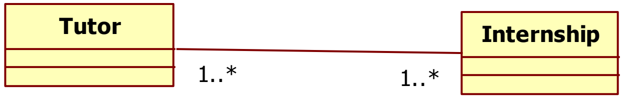
    - On all objects
    - On part of objects

Relationships- low level design

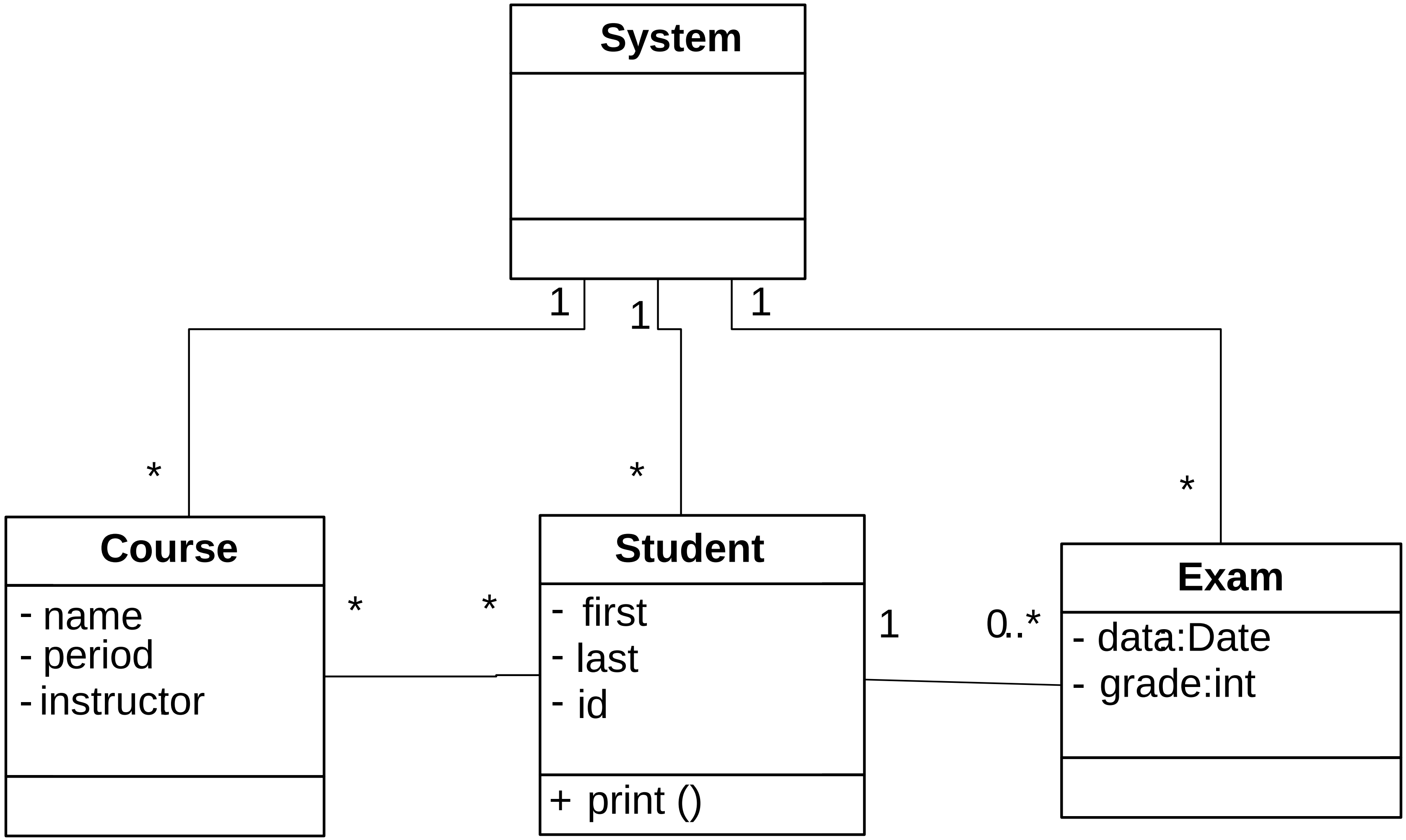
Relationships – 1–1*



Many many

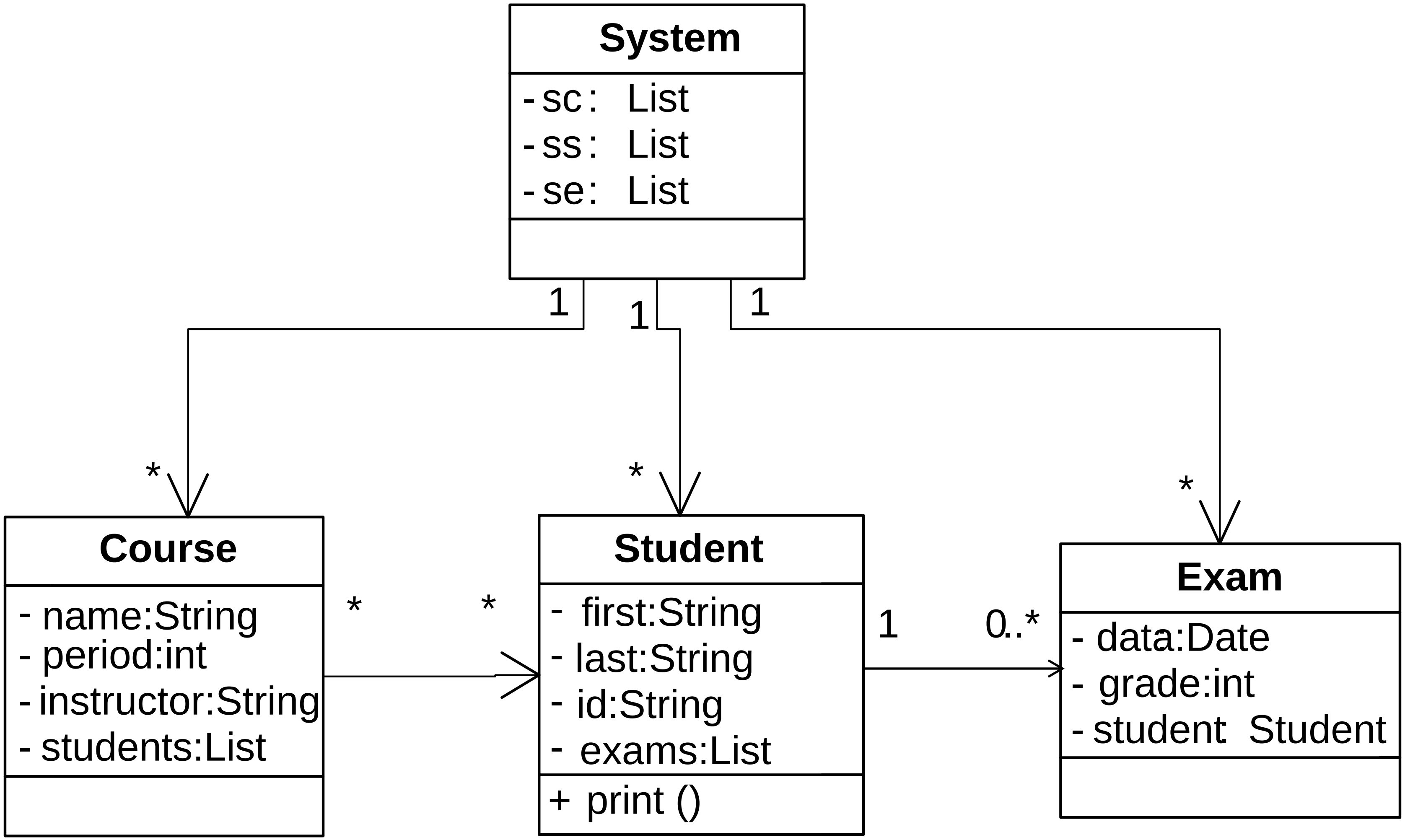


Example – glossary



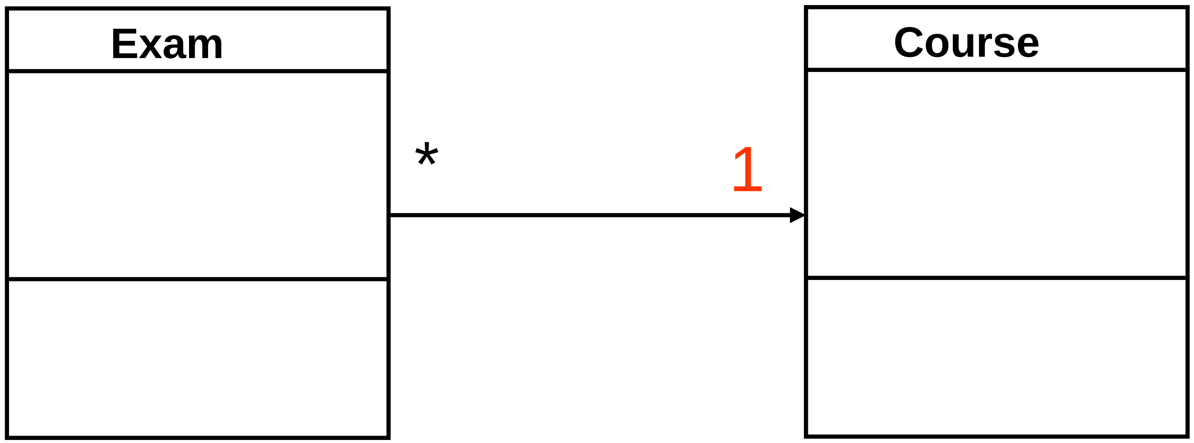
SoftEng
http://softeng.polito.it

Example


System
- sc : List
- ss : List
- se : List
1
1
1
*
*
*
Course
- name:String
- period:int
- instructor:String
- students:List
*
*
Student
- first:String
- last:String
- id:String
- exams:List
+ print ()
1
0..*
Exam
- data:Date
- grade:int
- student : Student

Association :1

- From Exam towards Course

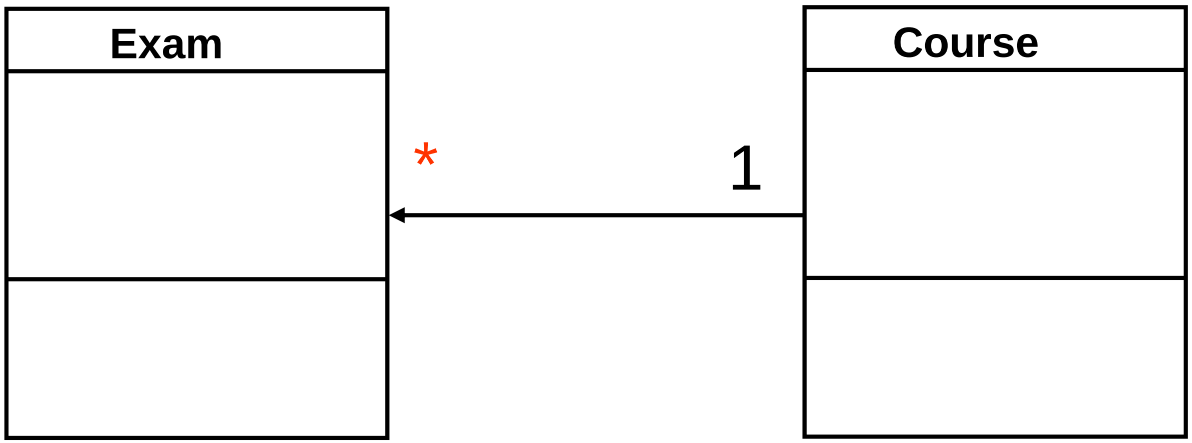



```
Class Exam {
  Course c;
  setCourse(Course c){
    this.c=c;}
}
```

```
Class Course {

}
```

Association :n

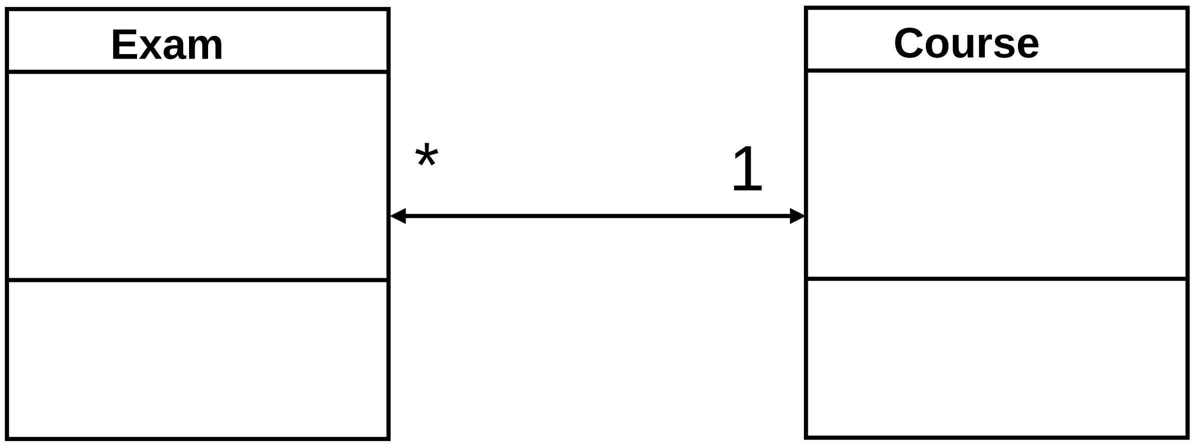
- From Course towards Exams




```
Class Course {
   ArrayList exams;
   Course(){ exams = new ArrayList(); }
   addExam(Exam e){ exams.add(e);}
}
```

