

Software Engineering - A.A. 22/23

**Java Basics: a primer on the Java language and idioms
with restrained intention on the method**



Enrico Vicario

Dipartimento di Ingegneria dell'Informazione
Laboratorio Tecnologie del Software – stlab.dinfo.unifi.it
Università di Firenze

enrico.vicario@unifi.it, stlab.dinfo.unifi.it/vicario

1/116

Aims and contents

- This is a java primer
 - It is intended to introduce the basics of the language and the idioms for its effective usage in 6-8 lessons, as a part of a class where we will later come on the method
 - It aims at providing a basis that makes accessible higher level concepts related to design patterns and development of a domain logic
 - It is intended for people who are familiar with the c language
- It is obtained through
 - an initial excerpt from David Flanagan, "Java in a nutshell", Chp.2 "How Java differs from c" and Chp.3 "Classes and objects", reworked to make more evident the relation with the c language
 - injection of various selected items of effective Java by Joshua Bloch
 - various stuff from Dzone or StackOverflow
 - an excerpt on Generics from Java Tutorials (not delivered) (<http://docs.oracle.com/javase/tutorial/java/generics/index.html>)
- All this repeatedly rethought year by year (starting with 2013/14)

2/116

- Java is an object-oriented language.
 - "Object-oriented" is a term that has become so commonly used as to have practically no concrete meaning.
- Some terminology
 - Object based: encapsulate together behavior and state
 - Object oriented: add inheritance (?!)
 - Much about discussions of the type:
"ada is object based, but not object oriented!"
"are c++ or Eiffel Object Oriented?"
...
- a pragmatic approach
 - we are not here to define a new language
 - we are interested in understanding the structure of Java
 - we will later extend this so as to use classes and objects also in conceptual models, not necessarily cast into code
 - we assume a good understanding of syntax and semantics of the c language, and try to follow the roadmap of that language

- Primitive and reference types, and variables
- Reference type definition, declaration, construction, and reference
- Arrays, Strings
- Garbage collection
- Class variables and methods

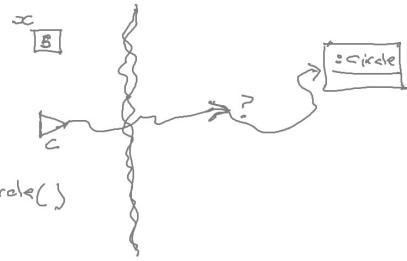
- A broad conceptualization
 - an object
 - has a state (data members)
 - has a behavior that uses and modifies the state (methods)
 - has identity (two objects with the same state are not the same object)
 - a class
 - is a type of objects
 - all this not only for OO programming, also for conceptual models
- intuitively enough
 - Object is related to the concept of variable,
much like a struct variable in c
 - ... and Class is related to the concept of type,
much like a user defined struct type in c,
somehow extended with functions accessing struct members
- Should we first define what is an Object or what is a Class?

- About types in c
 - a type is a set of values equipped with operations (e.g. float and +)
 - types are primitive (e.g. float) or user defined aggregations (struct)
- Java emphasizes more than c
the difference between 2 kinds of variables
 - types are either primitive (e.g. float, boolean, ...)
 - or user defined aggregations of data members and methods (class),
called reference data types (i.e. references to objects)
- Variables in the two kinds are managed differently
 - variables of primitive types (int, float, ...) are “immediate”
 - objects are managed only through their reference

sketch on the different management of primitive and reference types

Reference vars \rightsquigarrow point to primitive types
 \rightsquigarrow op. ↓

```
int x;  
x = 5  
Circle c;  
// C.radius =  
c = new Circle();
```



7/203

objects ... as intended in Java, wrt c

- About variables in c
 - a variable is a location of memory, carrying a value of some type
 - the value can be varied along the computation, the type is invariant
 - a declaration associates a name with the variable
 - but, the identity of a variable is only in its address
 - the program has direct access to all the memory, and the type can be re-cast, so, any location is potentially a variable of any type (:)
- About objects in Java
 - memory management is separated from the program
 - a declaration creates a location of memory of a specified type, and returns a reference to the location
 - the location carries a value of the specified type
 - the value can be varied along the computation, the type is invariant.

8/116

■ Java Primitive Data Types

Type	Contains	Default	Size
boolean	true or false	false	1 bit
char	Unicode character	\u0000	16 bits
byte	signed integer	0	8 bits
short	signed integer	0	16 bits
int	signed integer	0	32 bits
long	signed integer	0	64 bits
float	IEEE 754 floating	0.0	32 bits
double	IEEE 754 floating	0.0	64 bits

■ Differences wrt c

- adds byte and boolean
- strictly defines size and signedness of types (notably of int)

9/116

■ the boolean primitive type

- boolean values are not int, may not be treated as int, and may never be cast to or from any other type.
- conversions between boolean b and int i are still possible
 - b = (i != 0); // integer-to-boolean
 - i = ((b) ?1:0); // boolean-to-integer

■ guards in statements (if/else, while, for, do/while)
must return a boolean

- the values 0 and null are not the same as false, and non-0 and non-null values are not the same as true.
- the following c code is not legal in Java

```
int i = 10;
while(i--){ Object o = get_object();
            if (o) { do { ... } while(j); } }
```
- ... to make it legal, each guard must become a boolean returning expression

```
int i = 10;
while(i-- > 0){ Object o = get_object();
                if (o != null) { do { ... } while(j != 0); }}
```

10/116

- the `char` primitive type
 - the `char` type holds a two-byte Unicode character
 - the Unicode character set is compatible with ASCII
 - the first 256 characters (0x0000 to 0x00FF) are identical to the ISO8859-1 (Latin-1) characters 0x00 to 0xFF.
 - If you are only using ASCII or Latin-1 characters, there is no way to distinguish a Java `char` from a C `char`.
 - values of type `char` do not have a sign
 - if a `char` is cast to a `byte` or a `short`, a negative value may result
 - a character may be represented with the Unicode escape sequence `\uxxxx`
 - Java also supports all of the standard c character escape sequences, such as `\n`, `\t`, and `\xxxx` (where `xxx` is three octal digits)
(why do programmers always mix up Halloween and Christmas?)
- Java does not support line continuation with `\` at the end of a line
 - long strings must either be specified on a single long line, or they must be created from shorter strings using the string concatenation `+`

11/116

- Integral Types
 - there are no modifiers `long`, `short`, `unsigned`
 - overcomes the mess of c integer types
 - makes them platform independent
 - a `long` constant may be distinguished from other integral constants by appending the character `l` or `L` to it.
 - integer division by zero or modulo zero causes an `ArithmaticException` to be thrown

12/116

■ floating-Point Types

- floating-point literals may be specified by appending an `f` or `F` to the value;
- they may be specified to be of type `double` by appending a `d` or `D`.
- special values
 - `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`
 - `NaN` (Not a Number)
- `NaN` is unordered--comparing it to any other number, including itself, yields `false`
- defined in `java.lang.Float` and `java.lang.Double` classes
 - Use `Float.isNaN()` or `Double.isNaN()` to test for `NaN`.
- Negative zero compares equal to regular zero (positive zero),
but the two zeros may be distinguished by division:
division by negative/positive zero yields negative/positive infinity
- Floating-point arithmetic never causes exceptions,
even in the case of division by zero.

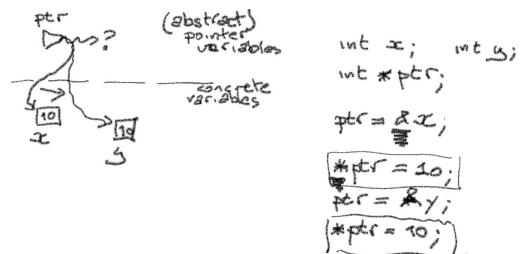
■ Reference data types are variables maintaining references to objects

- ... and to arrays
- since objects have a type, reference types also have one
- somehow similar to pointers in C

■ much of the difficulty in handling reference types
often comes from confusion
between objects and references to objects

about variables and pointers in c ...