Association 1:n

- Both directions

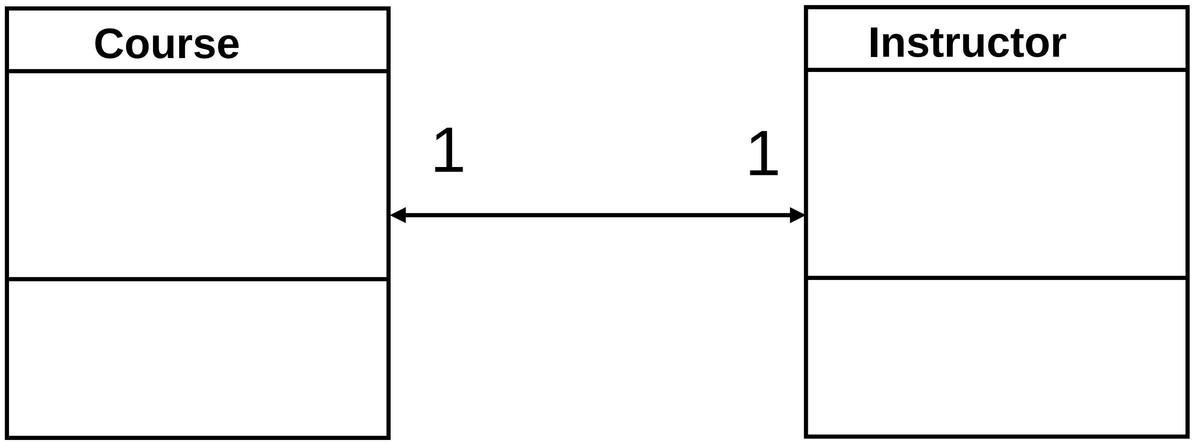



```
Class Exam {
  Course c;
  setCourse(Course c){
    this.c=c;
  }
}
```

```
Class Course {
 ArrayList exams;
 Course(){ exams = new ArrayList ();
}
  addExam(Exam e){ exams.add(e);}
}
```

Association 1:1

- Both directions

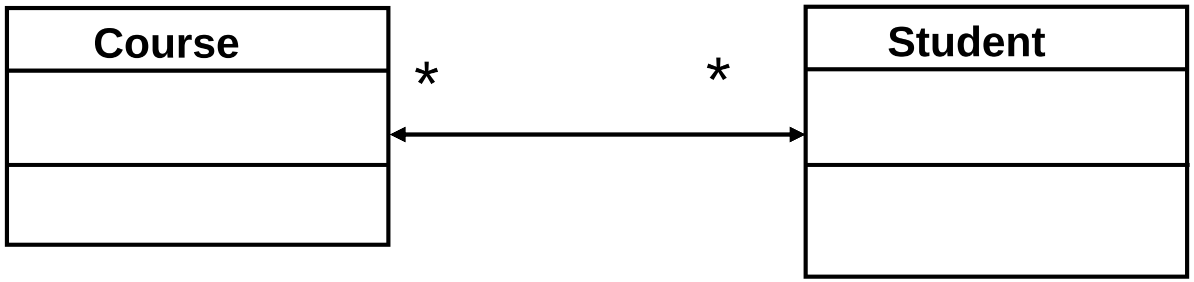



```
Class Course {
    Instructor i;
}
```

```
Class Instructor  {
  Course c;
}
```

Association n:m

- Both directions – option1

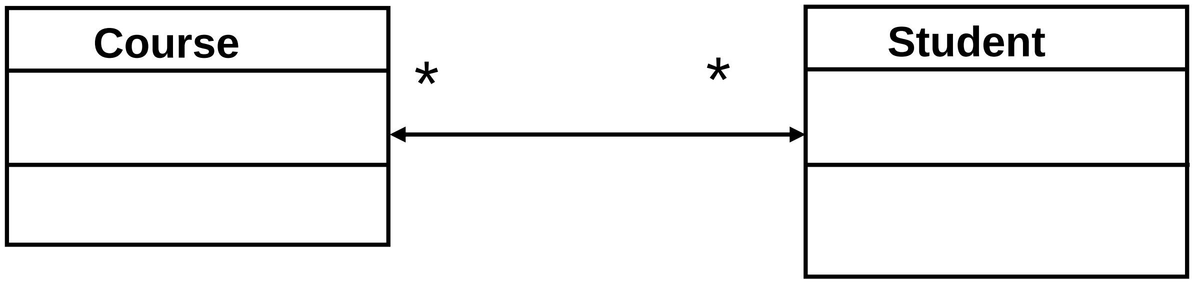



```
Class Course {
  ArrayList students;
  Course(){
     students = new ArrayList();
  }
  addStudent(Student s){
    students.add(s);
  }
}
```

```
Class Student {
  ArrayList courses;
  Students(){
    courses = new ArrayList();
  }
  addCourse(Course c){
    courses.add(c);
  }
}
```

Association n:m

- Both directions – option 2




```
Class Course {
  ArrayList<Pair> p;
}
```

```
Class Student {
  ArrayList<Pair> p;
}
```

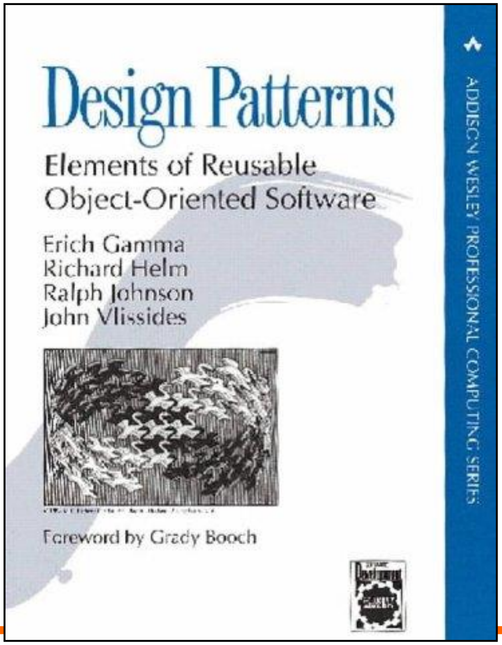
```
Class Pair {
  StudentKey, CourseKey
}
```

Design Patterns (GoF)

- Describe the structure of components
- Most widespread category of pattern
- First category of patterns proposed for software development

Design Patterns (GoF)

- Creational
  - E.g. Abstract Factory, Singleton
- Structural
  - E.g. Façade, Composite
- Behavioral
  - *Class:* e.g. Template Method
  - *Object:* e.g. Observer



Design patterns

- Description of communicating objects and classes that are customized to solve a general design problem in a particular context
- A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design

Description

- Name and classification
- Intent
  - Also known as
- Motivation
- Applicability
- Structure
- Participants
- Collaborations

Description

- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Classification

- Purpose
  - Creational
  - Structural
  - Behavioral
- Scope
  - Class
  - Object

Classification

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	1	1	2
	Object	4	6	10

Pattern selection

- Consider how patterns solve problems
- Scan intent sections
- Study how pattern interrelate
- Study patterns of like purpose
- Examine a cause of redesign
- Consider what should be variable in your design

Using a pattern

- Read through the pattern
- Go back and study
  - Structure
  - Participants
  - Collaborations
- Look at the sample code

Using a pattern

- Choose names for participants
  - Meaningful in the application context
- Define the classes
- Choose operation names
  - Application specific
- Implement operations

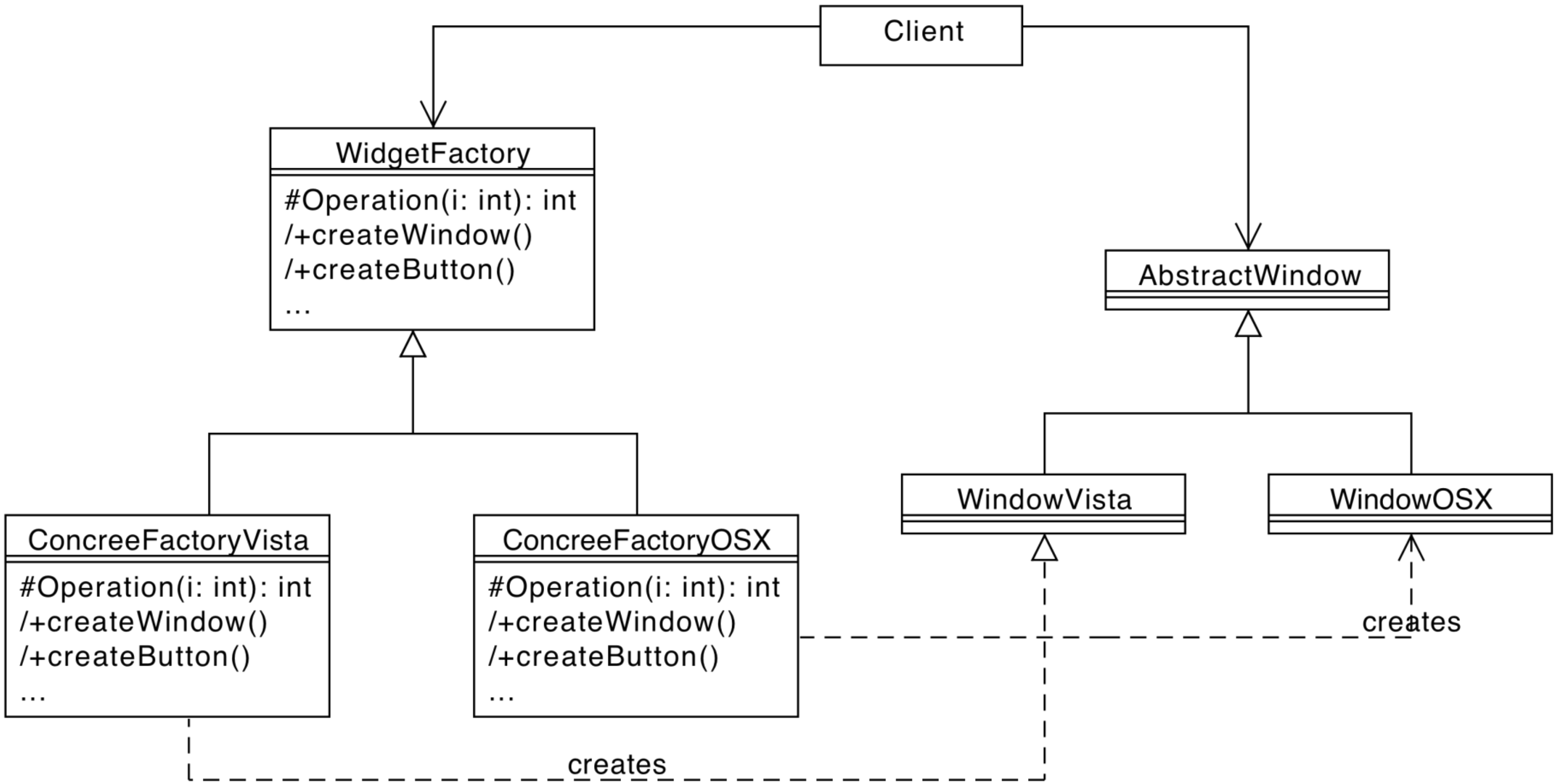
Creational patterns

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

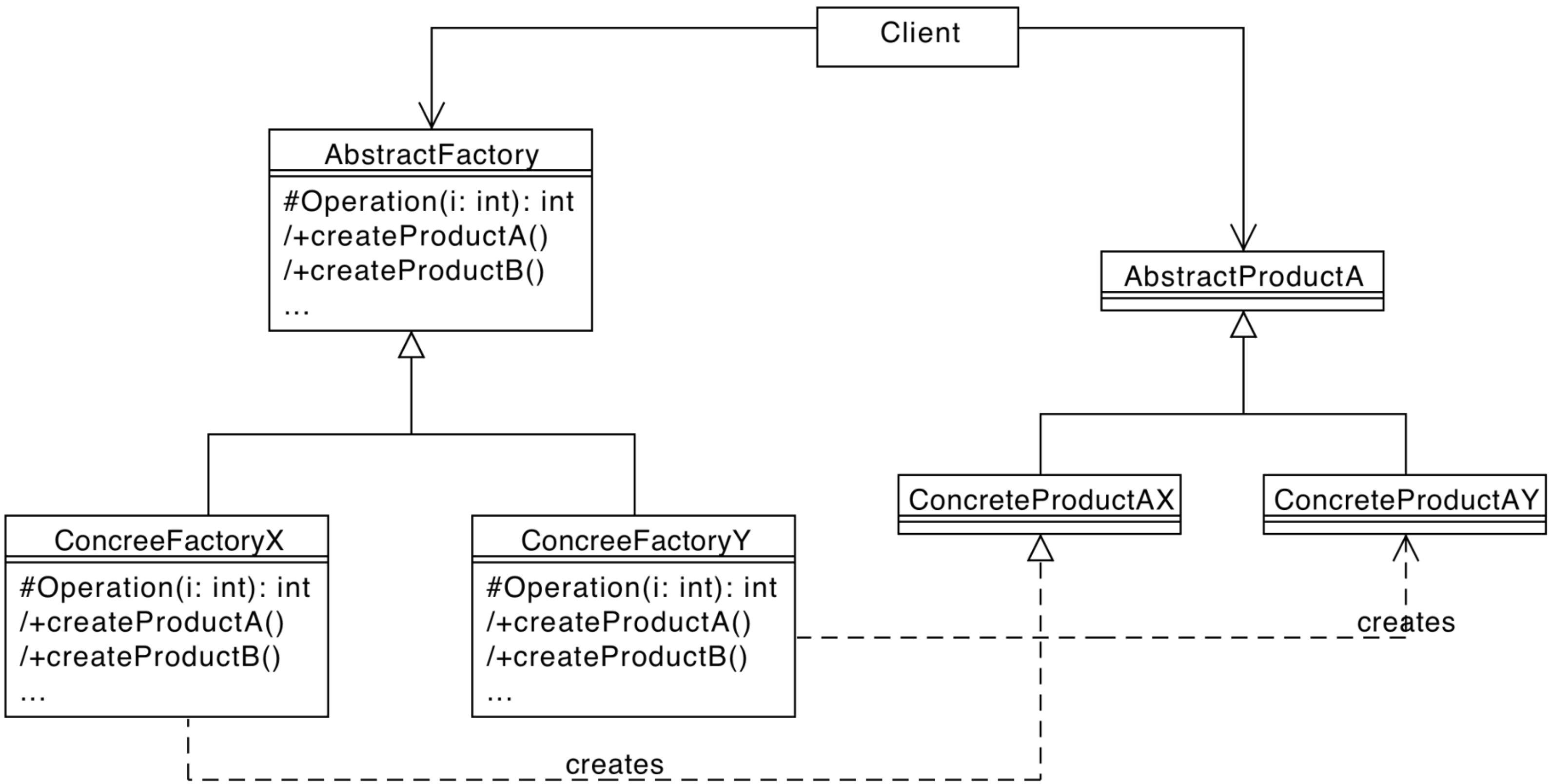
Abstract Factory

- Context
  - A family of related classes can have different implementation details
- Problem
  - The client should not know anything about which variant they are using / creating