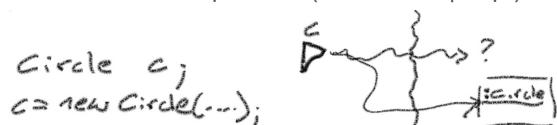
- In c, information is encoded in two layers of variables
 - (I would draw all this in a horizontal picture)
 - a concrete layer of variables that carry values
 - an abstract layer of pointers that identify concrete variables or pointers
 - the two layers comprise a kind of “reflection” architecture:
the abstract level encodes information about the concrete level
(a.k.a. knowledge and operation layers)
- c provides referencing/dereferencing operators & and *
 - intertwine the syntactic classes of <Variables> and <Expressions>
 - changing a value in the abstract layer
results in changing the identified variable in the concrete layer
 - allow the programmer to jump up and down across the two layers



15/116

... and about objects and references in Java – 1/2

- in Java, the picture is somehow vertical, and jumps are not allowed
 - objects are maintained in a separate space,
accessed only by the memory manager
 - at the creation of the object, the program receives a reference value,
which can be stored in a variable of reference type
 - the reference type variables of Java are much like the pointers of c,
but no dereferencing is allowed:
Java does not provide referencing/dereferencing operators &, *
 - objects are managed only through their references,
but references are always managed by value
- Java does not enable manipulation of pointers or memory addresses
 - no cast of object or array references into integers or vice-versa
 - no pointers arithmetic
 - no evaluation of the size in bytes of primitive or reference type
 - no & to get an Expression from a VariableReference (in c: Expr:=...|&Var)
 - no * to get a VariableReference from an Expression (in c: Var:=...|*Expr)



16/116

- this has various consequences, not much surprising when the memory model is understood
 - Java references are variables that may contain the reference to an object
 - objects live somewhere else, they can be created through the "new" keyword (much similar to malloc()), they can be accessed only through the name of a reference, they cannot be passed by value
- positive consequences
 - avoid pointers related faults
 - enable Java's run-time checks and security mechanisms

- a subtle difference wrt to C, major in the concept, minor in the impact
 - Java does not allow referencing/dereferencing through & and *, and thus, a variable can be referenced only through its name
 - this rules out the C idiom:
"pass the value of &x to let a function produce a side effect on x", which is the standard way to side-effect a variable through an expression out of the variable declaration scope
 - (see next slide for illustration)
- corollary:
 - a variable in a primitive type declared within the body of a function f() can be side-effected only in the body of f()
- can be disappointing for a C-programmer, but has not much impact
 - this is about how local variables are meant to be used in Java
 - and, if needed, the limit can be circumvented by "boxing" the primitive value within an object (e.g. Integer, Boolean, ...) whose reference can be passed to a function

- a ground idiom of C

```
thisFunc()  
{ int x;  
...  
f(&x);  
  
§  
f(int *ptr)  
{ ...  
*ptr=3  
§
```

- ... and its surrogate in Java

```
class MyClass  
{  
    int x;  
    setX(...)  
    {  
        delegateXChange(obj o)  
        {  
            o.setX(Integer(x))  
        }  
    }  
}
```

19/203

definition of a class

- A class is a collection of variables and methods that operate on that data
 - state and behavior
 - Introduces a user-defined type
- A class definition collects variable *declarations* and methods *definitions*
 - much like the definition of a struct in C, encapsulating function definitions

```
public class Circle {  
    // variable declarations  
    public double x, y; // The coordinates of the center  
    public double r; // The radius  
    // method definitions  
    public double circumference() { return 2 * 3.14159 * r; }  
    public double area() { return 3.14159 * r*r; }  
}
```

- about quasi equivalent terms

- behavior, operation, method, function, ...
- state, attribute, data member, field, ...
- “definition” is usually replaced by “implementation”

20/116

declaration of an object reference

- A reference type variable is declared the same way as that of a primitive type

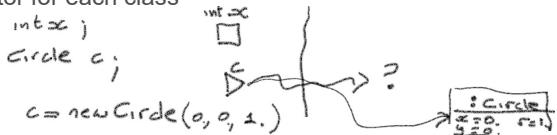
```
int x;      // a variable of primitive type
Circle c; // a variable of reference type
```

... yet,

- while the declaration of the int x reserves the memory space for an int
- the declaration of c does not reserve the space for a Circle, but only the space for a reference to a Circle
- much like the C language when you declare a pointer, which does not imply the existence of any variable at the pointer address

- to have an actual object be pointed by the reference we need to create the object and assign its reference

- much like the C language when you call malloc(), and assign the returned value to a pointer variable
- with the difference that malloc() is specialized by type, with a different constructor for each class



21/116

object construction

- the constructor is a method with the same name of the class (Uppercase)
 - with possible formal parameters
 - returns a reference to an object in the class (omitted in the definition)
 - performs any needed initialization at creation of an object

```
public class Circle {
    public double x, y, r; // center and radius
    // constructor method.
    public Circle(double xCenter, double yCenter,
                 double radius)
    {   x = xCenter; y = yCenter; r = radius; }
    ...
    public double circumference(){return 2 * 3.14159 * r; }
    ...
}
```

22/116

- An object is created through the `new` keyword by invoking a constructor of the `Object` class

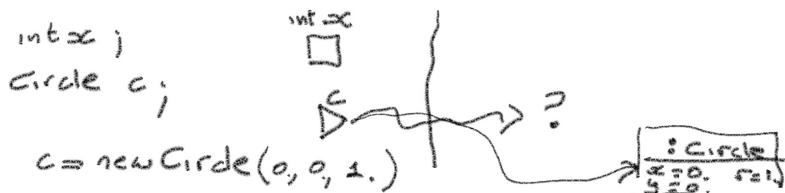
```
public class Circle {  
    public double x, y, r; // The center and the radius  
    public Circle(double x, double y, double r)  
    { this.x = x; this.y = y; this.r = r; }  
    ...  
}  
// in some client object:  
Circle c = new Circle(1.414, -1.0, .25);  
// or equivalently:  
Circle c;  
c = new Circle(1.414, -1.0, .25);
```

- for Strings a contracted form is allowed

```
String s = "This is a test";
```

- The memory for newly created objects is dynamically allocated.

- like calling `malloc()` in C to allocate memory for an instance of a struct



■ accessing object data

- much like using . on dereferenced pointers to structures in c
- ```
Circle c = new Circle();
c.x = 2.0; // Initialize our circle to have center (2, 2)
c.y = 2.0;
c.r = 1.0; // ... and radius 1.0.
```

■ invoking object methods

- refer to methods as if they were fields in the object itself:
- ```
Circle c = new Circle();  
double a;  
c.r = 2.5;  
a = c.area(); // cmp this to a = area(c);
```

■ methods operate in the context of the object

- r can be used by c.area() even if it is not passed as an actual parameter
- somehow like if you would write area(c)

TBD: add more comments on the concept of (single) dispatch
TBD: add a remark on the disappearance of ->

25/116

■ An object can be the target of multiple references
(this is called shared reference, or concurrency,
and becomes relevant in database mapping)

```
Button p, q;  
p = new Button(); // p refers to a Button object  
q = p; // q refers to the same Button.  
p.setLabel("Ok"); // A change to the object through p...  
String s = q.getLabel(); // ...is also visible through q.  
// s now contains "Ok".
```

■ conversely, a variable of primitive type can be referred only by its name
and within its declaration context (scope and lifetime)

- int i = 3; // i contains the value 3.
- int j = i; // j contains a copy of the value in i.
- i = 2; // Changing i doesn't change j.
- // Now, i == 2 and j == 3.

26/116

remark: some obvious effects on reference data types – 2/2

- assigning a reference variable to another does not copy the object

```
Button a = new Button("Okay");
Button b = new Button("Cancel");
a = b;
// a refers to the same object as b.
// The object initially referred by a is lost.
```

TBD: should something about defensive copies be anticipated here?

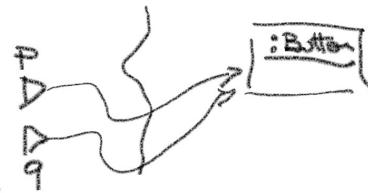
Button p, q;

p = new Button(...)

q = p;

...

f(q == p) // q == p button
is still the same object



27/116

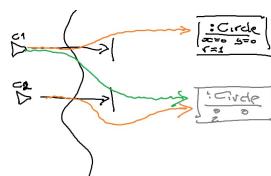
sketches on objects vs object references

Circle c1, c2;

c1 = new Circle(0, 0, 1);

c2 = new Circle(0, 0, 2)

c1 == c2

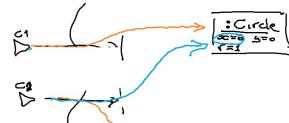


Circle c1, c2;

c1 = new Circle(0, 0, 1);

c2 = c1

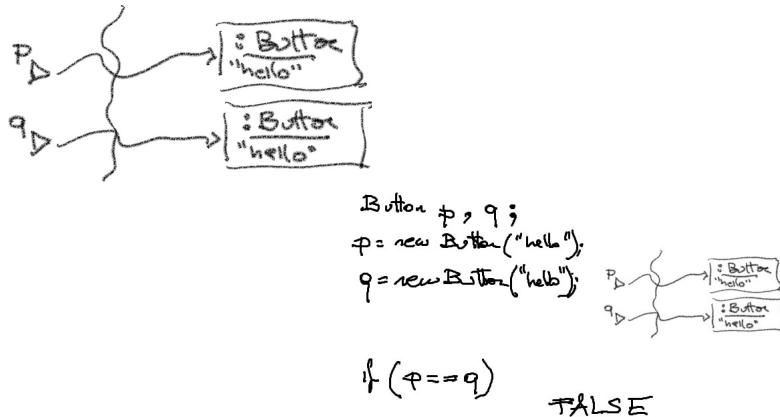
c2.xc = 2



28/203

■ Checking Objects for Equality

- the `==` operator tests whether two variables refer to the same object (this is about identity)
- classes often provide an `equals()` method, which tests whether two objects have equal values
- a third concept of equivalence relies on identity in the database storage



29/203

more on the constructor - *default*

■ each class in Java has at least one constructor method

■ if no constructors are defined

- a default constructor is created
- the default constructor takes no arguments and performs no special initialization, which is left to the invoking code

```
Circle c = new Circle(); //note the () syntactic modifier  
c.x = 1.414; c.y = -1.0; c.r = .25;
```

■ but, if at least one constructor is defined and it has arguments

- the default constructor is not created, and the class has no void constructors

30/116

- A class may have multiple methods with the same name
 - with different signatures and returned type equal or subtyped
 - calls are routed to the method with matching signature

```
public class Circle {  
    public double x, y, r;  
    public Circle(double x, double y, double r)  
        { this.x = x; this.y = y; this.r = r; }  
    public Circle(double r) { x = 0.0; y = 0.0; this.r = r; }  
    public Circle(Circle c) { x = c.x; y = c.y; r = c.r; }  
    public Circle() { x = 0.0; y = 0.0; r = 1.0; }  
    ...  
}
```

- Overloading is easily managed by the compiler
 - just a convenience for the programmer
 - an imperfect mechanism, not sufficient for all the needs
 - often applied to constructors, motivated by the constraint on the name

- `this` is a keyword, an expression that returns a reference to "this object"
 - "this object" is the object in which the method using `this` runs
 - `this` can be used to refer to the fields of this object

```
public double area() { return 3.14159 * this.r * this.r; }
```
 - `this` becomes necessary to disambiguate cases where a data field has the same name as a formal parameter

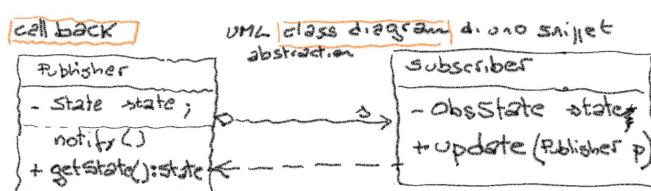
```
public class Circle {  
    public double x, y, r;  
    public Circle(double x, double y, double r) {  
        this.x = x; this.y = y; this.r = r; }  
}
```
 - `this` is often used by an object to pass its reference to enable a call-back (a fundamental idiom of Java, e.g. used in the Observer pattern)

- **this()** can be used in a constructor to use another constructor
 - may only appear as the first statement in a constructor
 - can serve to avoid duplication by reuse of a basic (helper) constructor
 - (note that () is a syntactic modifier that gets a function from an expression)

```
public Circle(double x, double y, double r)
{this.x = x; this.y = y; this.r = r; } //basic constructor

public Circle(Circle c)
{return this(c.x, c.y, c.r); } // copy constructor
```

- (sketches in 2020)



callback idiom:

- un Publisher invoca update(..) su un Subscriber, e gli passa il proprio indirizzo (this)
- il Subscriber può invocare il Publisher (e.g. invoca getState() per ricevere lo stato)

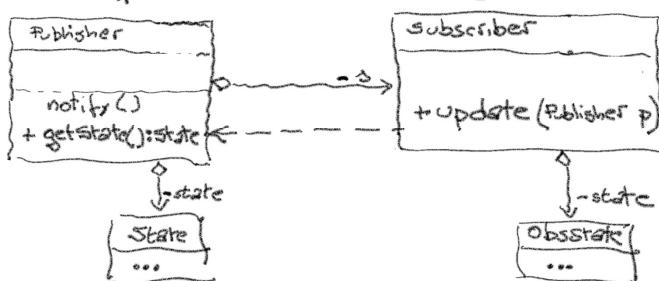
lo schema UML sotto è una implementazione:
(close enough)

```
class Publisher{
    private State state=viewState(...);
    private Subscriber s;
    ... // set Subscriber reference, somehow
    public void notify(void)
    {
        s.update(this);
    }
}

class Subscriber{
    private ObsState state=new ObsState(...);
    public void update(Publisher p)
    {
        ... = p.getState();
        ...
    }
}
```

35/203

• different UML abstractions



36/203

• idiom of a language

- un modo ricorrente, stile, virtuoso, ... di usare dei contesti di un linguaggio
- e.g. la call back è un idiom *java* che usa `this`.
- e.g. in C,

`int x;`
...
`... f(&x)...`

delego `f()` a realizzare un side effect sulla variabile `x` passandogli (non `x` bensì) `&x` (i.e. un'expr che restituisce l'indirizzo di `x`)

- e.g. in C, un idiom molto usato è uso il doppio prototipo per delegare side effects su un prototipo

Invece, non farò idiomi stili che richiedono il doppio prototipo.

37/203

- e.g. altri idiomi di *java*:

- copia difensiva in costruttore di copia
- oggetto immutabile `final` e `final`
- il Singleton con costruttore privato e qualche statica
- costruttore di copia

...
...

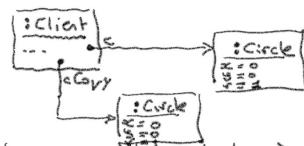
38/203

Sketches on the copy constructor idiom

Costruttori di copia

```
public class Circle {
    public double x, y, r;
    public Circle(double x, double y, double r) {
        this.x = x; this.y = y; this.r = r;
    }
    public Circle(double r) { x = 0.0; y = 0.0; this.r = r; }
    public Circle(Circle c) { x = c.x; y = c.y; r = c.r; }
    public Circle() { x = 0.0; y = 0.0; r = 1.0; }
}
public Circle(double x, double y, double r)
(this.x = x; this.y = y; this.r = r); // basic constructor
public Circle(Circle c)
{return this(c.x, c.y, c.r);} // copy constructor
```

2. Object of type Client



nel Client ~~do~~ in qualche metodo è scritto

~~some()~~

... Circle c = new Circle(0,0,1);
... c.setCircle(...)

Circle cCopy = new Circle(c);

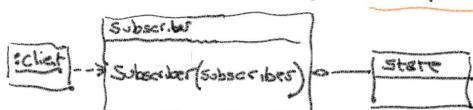
}

Remark: this is about

identity vs equality

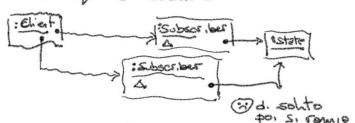
39/203

- un altro idiommo (o volendo una specializzazione del costruttore di copia)
è il costruttore di copia deepCopy

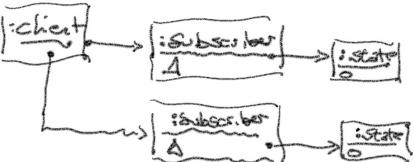


Si illustra bene su un Object Diagram

se la copia è shallow



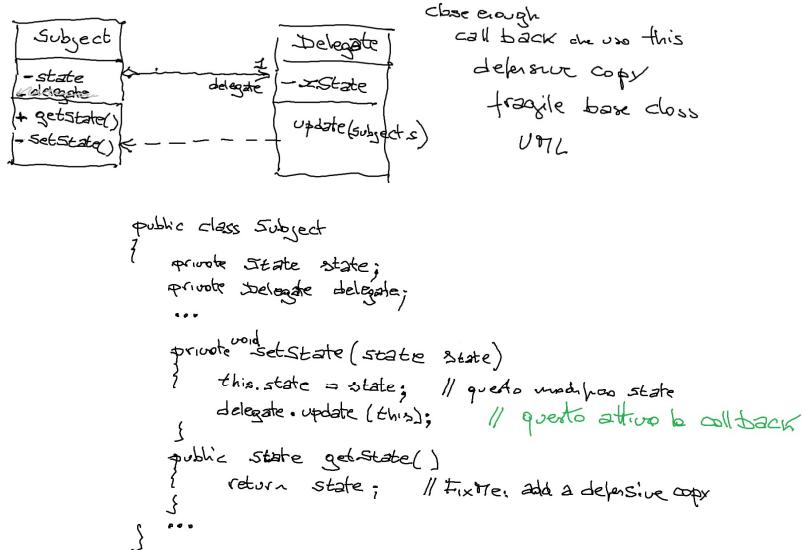
se la copia è deep



40/203

sketches on various concepts starting from *this* and *callback*

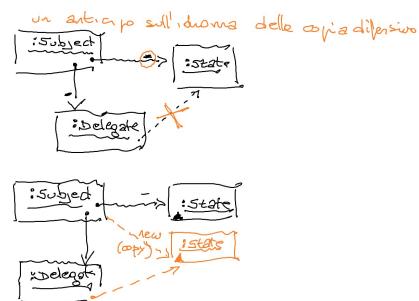
- (similar sketches in 2021)