Abstract Factory Example



Abstract Factory



Singleton

- Context:
  - A class represents a concept that requires a single instance
- Problem:
  - Clients could use this class in an inappropriate way

Singleton

- Count how many objects in my program
- Class ObjectCounter {
- static boolean new = false;

ObjectCounter () { if new == false then }

static counter = 0; new = true}

else donothing

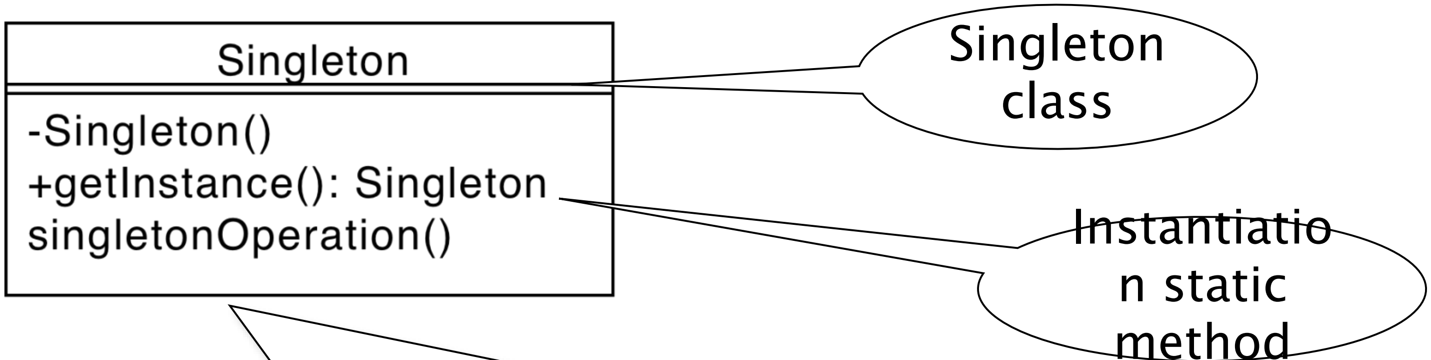
add() {counter++;}

sub() {counter--;} }

Client : ObjectCounter oc = new ObjectCounter();

.... Oc.add(); ... Oc.sub

Singleton




```
private Singleton() { }
private static Singleton instance;
public static Singleton getInstance(){
    if(instance==null)
        instance = new Singleton();
    return instance;
}
```

Structural patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.

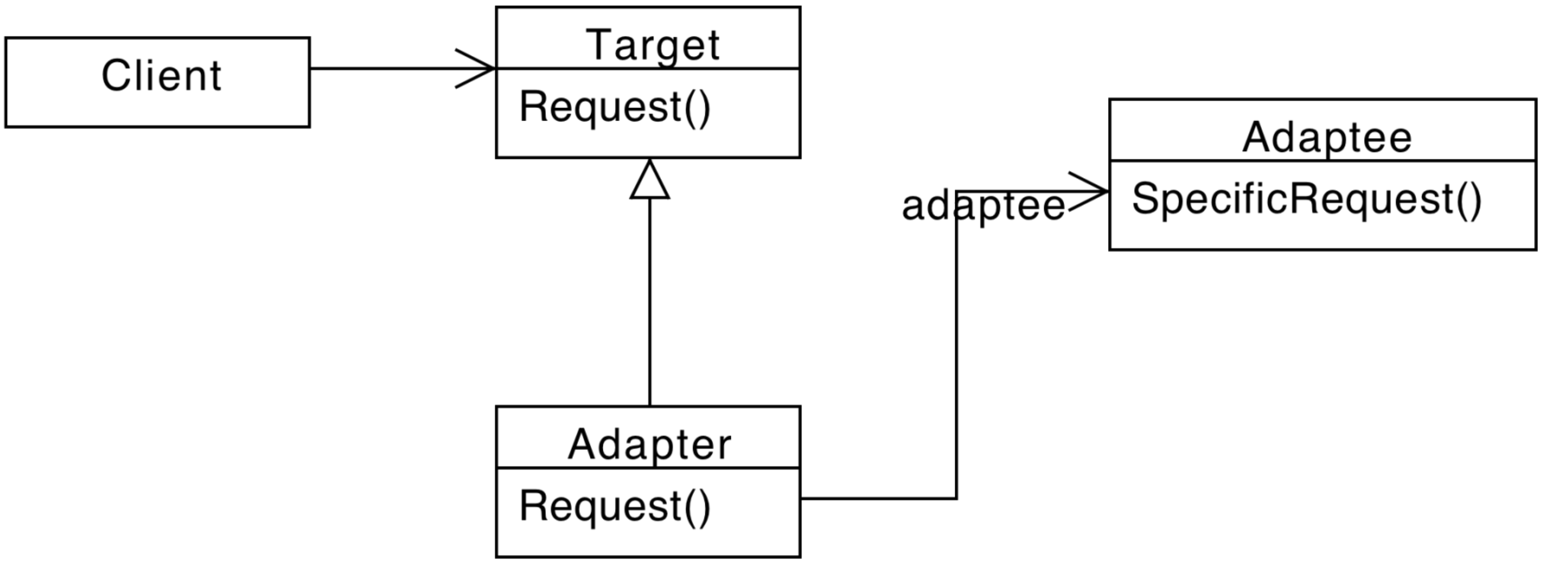
GoF structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

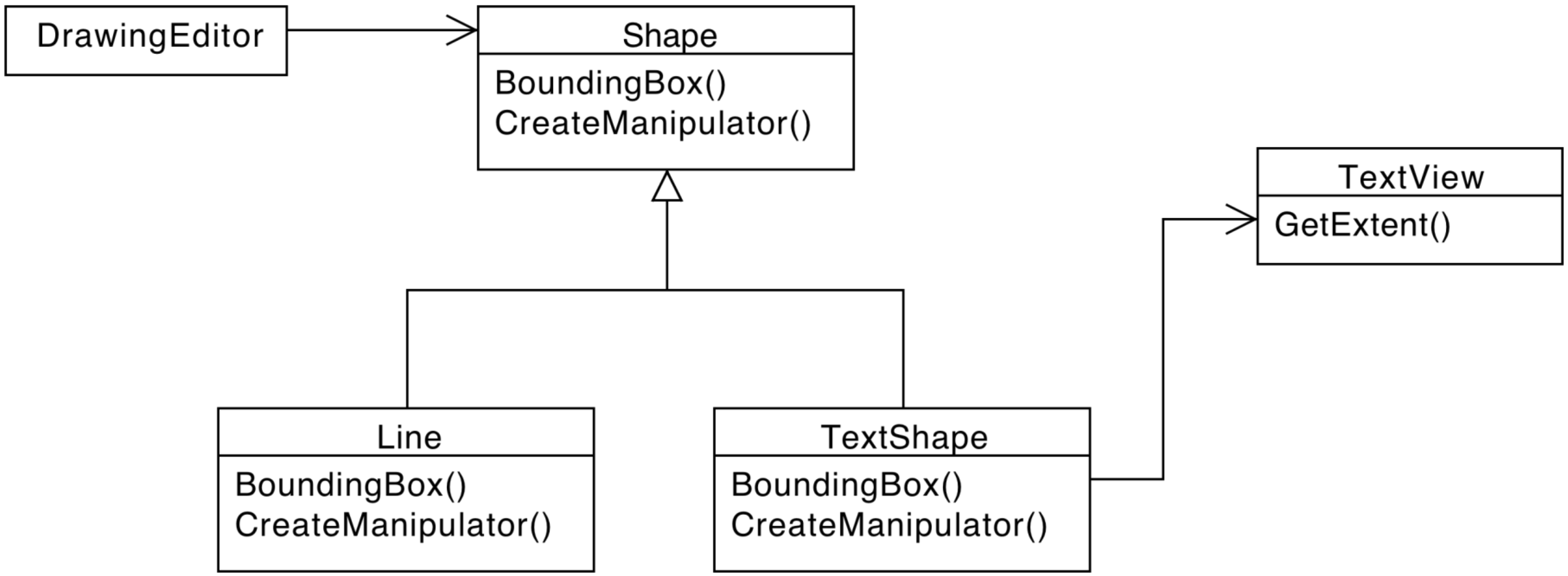
Adapter

- Context:
  - A class provides the required features but its interface is not the one required
- Problem:
  - How is it possible to integrate the class without modifying it
    - Its source code could be not available
    - It is already used as it is somewhere else

Adapter


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()

Adapter example



Java Listener Adapter

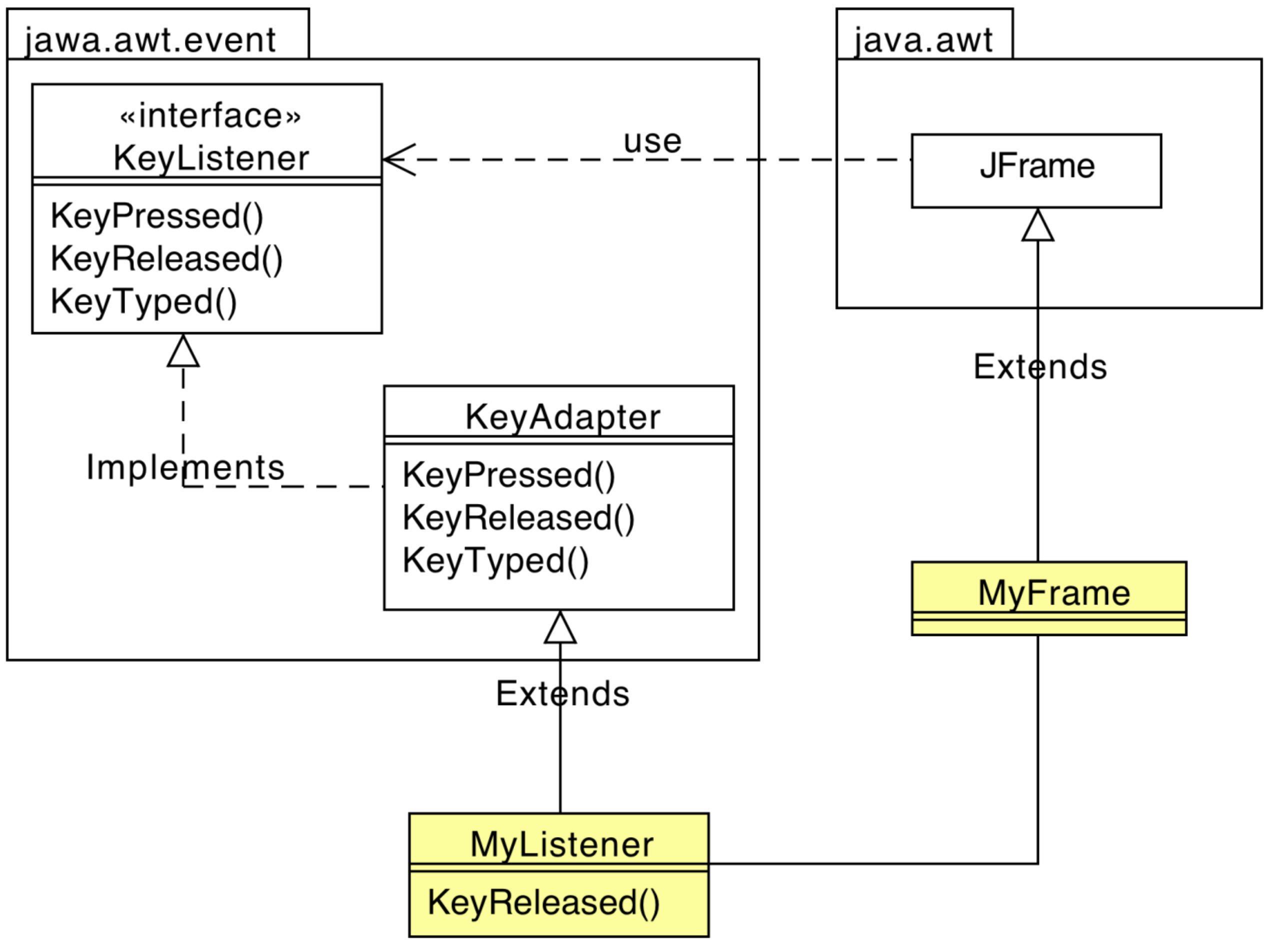
- In Java GUI events are handled by Listeners
- Listener classes need to implement Listener interfaces
  - Include several methods
  - They all should be implemented