41/203

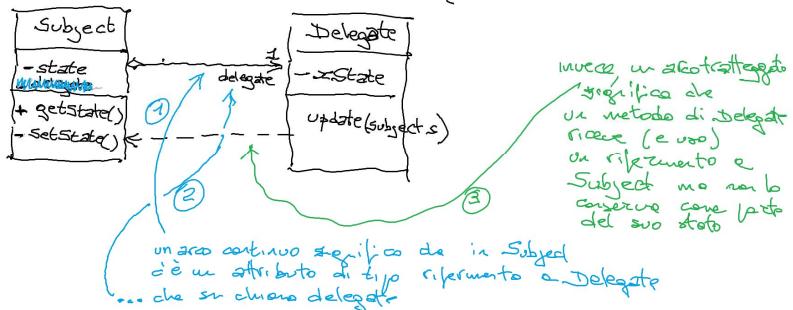
```

public class Delegate
{
    private State xState;
    public void update(Subject sub)
    {
        xState = sub.getState(); // questo è lo callback
    }
    ...
}
    
```



42/203

■ ... circa UML



43/203

■ null is a reserved value that indicates "absence of reference"

- It is not equal to zero, and cannot be cast to any primitive type
- it is the default initial value for variables of all reference types
- It can be assigned to variables of any reference type

▪ *TBD: is this an exception to strong type checking, or is null a reference to Object?*

44/116

■ Creating Arrays

- uses new, and specifies how large the array should be

```
byte octet_buffer[] = new byte[1024];  
Button buttons[] = new Button[10];
```
- creating an array does not create objects referenced by references in the array

```
for (count=0; count<10; count++)  
    buttons[count] = new Button(...);
```

TBD: Add a picture here showing references

- The elements of the array are initialized to the default value for their type (0 for primitive types, null for reference types)

■ An array can also be created with a static initializer

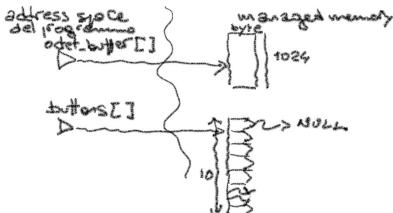
- ```
int lookup_table[] = {1, 2, 4, 8, 16, 32, 64, 128};
```
- the elements specified in an array initializer may be arbitrary expressions (this is different than in C, where they must be constant expressions)

■ Destroying arrays

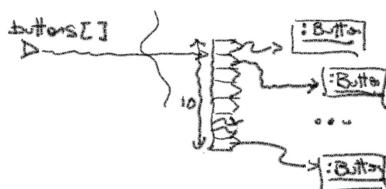
- Arrays are automatically garbage collected, just like objects are (see later the concept of garbage collection)

■ (sketches del 2020)

```
byte octet_buffer[] = new byte[1024];
Button buttons[] = new Button[10];
```



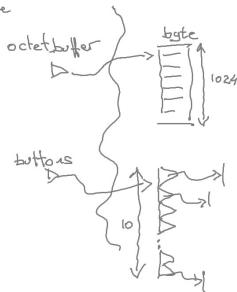
```
Button buttons[] = new Button[10];
for (count=0; count<10; count++)
 buttons[count] = new Button(...);
```



## ■ (same sketches nel 2021)

### Creating Arrays

- uses new, and specifies how large the array should be
  - byte octet\_buffer[] = new byte[1024];
  - Button buttons[] = new Button[10];
- creating an array does not create objects referenced by references in the array
  - for(count=0;count<10;count++)  
  buttons[count]=new Button(...);
- The elements of the array are initialized to the default value for their type (0 for primitive types, null for reference types)



47/203

## ■ Accessing Array Elements

- Array access in Java is just like array access in C
  - access an element of an array by putting an integer-valued expression between square brackets after the name of the array:

```
int a[] = new int[100];
a[0] = 0;
for(int i = 1; i < a.length; i++) a[i] = i + a[i-1];
```
- Notice how we computed the number of elements of the array in this example by accessing the length field of the array.
- In all Java array references, the index is checked to make sure it is not too small (less than zero) or too big (greater than or equal to the array length).
- If out of bounds, an `ArrayIndexOutOfBoundsException` is thrown.

48/116

- Are Arrays Objects?
  - arrays use the object syntax `.length` to refer to their length.
  - Arrays may also be assigned to variables of type `Object`, and the methods of the `Object` class may be invoked for arrays.
  - Java defines enough special syntax for arrays, however, it is still most useful to consider them a different kind of reference type
- Arrays are also reference types
  - assigning an array simply copies a reference to the array.
  - To actually copy the values stored in an array, you must assign each of the values individually or use the `System.arraycopy()` method.

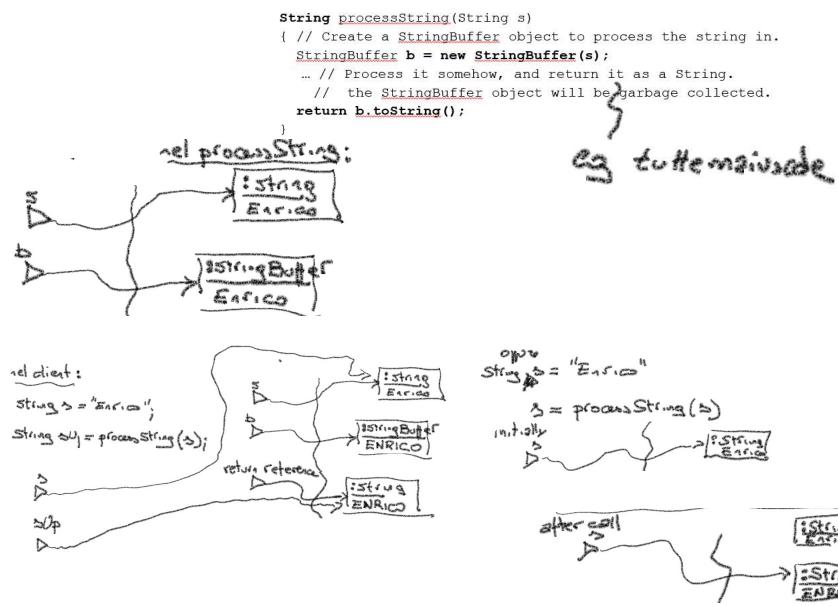
- Multidimensional Arrays
  - implemented as arrays-of-arrays, as they are in the c language
  - You allocate a multidimensional array with `new` by specifying the appropriate number of elements (between square brackets)  
`byte TwoDimArray[][] = new byte[256][16];`
  - Declares a variable named `TwoDimArray`. This variable has type `byte[][]` (array-of-array-of-byte).
  - Dynamically allocates an array with 256 elements. The type of this newly allocated array is `byte[][]`. Each element of this new array is of type `byte[]`
  - Dynamically allocates 256 arrays of bytes, each of which holds 16 bytes, and stores a reference to these 256 `byte[]` arrays into the 256 elements of the `byte[][]` array allocated in the second step. The 16 bytes in each of these 256 arrays are initialized to their default value of 0.

▪ *TBD: you really need a multidimensional array ?  
Sheldon Cooper: Why do we have a geology book?  
Leonard, did you throw a children's party while I was in Texas?*

- Strings in Java are instances of the **java.lang.String** class
  - a complete set of methods for string handling and manipulation
 `length(), charAt(), equals(), compareTo(), indexOf(), lastIndexOf(), substring(), ...`
  - <http://docs.oracle.com/javase/8/docs/api/>
- Strings are not null-terminated arrays of characters as they are in C
  - yet, to some extent they are treated by the compiler as primitive types
  - it automatically creates a String object when it encounters a double-quoted constant in the program.
  - the language defines the + operator for String concatenation.
- String objects are **immutable**
  - If you need to modify a String, you have to create a StringBuffer

```
String processString(String s)
{ // Create a StringBuffer object to process the string in.
 StringBuffer b = new StringBuffer(s);
 ... // Process it somehow, and return it as a String.
 // the StringBuffer object will be garbage collected.
 return b.toString();
}
```

51/116



52/203

## sketches on garbage collection

during execution  
when local reference variables  
are released at control returns  
or when they are side effected  
changing the object they refer to



some objects in the managed memory space  
may remain dangling/unreachable

in progettivo finale  
un oggetto irraggiungibile  
è come di scarica

progettivo finale:  
come esegue il programma,  
che risultato produce  
è il fenomeno desiderato

però l'oggetto occupa memoria  
e quindi condiziona lo progettivo non funzionante  
(eg performance)

53/203

## object destruction by garbage collection

- the Java Runtime Environment (JRE)  
maintains a map of objects in memory
  - which objects have been allocated
  - which is the type in which they were created
  - which reference variables refer to which objects
    - *TBD: add something here about the organization of the memory map, and about the classes Class and Object*
- JRE can thus detect objects that are not reachable from any reference
  - including cycles of object references  
that are not reachable to by any live objects
- The Java garbage collector detects and destroy all these objects
  - runs as a low-priority thread during idle times,  
or when the interpreter has run out of memory.
  - garbage collection is not for efficiency, it is rather for abstraction
  - makes programming easier and less error-prone.