Java Listener Adapter

```
class MyListener{
public void KeyPressed(..){}
public void KeyReleased(..){
// … handle event
}
public void KeyTyped(..){} }
```

```
class MyListener{
public void KeyReleased(..){
// … handle event
}
}
```

Java Listener Adapter



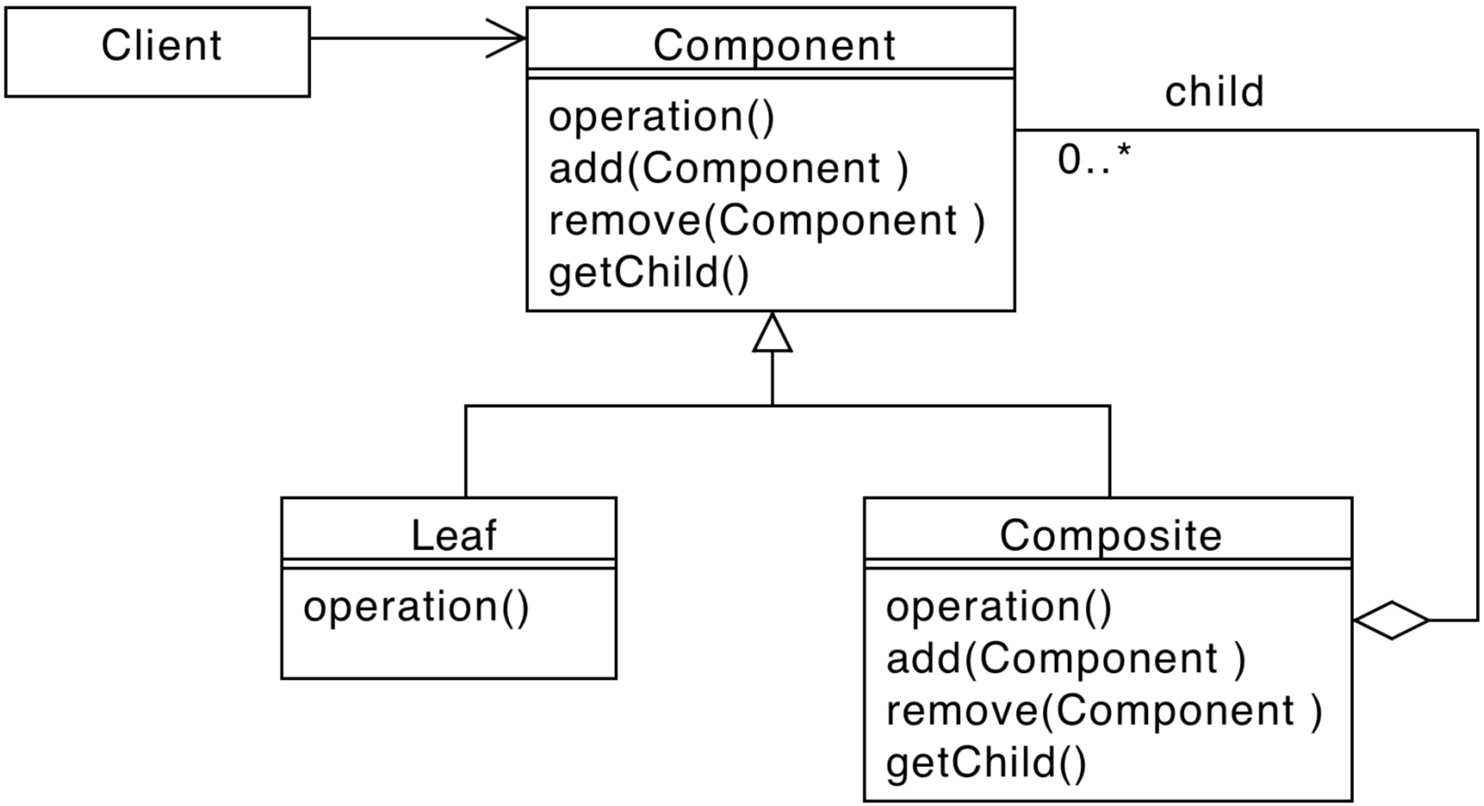
Structural Class Patterns

- Adapter pattern
  - Inheritance plays a fundamental role
  - Only example of structural class pattern

Composite

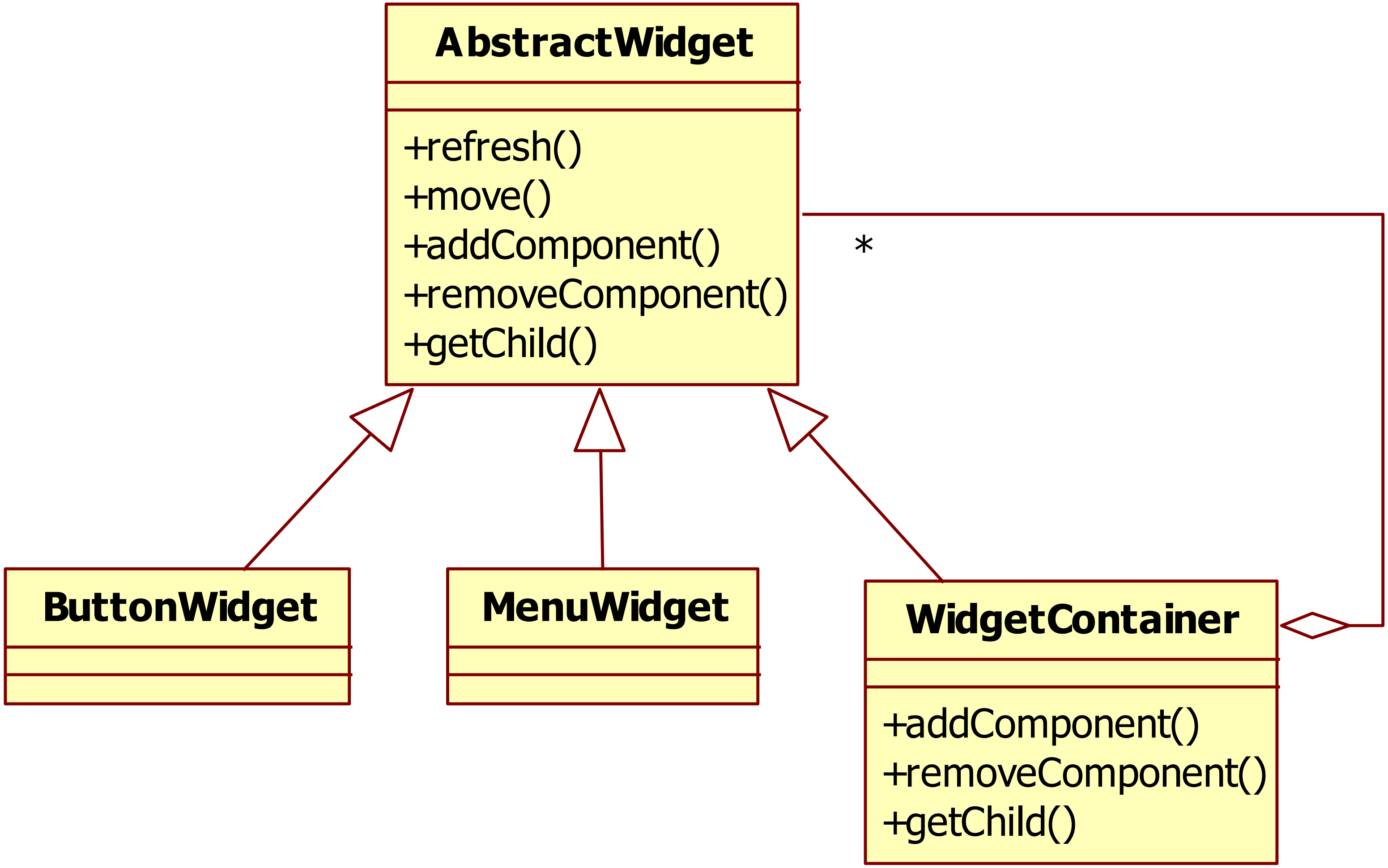
- Context:
  - You need to represent part-whole hierarchies of objects
- Problem
  - Clients are complex
  - Difference between composition objects and individual objects.

Composite



Client
Component
operation()
add(Component)
remove(Component)
getChild()
child
0..*
Leaf
operation()
Composite
operation()
add(Component)
remove(Component)
getChild()

Example: widgets in GUI

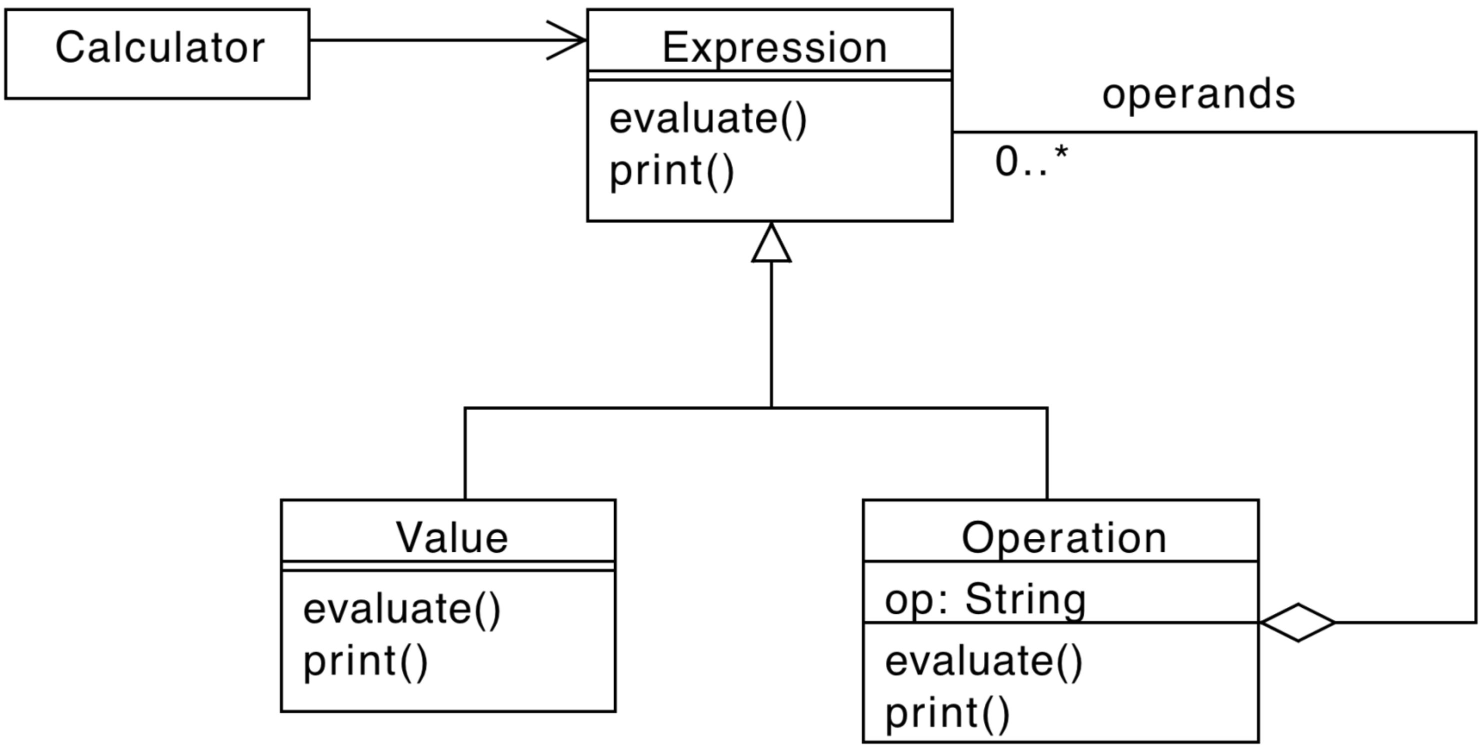


Composite Example

- Arithmetic expressions representation
  - Operators
  - Operands

  - A+ B * (A + B)
- Evaluation of expressions

Composite Example



Composite Example

```
abstract class Expression {
	public abstract int evaluate();
	 public abstract String print();
}
```

Composite Example

```
class Value {
        private int value;

  public Value(int v){
        value = v;
  }
  public int evaluate(){
        return value;
  }
  public String print(){
        return new String(value);
  }
}
```

Composite Example

```
Class Operation {
        private char op; // +, -, *, /
        private Expression left, right

  public Operation(char op,
        Expression l, Expression r){
        this.op = op;
         left = l;
         right= r;
  }
...
```

Composite Example

```
class Operation {
        …
  public evaluate(){
    switch(op){
      case '+': return
                      left.evaluate() +
                              right.evaluate();
                 break;
        …
    }
  }
        …
```