54/116

## object destruction and garbage collection

- Garbage collection promotes a liberal use of temporary objects
- a typical idiom
  - in the body of a method definition,  
an object is created and its reference is assigned to a local variable
  - ... when the method terminates, the local variable is removed from the stack,  
and the referred object becomes ready for garbage collection
- ```
String processString(String s)
{ // Create a StringBuffer object to process the string in.
    StringBuffer b = new StringBuffer(s);
    ... // Process it somehow, and return it as a String.
    // the StringBuffer object will be garbage collected.
    return b.toString();
}
```
- garbage collection enables abstraction
on the construction of objects and their lifecycle
 - relevant for effective usage of *constructional* Design Patterns
 - one of the aspects where Java outperforms C++

55/116

(try to) anticipate garbage collection

- The code can try to anticipate garbage collection by nulling references to obsolete (heavy) objects

```
public static void main(String argv[])
{ int big_array[] = new int[100000];
    // Do some computations with big_array and get a result.
    int result = compute(big_array);
    // assume we no longer need big_array.
    // by assigning null to big_array, the garbage collector can
    // reclaim memory before the method termination
    big_array = null;
    ... // do other things, not requiring big_array
}
```
- Yet, nulling out object references should be an exception, not the norm
 - garbage collection will run eventually, out of the programmer control
 - the real golden rule is “do not mess with memory management”
 - anti-pattern: the fact that elements[size] is out of scope is in the intended invariant of the data structure, not enforced at the language level
 - whenever a class manages its own memory, memory leaks are possible
 - similar common sources of memory leaks are caches, listeners, callbacks, ...

56/116

- *Introduce the concept of Idiom, and the reference book Effective Java by Joshua Bloch and its items and its educated but also over-complicated style and its references to the ground choices in the architecture of standard libraries*

- A more complex example about memory leakage
 - a stack implemented over an array of references with an integer size as TopOfStack
 - at each pop(), the stack decreases but the popped reference is still in memory:
it is logically not valid according to the structure invariant, but the garbage collector does not know about that, and the referenced object is not removed (unintentional object retention)
 - the final effect is that size grows and decreases, but the length of elements can only be increased

TBD: l'esempio puo' essere molto migliorato, per evidenziare l'idioma su un caso di minima complessita'. La complessita' di un array di dimensione variabile e' irrilevante. Il tema e' avere uno stack di references, a cui sono associati oggetti, che non possono essere revocati quando il TOS li rende comunque irrilevanti

Effective Java Item 5: Eliminate obsolete object references – 2/3

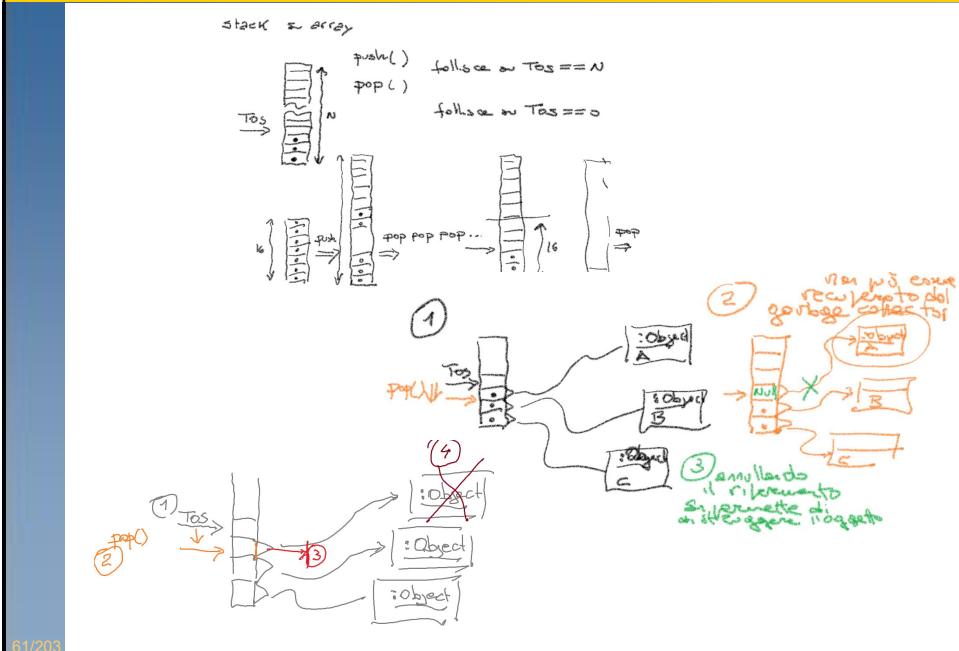
```
public class Stack {  
    private Object[] elements;  
    private int size = 0;  
    private static final int DEFAULT_INITIAL_CAPACITY = 16;  
    public Stack() { elements = new Object[DEFAULT_INITIAL_CAPACITY];}  
    public void push(Object e) { ensureCapacity(); elements[size++] = e; }  
    public Object pop() {  
        if (size == 0) throw new EmptyStackException();  
        return elements[--size]; }  
        // doubles the array capacity each time it needs to grow  
    private void ensureCapacity() {  
        if (elements.length == size)  
            elements = Arrays.copyOf(elements, 2 * size + 1);  
            // creates a new array and copies each element onto  
    }}  
59/66
```

Effective Java Item 5: Eliminate obsolete object references – 3/3

- the unintentional object retention can be avoided by nulling references out of use

```
public Object pop() {  
    if (size == 0) throw new EmptyStackException();  
    Object result = elements[--size];  
    elements[size] = null; // Eliminate obsolete reference  
    return result;  
}
```
- Yet, despite item #5,
nulling out object references should be an exception, not the norm
 - garbage collection will run eventually, out of the programmer control
 - the real golden rule is “do not mess with memory management”
 - anti-pattern: the fact that elements[size] is out of scope is in the intended invariant of the data structure, not enforced at the language level
 - whenever a class manages its own memory, memory leaks are possible
 - similar common sources of memory leaks are caches, listeners, callbacks, ...

sketches on item #5 eliminate obsolete references



61/203

object destruction and finalization

Object Finalization

- a finalizer method may perform needed closure operations on resources held by an object, that the garbage collector can't free (e.g. a file handle)
- the finalizer, is invoked before the system garbage collects the object.

```
/**
 * Closes the stream when garbage is collected.
 * Checks the file descriptor first
 */
protected void finalize() throws IOException {
    if (fd != null) close();
}
```

Finalizers are not used by "real Java programmers"

- the Java interpreter may exit without garbage collecting all outstanding objects, so some finalizers may never be invoked, neither it is guaranteed any order
- a finalizer method may "resurrect" an object by storing the `this` pointer, but the finalizer method is never invoked more than once.
- A finalizer may declare that it may throw an exception, which, if uncaught, will be ignored by the system.

62/116

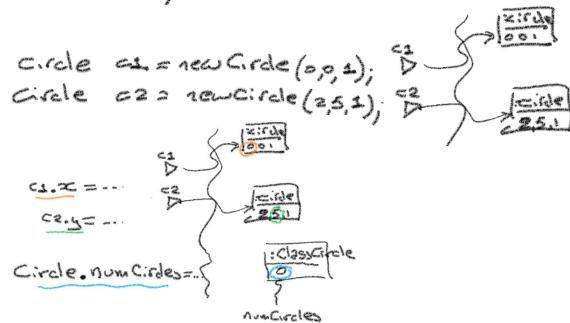
- What we saw up to here are called "instance variables"
 - a variable for each object, i.e. for each instance of the class


```
public class Circle {
    public double x, y, r;
    ... } // each instance of Circle has its own x, y, r
```
 - the so-called "state" of the object: r is the radius of this object
- A "class variable" is associated with the class
 - only one copy of the variable, initialized when the class is loaded
 - does not refer to the state of any specific instance
 - exists and can be used even if there is no object instantiated in the class
 - declared with the `static` modifier

```
public class Circle {
    static int num_circles = 0; //class var: counts Circle instances
    public double x, y, r; // instance vars: center and radius
    ...
}
```

63/116

```
public class Circle {
    double r; // instance variables
    double x; // re è uno uterino
    double y; // per ogni istanza di Circle
    public Circle(double x, double y, double r) {
        this.r = r;
        this.x = x;
        this.y = y;
    }
    public Circle() {
        static int numCircles = 0;
        ...
    }
}
```



64/203

- a class variable is much like a kind of global variable
 - Without possible name conflicts,
and possibly restrained in the scope of class methods,
 - but with all the other drawbacks

*TBD: say something about how Class variables are represented
and about the Class type
(heap vs permgen area)*

- An Example

```
public class Circle {  
    static int num_circles = 0; //class var: #of Circle instances  
    public double x, y, r; // instance vars: center and radius  
  
    public Circle(double x, double y, double r) {  
        this.x = x; this.y = y; this.r = r;  
        num_circles++; //increment at each constructor call  
    }  
    public Circle(double r) { this(0.0, 0.0, r); }  
    public Circle(Circle c) { this(c.x, c.y, c.r); }  
    public Circle() { this(0.0, 0.0, 1.0); }  
    public double circumference() { return 2 * 3.14159 * r; }  
    public double area() { return 3.14159 * r*r; }}
```

- Access to class variables expressed through the class name

```
System.out.println("Num.of circles: " + Circle.num_circles);  
■ note that Circle is uppercase, and it thus identifies a class
```

■ Constants can be encoded as immutable class variables

```
public class Circle {  
    public static final double PI = 3.14159265358979323846;  
    public double x, y, r;  
    // ... etc....  
}
```

- **final** modifier: this variable can not be muted after initialization
- as in the C convention, constant names are capitalized

■ Some consequences (about intention enforced in language idioms)

- PI is not about "this" Circle, it is about any Circle (static)
- can use PI also without having any instance of Circle (static)
- PI can also be for other objects that are not Circles (public)
- PI is a constant (final)

■ The Java compiler is smart about variables declared both **static** and **final**

```
double circumference = 2 * Circle.PI * radius;
```

- the compiler precomputes the value 2 * Circle.PI

67/116

■ Class methods (aka static methods)
operate in the context of class variables

- they are defined using the **static** declaration modifier
- have no direct access to the state of any specific object instance

```
public class Circle {  
    public double x, y, r;  
    // A class method. Returns the bigger of two circles.  
    public static Circle bigger(Circle a, Circle b) {  
        if (a.r > b.r) return a; else return b;  
    }  
    ...  
}
```

■ Some consequences

- can be invoked also if no object has been instantiated in the class
(which becomes relevant for some fundamental patterns, e.g. the Singleton)
- make explicit that an operation is neutral w.r.t. specific objects
- much like functions in C, which are by construction declared at the global level

68/116

- Example: the same operation with an instance or class method
 - about the independence from the instance context and the value of neutrality

```
public class Circle {  
    public double x, y, r;  
    // An instance method. Returns the bigger of two circles.  
    public Circle bigger(Circle c) {  
        if (c.r > r) return c; else return this;  
    }  
    // A class method. Returns the bigger of two circles.  
    public static Circle bigger(Circle a, Circle b) {  
        if (a.r > b.r) return a; else return b;  
    }  
    ...  
}  
  
Circle a = new Circle(2.0);  
Circle b = new Circle(3.0);  
Circle c = a.bigger(b);           // or, b.bigger(a);  
Circle d = Circle.bigger(a,b);   // much more neutral
```

- Example: using/recognising a class method

```
public class Circle {  
    double x, y, r;  
    // is point (a,b) inside this circle?  
    public boolean isInside(double a, double b)  
    {  
        double dx = a - x;  
        double dy = b - y;  
        double distance = Math.sqrt(dx*dx + dy*dy);  
        if (distance < r) return true;  
        else return false;  
    }  
    .  
    . // Constructor and other methods omitted.  
    .  
}
```

- Math is uppercase, and it is thus a classname, not an object reference
- sqrt() is the name of a class method defined in Math
- to evaluate a square root, you don't need a state, and you don't need an object be created

Enrico Vicario
SWE – AA22/23

class methods

- `System` is another class that defines only class methods
 - <http://docs.oracle.com/javase/8/docs/api/> (`java.lang.System`)

```
System.out.println("Hello world!");
```

- `System` is a class, with a class-variable named `out`.
- `System.out` refers to an object of type `PrintStream`
- The class `PrintStream` has an instance method named `println()`.

The screenshot shows two Java API documentation pages side-by-side.

Left Page (System Class):

- Class Summary:** Shows `compact1`, `compact2`, `compact3` from `java.lang`.
- Class System:** Shows `java.lang.Object` and `java.lang.System`.
- Field Summary:** Shows three static fields: `err` (error output stream), `in` (standard input stream), and `out` (standard output stream).
- Method Summary:** Shows the `println()` method which terminates the current line by writing the line separator string.

Right Page (PrintStream Class):

- Class PrintStream:** Shows `java.lang.Object`, `java.io.OutputStream`, `java.io.FilterOutputStream`, and `java.io.PrintStream`.
- All Implemented Interfaces:** Shows `Closeable`, `Flushable`, `Appendable`, and `AutoCloseable`.
- Direct Known Subclasses:** Shows `LogStream`.
- Method Summary:** Shows `println()` with its description: "Terminates the current line by writing the line separator string."

71/116

Enrico Vicario
SWE – AA22/23

class methods

- Class variables are initialized when the class is first loaded.


```
static int num_circles = 0; // once, at class loading
float r = 1.0;           // at each instance creation
```
- static initializer method
 - without arguments or return value, without name, invoked when the class is loaded
 - used to perform any needed initialization on class variables

```
public class Circle {
    static private double sines[] = new double[1000];
    static private double cosines[] = new double[1000];
    // Here's a static initializer "method"
    static {double x, delta_x; int i;
        delta_x = (Circle.PI/2)/(1000-1);
        for(i = 0, x = 0.0; i < 1000; i++, x += delta_x) {
            sines[i] = Math.sin(x); cosines[i] = Math.cos(x); }
    } ...
}
```

 - any number of static initializer blocks are allowed, automatically packed in their order into a single class initialization routine, executed when the class is first loaded

72/116

```
// draw the outline of a circle using trigonometric functions.  
// pre-compute a bunch of values to improve performance  
  
public class Circle {  
    // Here are our static lookup tables, and their initializers.  
    static private double sines[] = new double[1000];  
    static private double cosines[] = new double[1000];  
    // Here's a static initializer "method" that fills them in.  
    // Notice the lack of any method declaration!  
    static {  
        double x, delta_x;  
        int i;  
        delta_x = (Circle.PI/2)/(1000-1);  
        for(i = 0, x = 0.0; i < 1000; i++, x += delta_x) {  
            sines[i] = Math.sin(x);  
            cosines[i] = Math.cos(x);  
        }  
    }  
    . . . // The rest of the class omitted.  
    . . .  
}
```

73/116

- static factory method (aka static constructor)
 - a static method that returns an instance of the class
 - to be used in addition-to the usual class constructors

```
public class Rectangle  
{    public Rectangle(){...} // a constructor  
    public static Rectangle getInstance ()  
    // a static factory method  
    {        Rectangle r;  
        ...  
        return r;  
    }  
}  
// client code:  
Rectangle r1 = new Rectangle();  
// creation through a standard constructor  
Rectangle r2 = Rectangle.getInstance ();  
// creation through a static factory method  
// which, much probably, uses Rectangle()
```

74/66

Item #1 - static factory methods - consequences

- static constructors do not require an object instance for being invoked
 - which would be inconvenient as this serves just to create an object
 - (e.g. used in the Singleton pattern)
- each static factory method can have a specific descriptive name
 - as opposed to standard constructors, which have the name of the class, (with different signatures)
 - e.g. create the embedding or the embedded Rectangle of a Circle
- static constructors are much like a c function, globally visible
- static constructors have no instance context
 - they only rely on class (static) fields and passed actual parameters
 - (like default implementations in Java8+)

75/66

Item #1 - static factory methods - consequences

- static factory methods can control the way how objects are created
 - they can return an object of any subtype of their return type
 - e.g. class A has 2 subclasses A1 and A2, it features a static constructor getInstance() which returns a reference to A, concretely instantiated as A1 or A2 depending on parameter values.
(e.g. two representations for a matrix, filled or sparse, chosen according to number of zeros)
(e.g. in the Strategy design pattern)
- instance-controlled classes
 - while a standard constructor always creates one object, a static factory method can return a reference to an existing instance, or more generally can limit the number of instances
(e.g. in the Singleton design pattern)