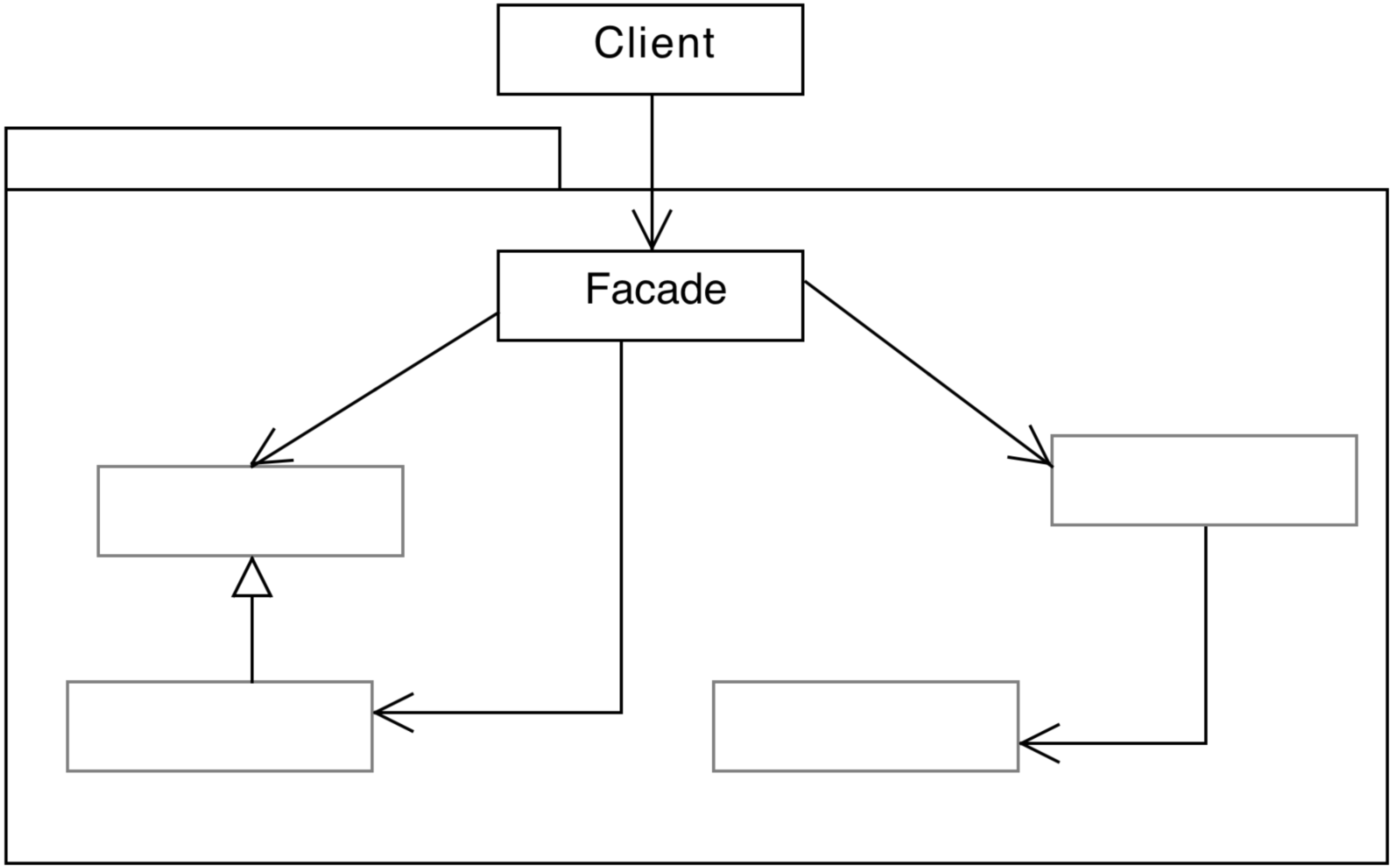
Composite Example

```
class Operation {
        …
  public print(){
    return left.print() + op +  right.print();
  }
}
```

Facade

- Context
  - A functionality is provided by a complex group of classes (interfaces, associations, etc.)
- Problem
  - How is it possible to use the classes without being exposed to the details

Façade



- Package

```
public Class A { public  void method1();}
public Class B {  public void method2();}
public Class C {  void method3();}
```

Without acade

- Client

```
A a; B b; C c;
 a.method1();
 b.method2();
 c.method3();
```

With facade

- Package

```
public class Facade {
        void method1( A.method1)}
        void method2( B.method2)}
        void method3( C.method3)}
}
```

- Client

```
 Facade.method1();
Facade.method2();
 Facade.method3();
```

Behavioral patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Not just patterns of objects or classes but also the patterns of communication.
  - Complex control flow that's difficult to follow at run-time.
  - Shift focus away from flow of control to let concentrate just on the way objects are interconnected.

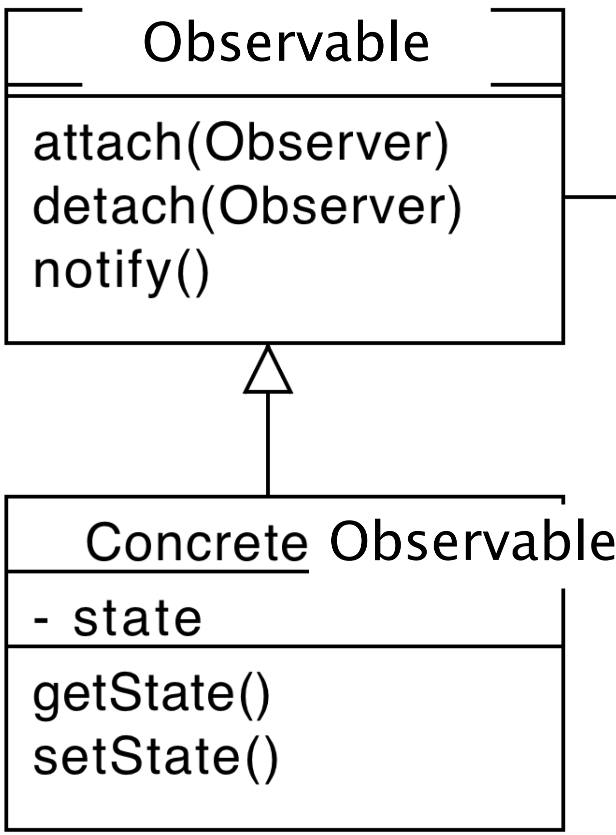
GoF behavioral patterns

- Object-level
  - Chain of Responsibility
  - Command
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Visitor
- Class-level
  - Template Method
  - Interpreter

Observer

- Context:
  - The change in one object may influence one or more other objects
- Problem
  - High coupling
  - Number and type of objects to be notified may not be known in advance

Observer


Observable
attach(Observer)
detach(Observer)
notify()
observers
0..*
Observer
update()
Concrete Observable
- state
getState()
setState()
ConcreteObserver
update()

Observer – Consequences

+Abstract coupling between Observable and Observer

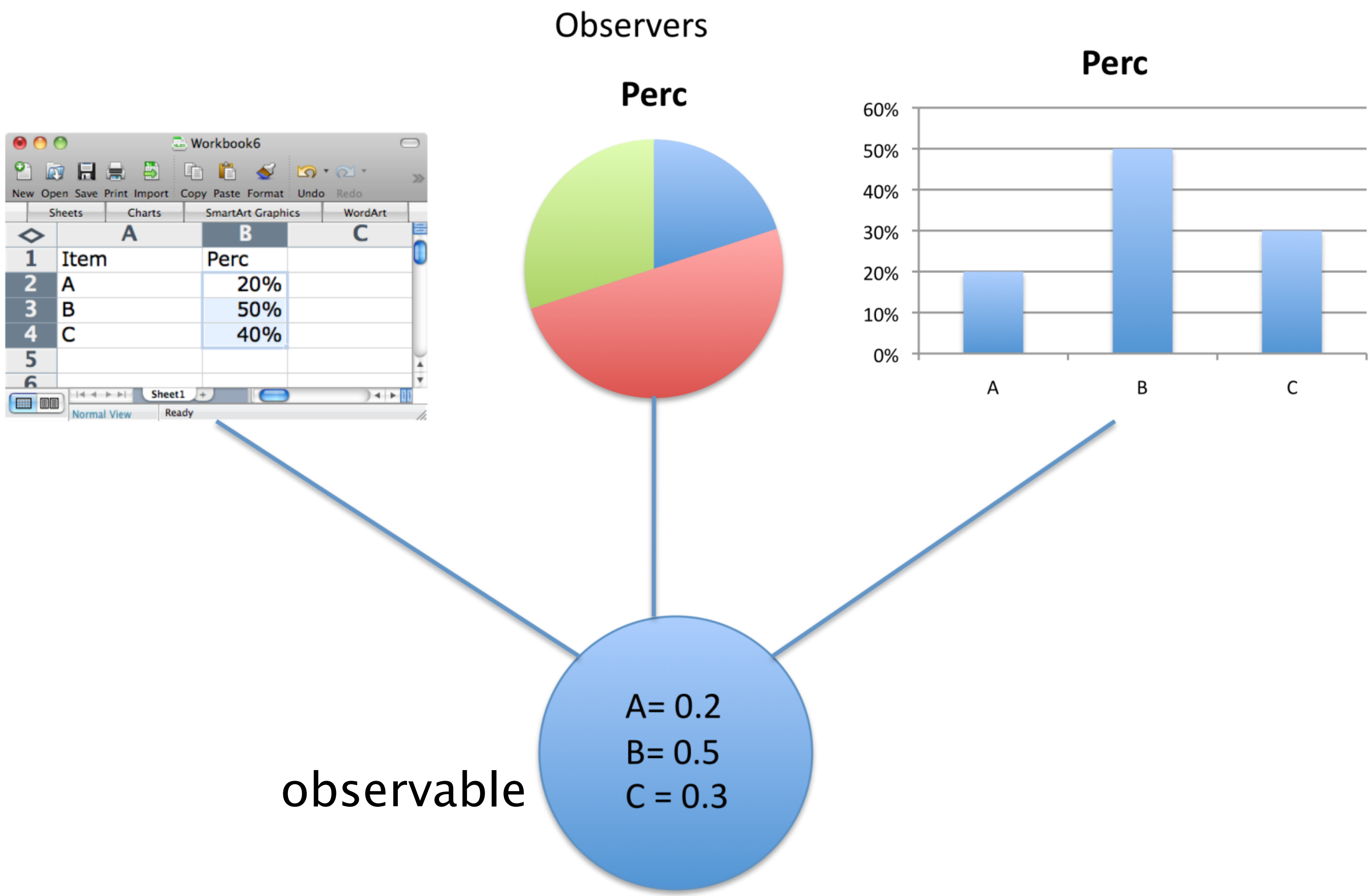
+Support for broadcast communication

-Unanticipated updates

Java Observer-Observable

```
class Observable{
        void addObserver(..){}
        void deleteObserver(..){}
        void deleteObservers(){}
        int countObservers() {}
        void setChanged() {}
        void clearChanged() {}
        boolean hasChanged() {}
        void notifyObservers() {}
        void notifyObservers(..) {}
}
```

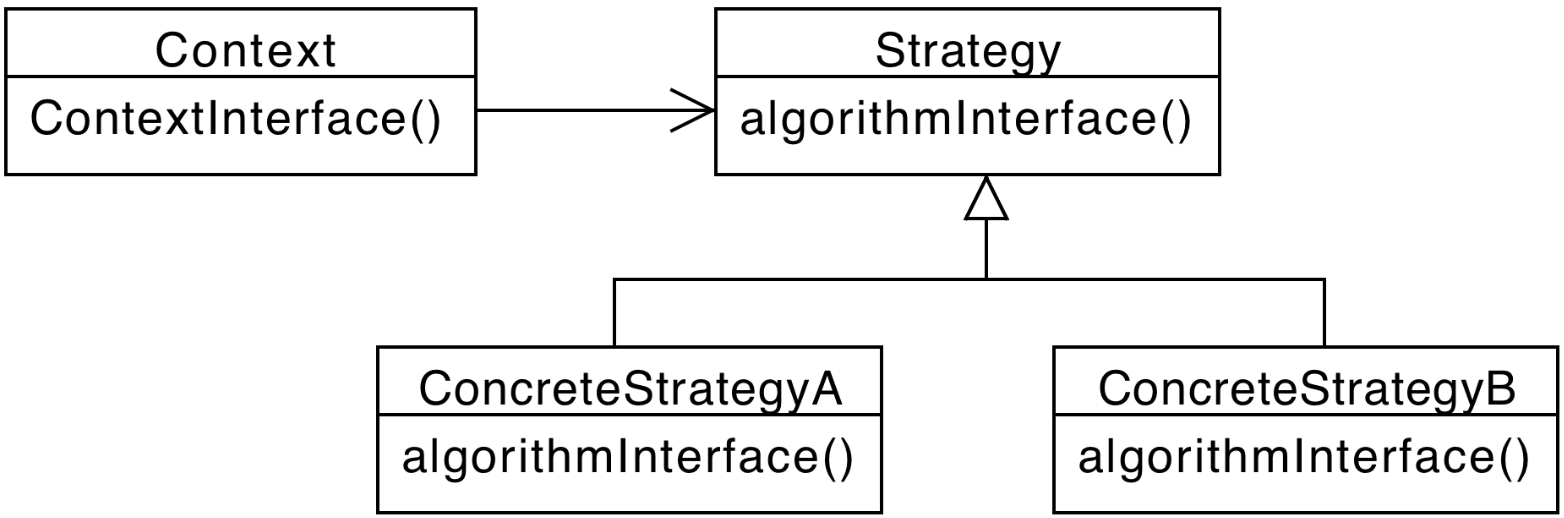
Observer Example



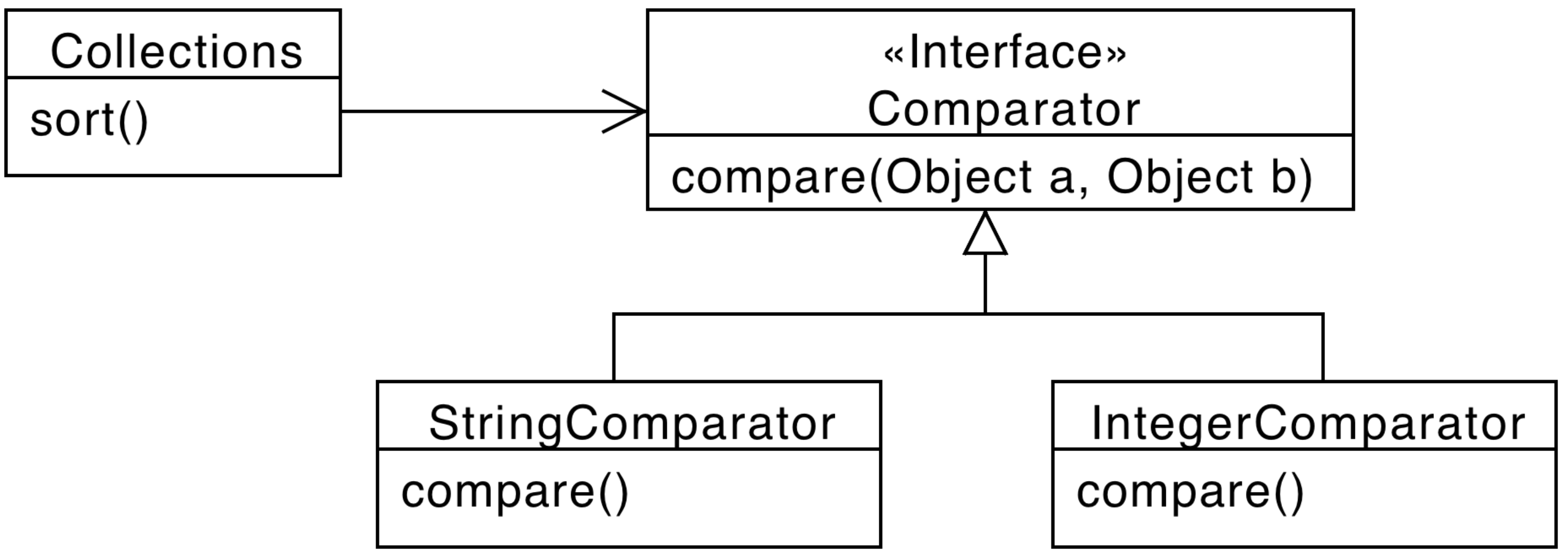
Strategy

- Context
  - Many classes or algorithms have a stable core and several behavioral variations
- Problem
  - Several different implementations are needed.
  - Multiple conditional constructs tangle the code.

Strategy


Context
ContextInterface()
Strategy
algorithmInterface()
ConcreteStrategyA
algorithmInterface()
ConcreteStrategyB
algorithmInterface()

Strategy Example



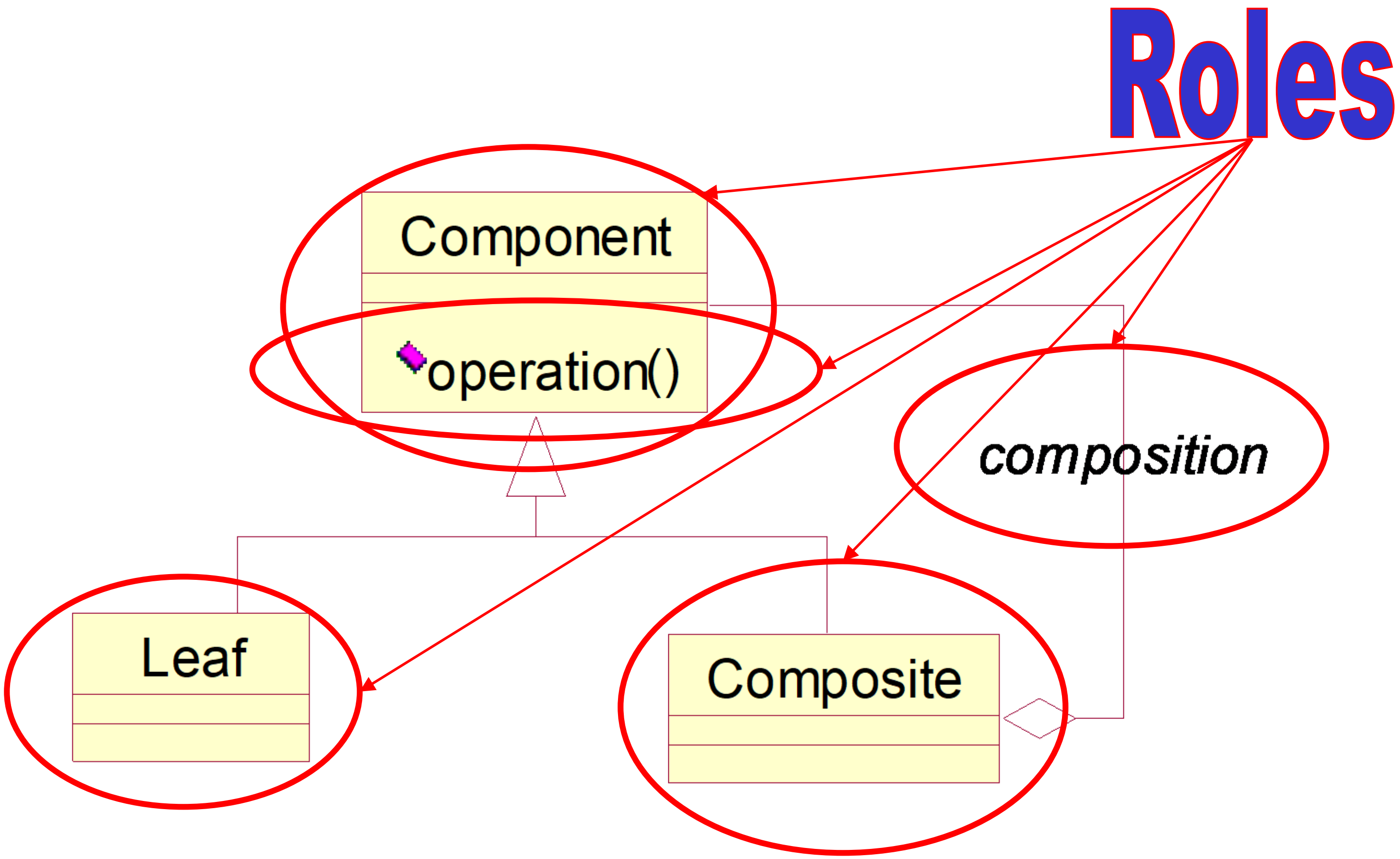
Consequences

+ Avoid conditional statements
+ Algorithms may be organized in families
+ Choice of implementations
+ Run-time binding
- Clients must be aware of different strategies
- Communication overhead
- Increased number of objects

Analysis with Patterns

- Process:
  - Find out what patterns are used
  - Find out what the role assignments are
  - Find out how functionalities are implemented by means of patterns
  - …use this knowledge

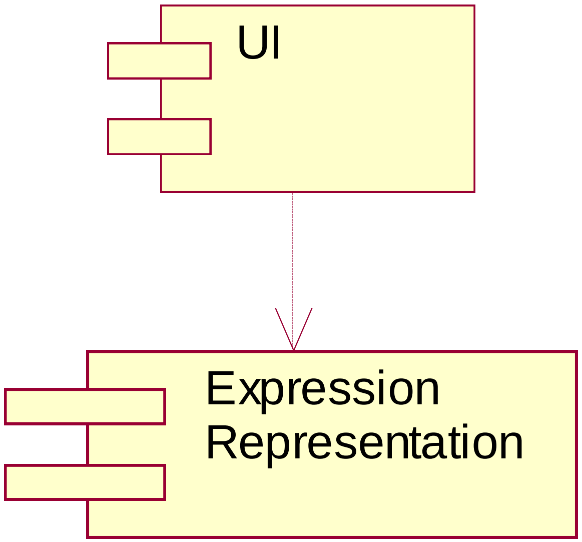
Example



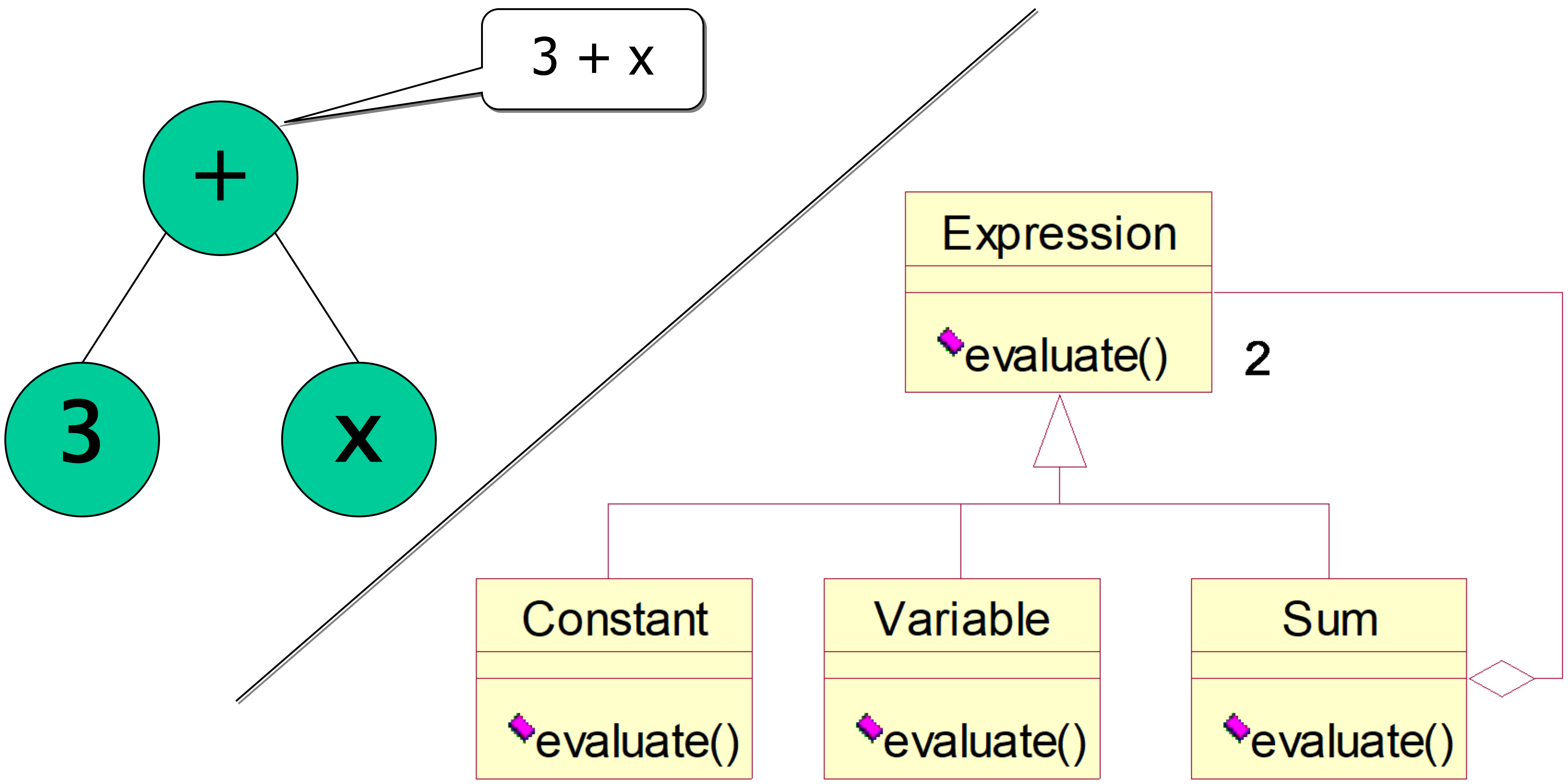
Example

- A program that handles symbolic algebraic expression manipulation
- Functionality:
  - Definition of expressions
  - Evaluation of expressions

Example – Architecture



Expression Representation

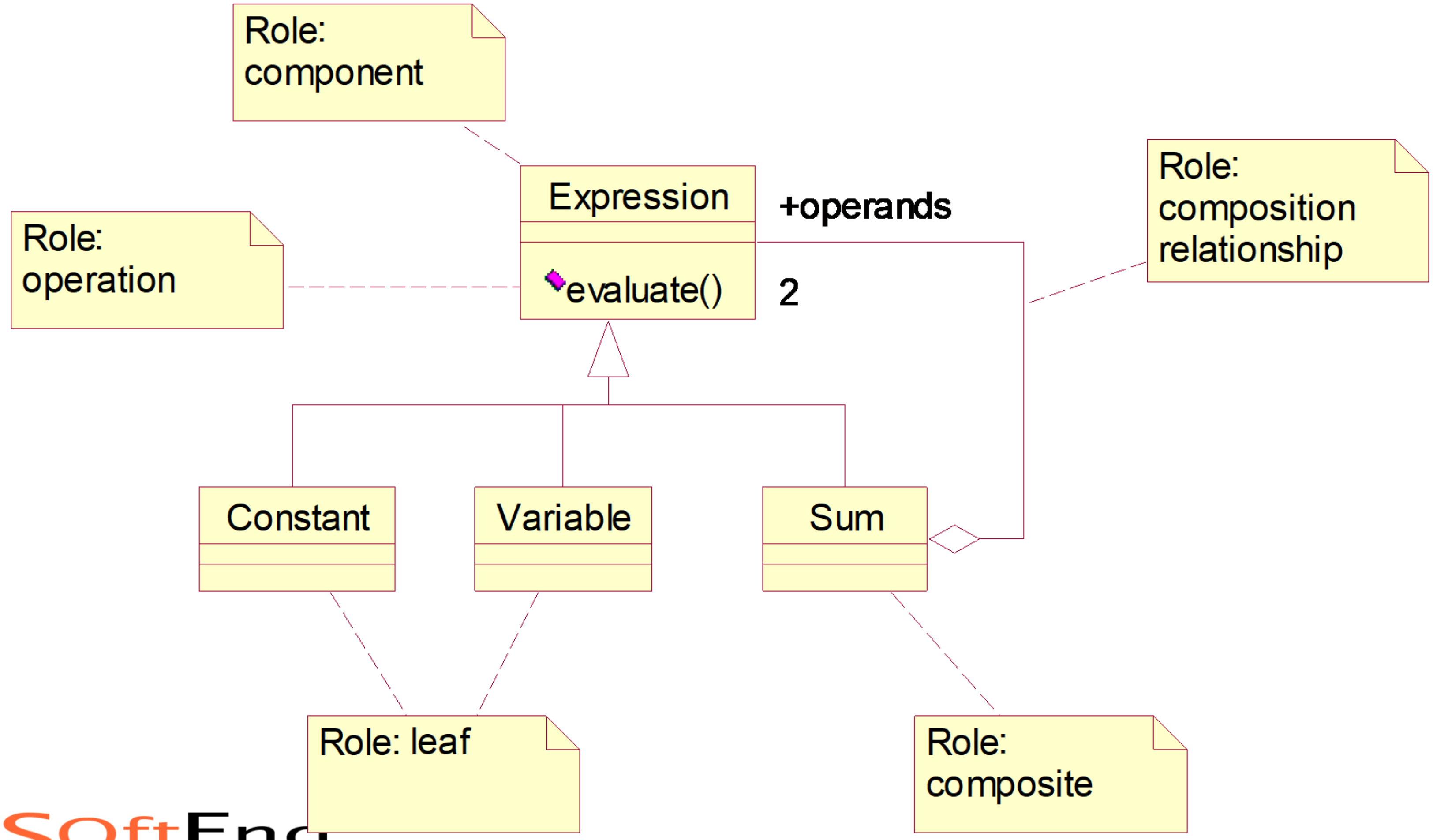


Expression Definition

```
Constant three=new Constant(3);
Variable x = new Variable(“x”);
Expression e = new Sum(three,x);
//…
float result = e.evaluate();
//…
```