76/66

Item #1 - static factory methods - consequences

- static factory methods are not distinguishable as constructors
 - leads to a code where object creation is not evident
- ... but, this might be useful for abstraction,
- ... and anyway, naming conventions can help
 - `getInstance`: the general case
 - `getType`: used when the factory method is in a different class *Type*
 - `newInstance`: each instance returned is distinct from all others.
 - `newType`: like `newInstance`, when the factory method is in a different class.
 - `valueOf`: returns an instance with the same value as its parameters.

```
public static Boolean valueOf(boolean b) {  
    return b ? Boolean.TRUE : Boolean.FALSE; }
```

77/66

Item #1 - static factory methods – example on controlling instances

- TBD: a composition of elements in a finite number of cases (like the pieces of a Lego box)
 - Each piece is created as immutable and shared by multiple references (e.g. in a case of the Composite design pattern)
 - *NB: condividere l'oggetto funziona bene per oggetti immutabili, per i quali l'uguaglianza dei valori ad un certo tempo si conserva per sempre.*

78/66

- Subclassing, inheritance, and substitutability
- Class hierarchy, Object, and construction chaining
- Override, fragile base class, composition vs inheritance
- Immutable objects

- Responsibilities of a class can be extended by *composition*
 - e.g. `GraphicCircle` develops `Circle`, by adding a method `draw()` method

```
public class GraphicCircle {  
    public Circle c;  
    // existing methods implemented by forwarding  
    public double area() { return c.area(); }  
    public double circumference() { return c.circumference(); }  
    // added graphic variables and methods  
    public Color outline, fill;  
    public void draw(DrawWindow dw) { ... }  
}
```
 - *TBD: add some UML to draw the object adapter, or wrapper*
- composition has some negative consequences
 - all existing methods must be implemented (by forwarding)
 - the extended type `GraphicCircle` cannot replace `Circle` in the type check (which is the real drawback, as we will discover later)

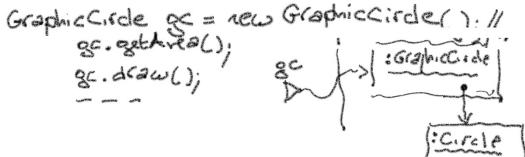
sketches on extending responsibility by composition



// nel Client
// in qualche metodo

```

public class GraphicCircle {
    Circle c;
    Color col;
    void draw() {
        ...
    }
    double getArea() {
        return c.getArea(); // forwarding
    }
    GraphicCircle(void) {
        c = new Circle();
        ...
    }
    // il costruttore assume la responsabilità
    // di creare il Circle e installare le dipendenze
}
  
```



81/203

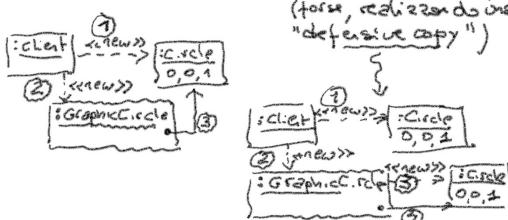
... offre la responsabilità di creare il Circle
al posto essere affidato al Client

// nel Client

```

Circle c = new Circle(...);
GraphicCircle gc = new GraphicCircle(c);
  
```

▲ occhio: noto lo dipende,
(forse, realizzando una
"defensive copy")



// è un GraphicCircle ma non un Circle
// i.e. dove è atteso il riferimento a un Circle
// sarà trovato invece un GraphicCircle

```

Circle c;
GraphicCircle gc = new GraphicCircle();
c = gc; // si rompe!
  
```

82/203

- in a different way, responsibilities can be extended by *subclassing*

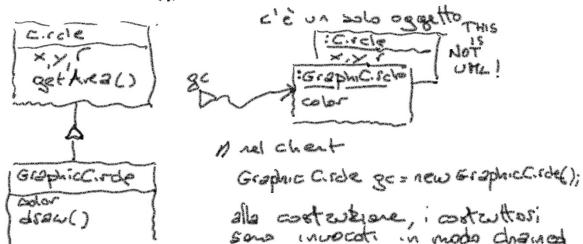
```
public class GraphicCircle extends Circle {  
    Color outline, fill;  
    public void draw(DrawWindow dw) {  
        dw.drawCircle(x, y, r, outline, fill);  
    }  
  
    // In the client:  
    GraphicCircle gc = new GraphicCircle();  
    double area = gc.area();  
    Circle c = gc;
```

TBD: add some UML to draw the class adapter

- Circle is said to be the **superclass** (or base class) of GraphicCircle

- subclassing can serve for two aims: inheritance and substitutability

- works just as if the area() method was defined in GraphicCircle, avoiding the need of implementing forwarding methods
- a GraphicCircle object is a legal instance of Circle
- substitutability is by far the most relevant positive consequence



A real cheat
GraphicCircle gc = new GraphicCircle();
alle costruzioni, i costruttori sono invocati in modo chained

Superclasses, Object, and the Class Hierarchy

- each class can declare a single superclass
 - If no superclass is declared, the superclass is the class `Object`
 - `Object` is the only class that does not have a superclass.
- classes thus form a class hierarchy, with `Object` at the root
- no multiple inheritance is allowed, as the superclass is unique

Constructor chaining

- Sometimes, subclass construction needs functionalities already implemented in the base class
 - E.g. in the construction of a `GraphicCircle`, we may duplicate the code of the `Circle` constructor

```
public GraphicCircle(double x, double y, double r,
    Color outline, Color fill)
{ this.x = x; this.y = y; this.r = r;
    this.outline = outline; this.fill = fill;
}
```
- The `super` reserved keyword permits the constructor of a base class be reused in the construction of subclasses
 - ```
public GraphicCircle(double x, double y, double r,
 Color outline, Color fill)
{ super(x, y, r);
 this.outline = outline; this.fill = fill;
}
```
  - `super.super` is not legal
  - The call to the superclass constructor must be the first statement in the constructor

■ Constructor Chaining

- If the first statement in a constructor is not a call to a constructor of the superclass, then Java implicitly inserts the call `super()`
- If the superclass does not have a constructor that takes no arguments, this causes a compilation error.
- If the first line of a constructor, C1, uses the `this()` syntax to invoke another constructor, C2, of the class, Java relies on C2 to invoke the superclass constructor, and does not insert a call to `super()` into C1.
- when we create a new instance of the `GraphicCircle` class, the body of the `Object` constructor runs first, followed by the body of the `Circle` constructor, and finally followed by the body of the `GraphicCircle` constructor.
- constructor calls are "chained", order from subclass to superclass on up to `Object`
- the body of the `Object` constructor always runs first, followed by the body of its subclass, and on down the class hierarchy to the class that is being instantiated.

■ Finalizer Chaining?

- The finalizer of a class does not automatically invoke the finalizer of its superclass.

■ The Default Constructor

- if a class is declared without any constructor at all, Java implicitly adds a default constructor, which does nothing but invoke the superclass constructor.

```
public GraphicCircle() { super(); }
```
- if the superclass, `Circle()`, did not declare a no-argument constructor, then this automatically inserted default constructor would cause a compilation error.
- If a class does not define a no-argument constructor, then all of its subclasses must define constructors that explicitly invoke the superclass constructor with the necessary arguments.
- *More precisely: if the base class has no constructor at all, then the default will be constructed and all will work fine. Yet, the base class declares a constructor with arguments but not a void constructor, then no default constructor will be generated*
- If a class does not declare any constructor, it is given a `public` constructor by default.
- Classes that do not want to be publically instantiated, should declare a `protected` constructor to prevent the insertion of this `public` constructor.
- Classes that never want to be instantiated at all should define a `private` constructor.

■ Shadowed Variables

- the subclass declares a field with the same name and type as one declared in some base class
- Example: `GraphicCircle` extends `Circle` and *shadows* the variable `r`

```
public class GraphicCircle extends Circle {
 Color outline, fill;
 float r; // New variable. Resolution in dots-per-inch.
 public GraphicCircle(double x, double y, double rad,
 Color o, Color f) {
 super(x, y, rad); outline = o; fill = f;
 }
 public void setResolution(float resolution) { r = resolution; }
 public void draw(DrawWindow dw) { dw.drawCircle(x, y, r,
 outline, fill); }
}
```

■ can use this and super to disambiguate

```
this.r // Refers to the GraphicCircle resolution variable.
super.r // Refers to the Circle radius variable.
((Circle) this).r // does the same using an up-cast
```

■ ... but, better avoid shadowing variables

■ Shadowed variables across multiple extensions

- refer to a shadowed variable defined in a class that is not the immediate superclass: if `C` is a subclass of `B`, which is a subclass of `A`, and class `C` shadows a variable `x` that is also defined in classes `A` and `B`, then you can refer to these different variables from class `C`

```
x // Variable x in class C.
this.x // Variable x in class C.
super.x // Variable x in class B.
((B)this).x // Variable x in class B.
((A)this).x // Variable x in class A.
super.super.x // Illegal; does not refer to x in class A.
```

- you cannot refer to a shadowed variable `x` in the superclass of a superclass with `super.super.x`.