SoftEng http://softeng.polito.it

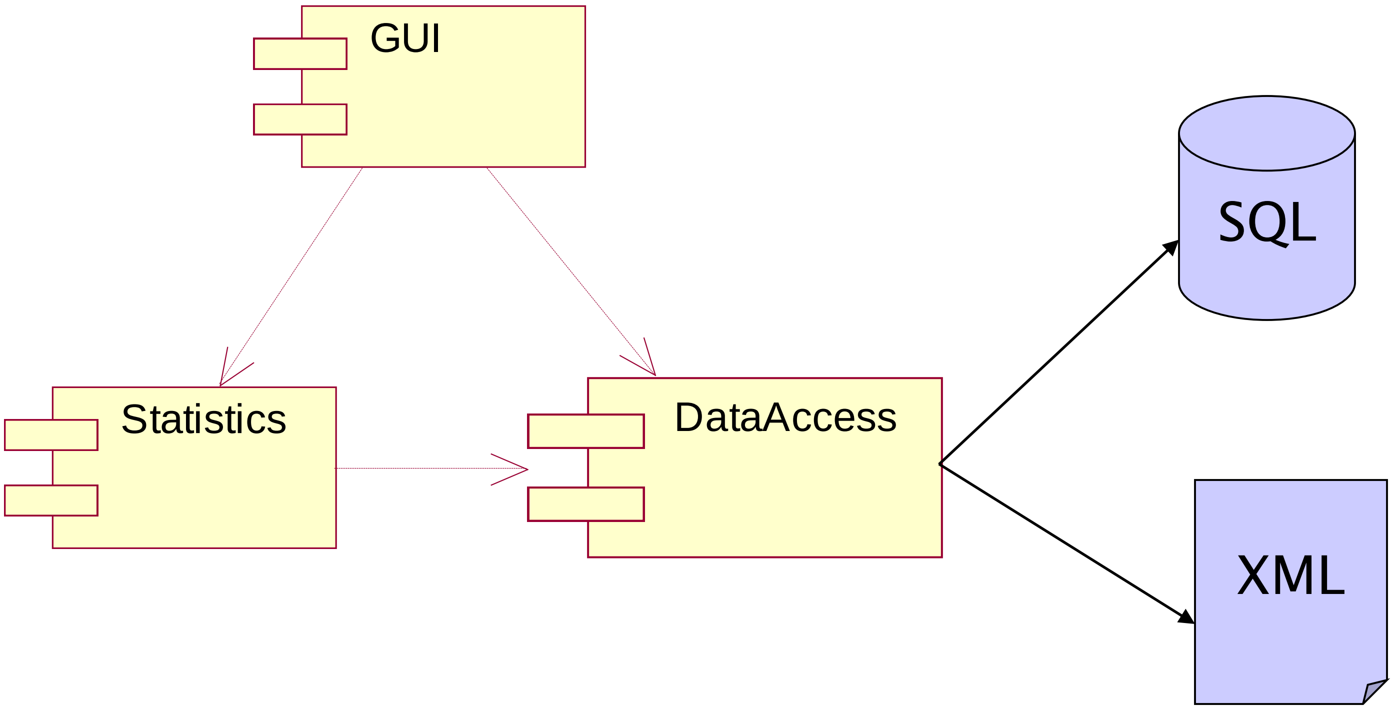
Roles Assignments



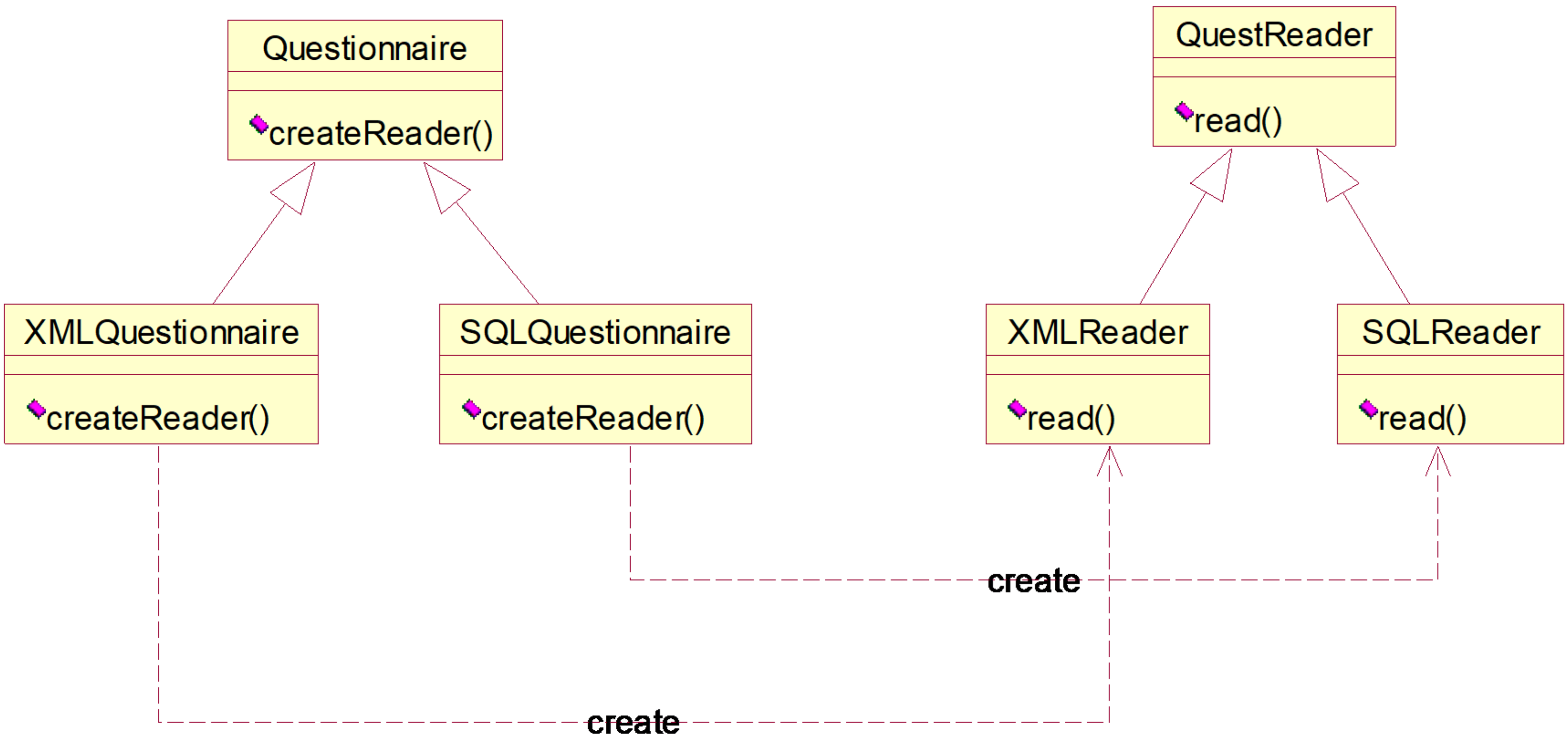
Exercise

- A program that calculates statistics for questionnaire replies.
- Data can be either in:
  - An XML file
  - A relational database
- All the statistics manipulations are independent from the medium

Exercise – Architecture



Exercise – Data Access



Exercise – Questionnaire

```
public abstract class Questionnaire{
  private static Questionnaire single;
  public static Questionnaire getQuestionnaire(){
    if(single!=null) return single;
    single = new something();
    return single;
  }
  public QuestReader createReader();
}
```

```
Questionnaire q =
  Questionnaire.getQuestionnaire();
QuestReader qread = q.createReader();
//...
q.read();
```

Exercise

- What patterns are used in this example?
- What are the role assignments?
- What purpose do(es) the pattern(s) serve?

Verification

Verification

- Functional requirements
  - Traceability matrix
  - Scenarios executed on architecture
  - Inspection
- Non functional requirements
  - Performance
    - Scenarios enriched with time model
  - (Inspection)

Traceability matrix

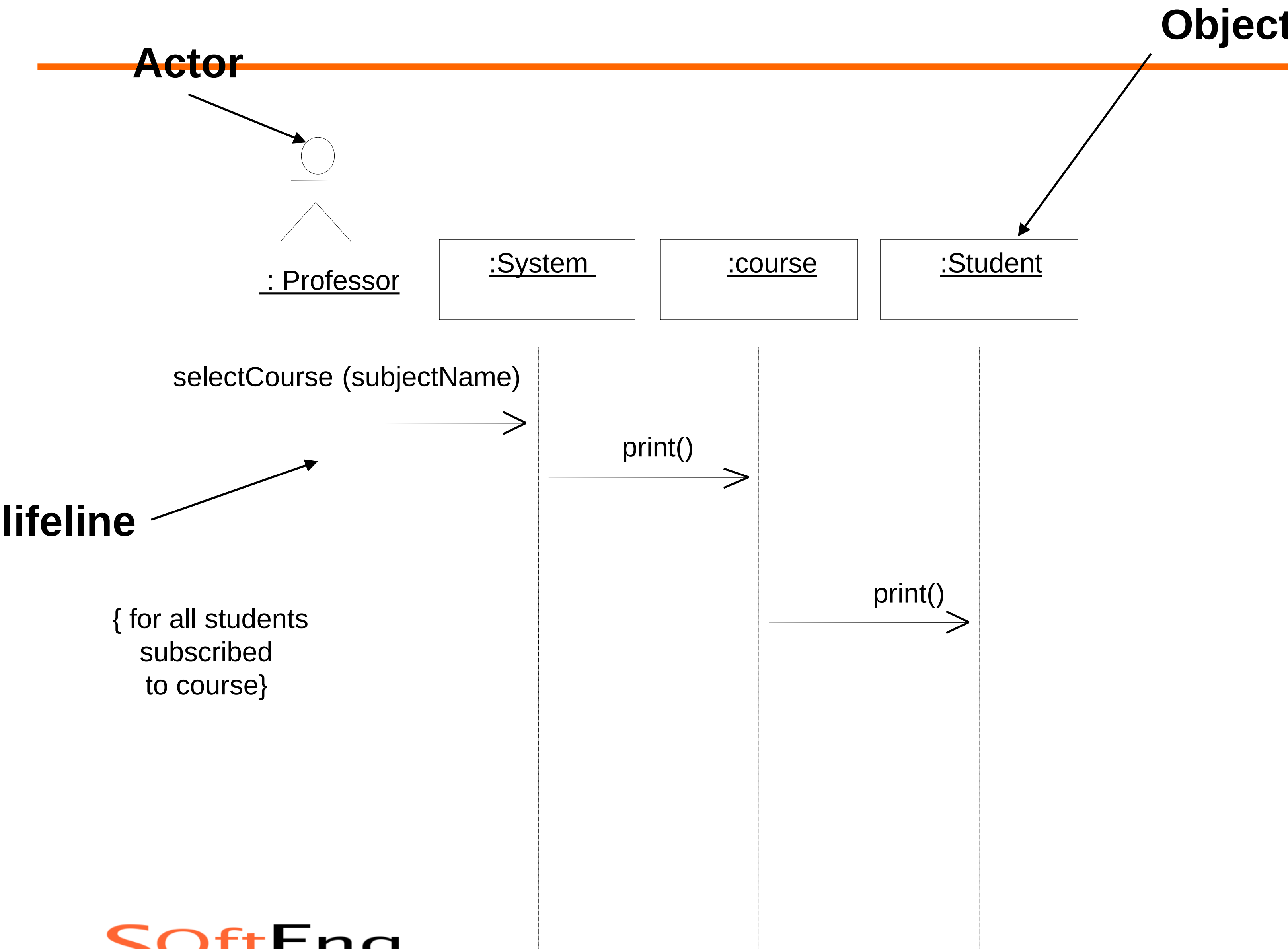
	AwayManagementStrategy	Boiler	CRoom	DefaultHouseSettings	Env	Environment	HouseController	InvalidTimeException	PhysBoiler	PresenceManagementStrategy	Room	RoomManagementStrategy	RoomSettings	SetRoomParametersActivity	SetRoomParametersDialog	XMLSettings
Temp-UR-F1											X		X	X	X	X
Temp-UR-F2											X		X	X	X	X
Temp-UR-F3											X		X	X	X	X
Temp-UR-F4											X		X	X	X	X
Temp-UR-F5											X		X	X	X	X
Temp-UR-F6		X	X		X	X	X		X	X	X	X				
Temp-UR-F7	X	X	X		X	X	X		X		X	X				
Temp-UR-F8		X	X		X	X	X		X	X	X	X				
Temp-UR-F9		X	X		X	X	X		X	X	X	X				
Temp-UR-F10	X	X	X		X	X	X		X		X	X				
Temp-UR-F11								X							X	
Temp-UR-F12				X									X	X		
Temp-UR-F13	X	X	X		X	X	X		X		X	X				
Temp-UR-F14	X		X		X	X	X			X	X	X				
Temp-UR-F15			X		X	X	X			X	X	X				
Temp-UR-F16	X									X						
Temp-UR-F17			X	X			X				X					X
Temp-UR-F18		X					X		X							
UR-Inv 1	X	X	X		X	X	X		X		X	X				
UR-Inv 2		X	X		X	X	X		X	X	X	X				

Traceability matrix

- Each functional requirement (from requirements document) must be supported by at least one function in one class in the software design
  - The more complex the requirement, the more member functions needed

Scenarios

- Each scenario (from requirements document) must be feasible
  - It is possible to define a sequence of calls to member functions of classes in the software design that matches the scenario



SoftEng
http://softeng.polito.it

Key points

- Architecture
  - defining high level components and their control, communication model
  - Tools: UML or ADL models, structural and dynamic
  - Styles: Layered, client server (2 tier, 3 tier), peer to peer, shared repository
- Design
  - Define internals of components
  - Tools: UML models
  - Design patterns

Key points

- Verification
  - inspections
    - Architecture can satisfy functional properties (as defined in requirements doc)?
      - Traceability matrixes
      - Scenario execution
    - Architecture can satisfy non functional requirements?
      - Enriched scenarios
  - build prototype

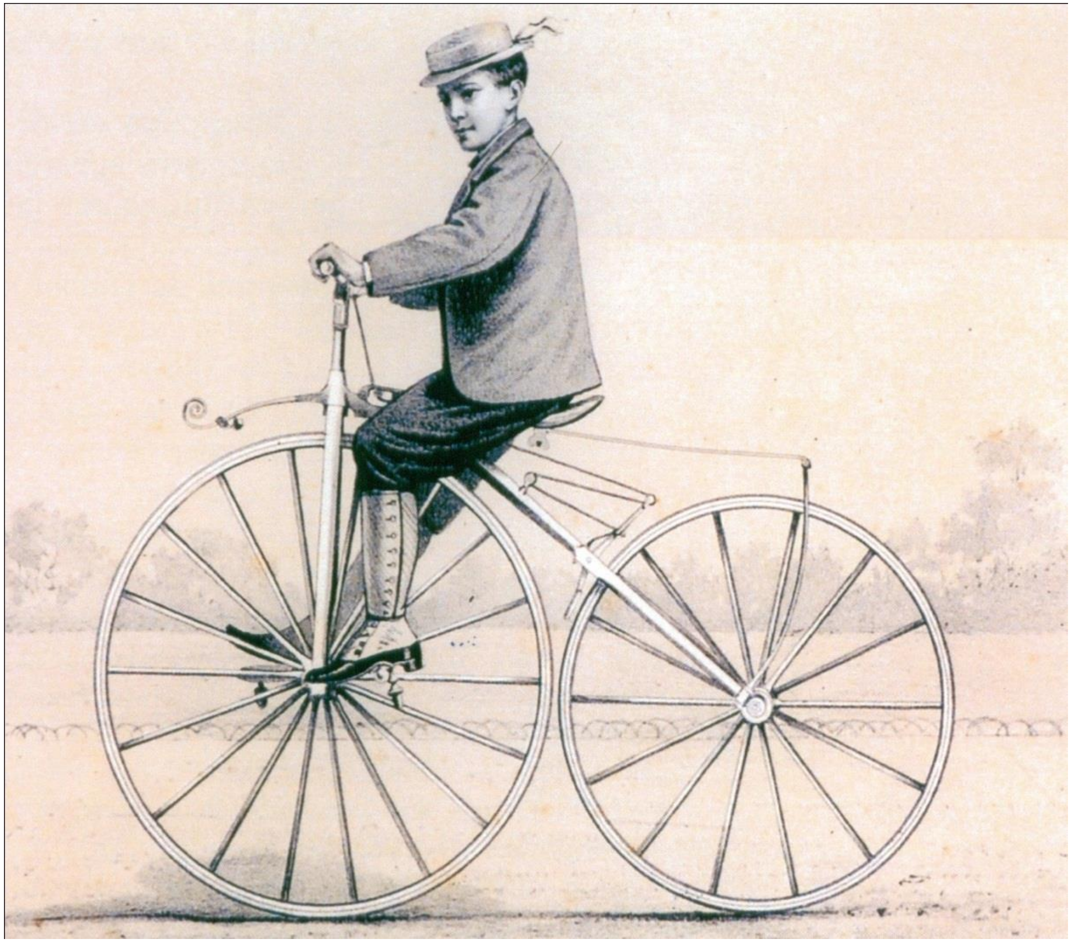
Bicycles ..

Draisine

- 1820
- Front wheel steering
- Foot powered

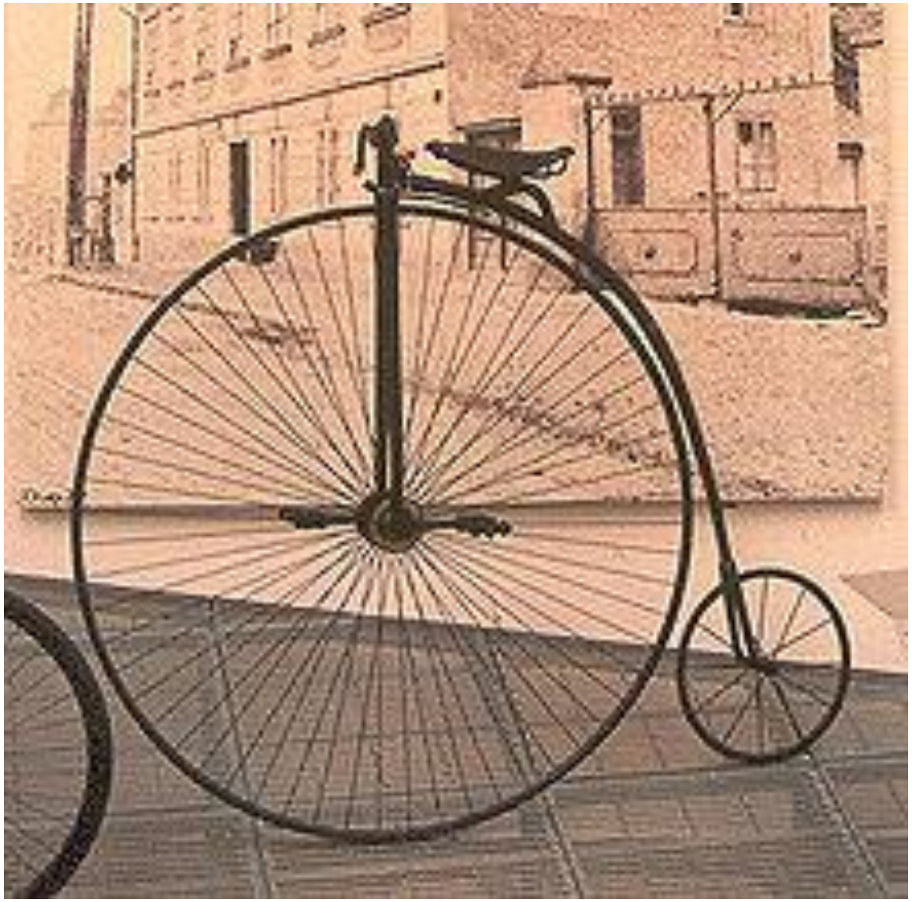
Velocipede

- 1860
- Front wheel steering
- Crank pedal on front wheel



Penny farthing

- 1870
- Larger front wheel
  - More speed
  - More comfort
  - unstable



Dwarf ordinary

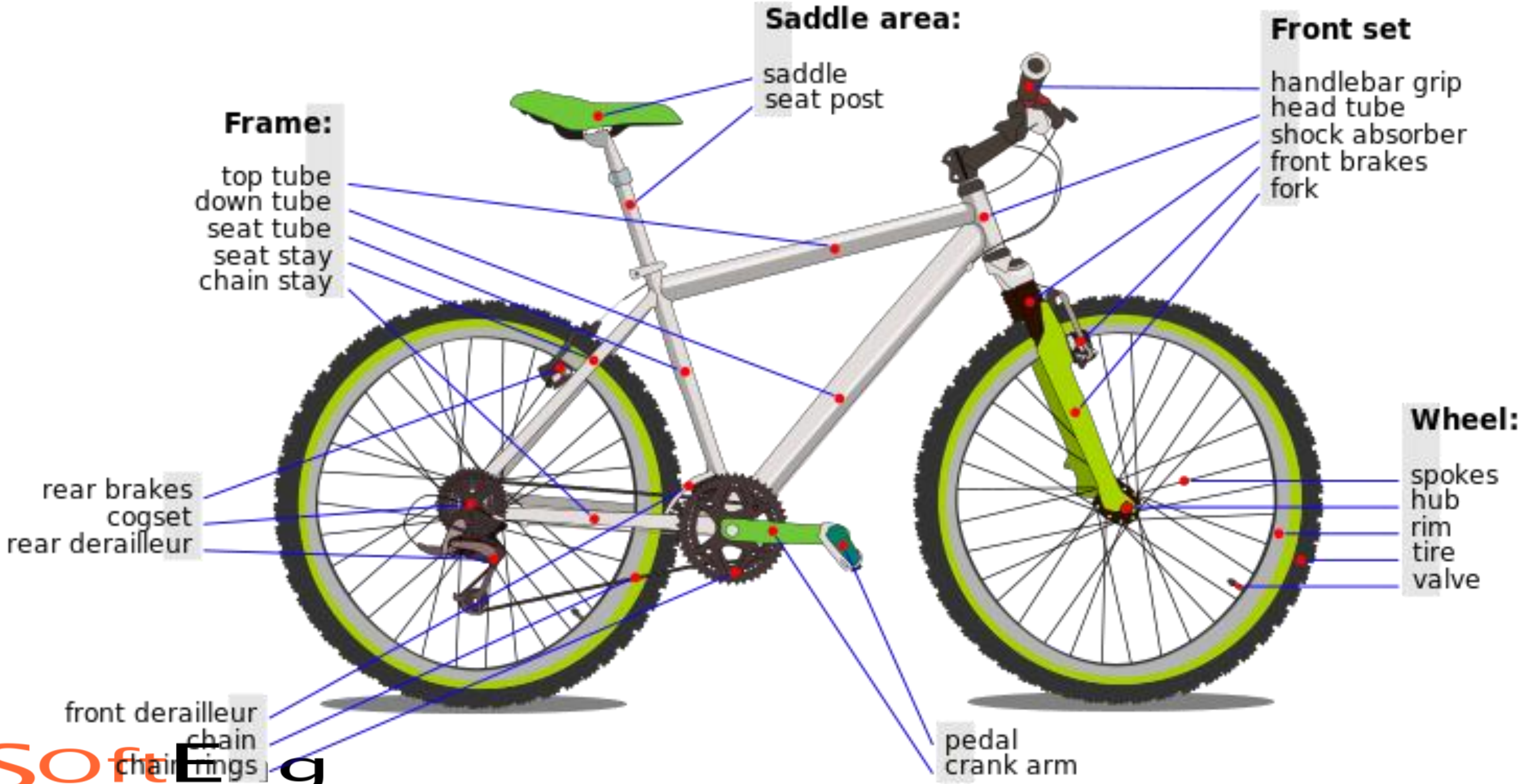
- Smaller front wheel, seat backwards
- More stable, less speed, less confort

THOMAS McCALL AND HIS BICYCLE.
(From a Photograph by Bruce and Howie, of Kilmarnock.)

And ..

- 1870, chain drive
  - Solves problem of steering and pedaling on front wheel
  - Pedals in middle, power to rear wheel
- 1885, seat tube (diamond frame)
- 1888, pneumatic tire (Dunlop)
  - Comfort
- 1890
  - Rear freewheel (coasting)
- 1905
  - Derailleur gears

Dominant design



Dominant design

- Requires time to develop and be commonly shared in the domain
- Requires specific components
- Leads to specialized companies / roles
  - Company to design/develop tyres
  - Company to design/develop chains
  - Etc..

Other designs

Requirements – bike

- Functional requirements
  - transport one person from place to place
    - Steer
    - accelerate
    - brake
- Non functional requirements
  - Efficiency : speed from 10 km/h to 50 km/h
    - (Speed from 10 km/h to 150 km/h)
  - Efficiency : weight between 10 and 15kg
  - Efficiency: reasonable torque to start: < 40Nmeters
  - Usability: out of 50 average users, at least 60% of them find the bicycle easy to use
  - Only human power (no engines)
  - Safety (no harm to driver)
  - Security (difficult to steal)
  - Cost (between 100 and 200 euro)

Design vs requirements

	Draisine	Velocipede	Penny farthing	Another design	Dominant design
Transport one person	y	Y	Y	Y	Y
Eff – speed	< 10kmh	Y	Y	Y	Y
Eff – torque at start	Y	N	N	Y	Y
Eff – weight	y	Y	Y	Y	y
Human power	y	Y	Y	Y	Y
safety	Driver less high	Driver vey high	Driver even higher	Y	Y
Reduce speed	With feet on road	Applying negative force to pedal	Applying negative force to pedal	Y	y brakes