- ... are there also shadowed methods?
  - "shadowed" methods are called overridden methods,  
and, they play a much more relevant role than shadowed variables
- Overridden methods
  - the subclass defines a method with the same name, type, and signature as one defined in some superclass
  - override/overridden: overrule, replace in importance, give a command that cancels the effects of something, ...
  - do not mess overriding with overloading
- Examples of application
  - if `Ellipse` extends `Circle`,  
methods `area()` and `circumference()` need to be overridden  
(by the way, is this reasonable? ... Liskov Substitution Principle)
  - more complex schemes: a fake/default implementation in the base class, with override in subclasses where the method is appropriate but not in subclasses that are not concerned (see later about the Composite design pattern)

- Dynamic Method Lookup
  - the Java policy to determine the method implementation invoked
  - binding is driven by the type of the object, not by that of the object-reference
  - this is made possible by interpretation and memory management provided by the runtime environment
  - when the interpreter interprets the expression `s.area()` it dynamically looks for an `area()` method associated with the particular object referred to by the variable `s`.
  - Java does not have a `virtual` keyword; methods in Java are "virtual" by default.
  - Dynamic method lookup is fast, but it is not as fast as invoking a method directly.  
in a number of cases, Java does not need to use dynamic method lookup.
  - `static` methods cannot be overridden.
  - `private` methods are not inherited by subclasses
  - `final` methods are invoked directly
  - when a method of a `final` class is invoked through an instance of the class or a subclass of it, then it, too, can be invoked without the overhead of dynamic lookup.

- overridden methods cannot be referred to by cast (as for shadowed variables)

```
class A {
 int i = 1;
 int f() { return i; } }
class B extends A {
 int i = 2; // Shadows variable i in class A.
 int f() { return -i; } // Overrides method f in class A. }
public class override_test {
 public static void main(String args[]) {
 B b = new B();
 System.out.println(b.i); // Refers to B.i; prints 2.
 System.out.println(b.f()); // Refers to B.f(); prints -2.
 A a = (A) b; // Cast b to an instance of class A.
 System.out.println(a.i); // Now refers to A.i; prints 1;
 System.out.println(a.f()); // Still refers to B.f(); prints -2;
 }
}
```

- if Ellipse extends Circle, objects of types Ellipse can be cast into Circle and stored in an array of type Circle[]
- the array can then be visited invoking the proper area method for each element, without distinguishing them explicitly

- Invoking an Overridden Method

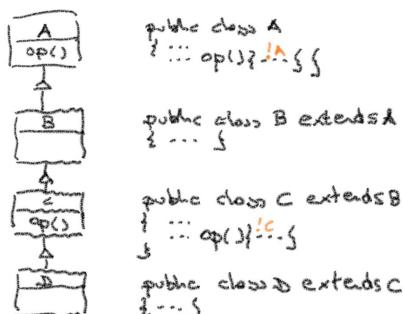
```
class A {
 int i = 1;
 int f() { return i; } // A very simple method. }
class B extends A {
 int i; // This variable shadows i in A.
 int f() { // This method overrides f() in A.
 i = super.i + 1; // It retrieves A.i this way.
 return super.f() + i; // And it invokes A.f() this way }}
```

- super can only be used to invoke overridden methods from within the class that does the overriding.
- super.super.f() is not legal Java syntax!
- If class A defines f(), B is a subclass of A, and C is a subclass of B that overrides method f, then super.f() invokes the overridden method from within class C.
- If classes A, B, and C all define method f, however, then calling super.f() in class C invokes class B's definition of the method.
- there is no way to write a program that invokes the Circle area() method on an object of type Ellipse. The only way to do this is to use super in a method within the Ellipse class.

■ Finalizer Chaining Revisited

- In Java, constructor methods are automatically chained, but finalizer methods are not.
- If your finalizer method does not invoke the superclass finalizer, the superclass finalizer never gets called
- you should be sure to invoke the superclass `finalize()` method.
- The best time to do this is usually after your `finalize()` method

```
super.finalize();
```



public class A  
{ ... op(); }  
A

public class B extends A  
{ ... }

public class C extends B  
{ ... op(); }  
C

public class D extends C  
{ ... }

// in our client

```
A a = new A();
B b = new B();
C c = new C();
D d = new D();

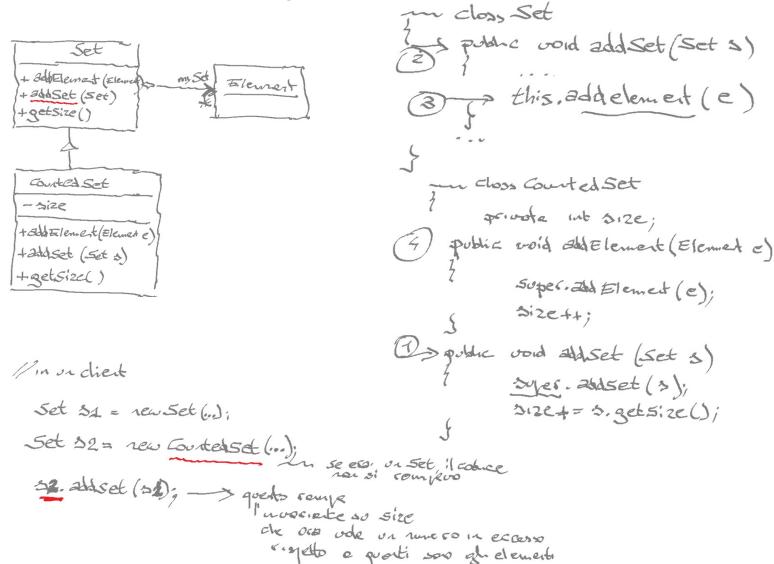
a.op(); // !A
b.op(); // !A
c.op(); // !C
d.op(); // !C
```

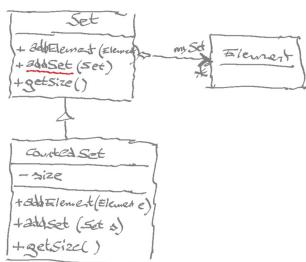
```

A a = new A();
A b = new B(); // abstraction:
 // b refers to a B
out Abstractor- // but it is a ref to A
p avere scelto il tipo dell'oggetto
(new B()) mi "dimentica"
straggo l'oggetto a) il suo tipo B
 // this is legal by
soluce che sceglie // substitutability
di uscire da b A c = new C();
 // i comportamento si.
 // c.op(); // !c
 // il binding è risolto
 // sul tipo (dinamico)
 // dell'oggetto
 // e non sul tipo (statico)
 // del riferimento
A d = new D();
d.op(); // !c

```

- ... sometimes it is not true that all that glitters is gold
- TBD: discuss here the fragile base class antipattern





*// in subclass*

Set s1 = new Set(...);

Set s2 = new CountedSet(...);

s2.addSet(s1); *in s2, in Set, il codice non si compone*

s2.addSet(s1); → questo rompe

*l'associazione fra size e addSet che ora vale un numero in eccesso*

*rispetto a quanti sono gli elementi*

*in class Set*  
② public void addSet(Set s)  
...  
    this.addElement(e)

*in class CountedSet*

③ private int size;

public void addElement(Element e)  
    super.addElement(e);  
    size++;

④ public void addSet(Set s)  
    super.addSet(s);  
    size+= s.getSize();

99/203

#### Item 14: Favor composition over inheritance

TBD: riallineare l'item al codice su fragile base class:

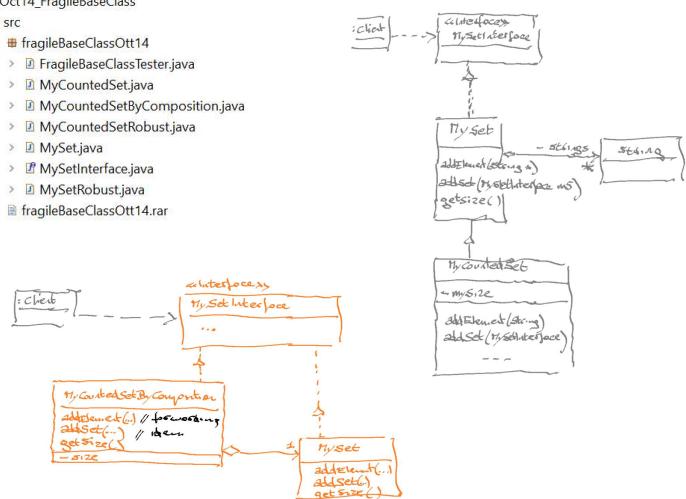
- Inizia con il paradigma open/closed, che in principio si può fare per ereditarietà o composizione.
- Ontologia errore/difetto/malfunzionamento
  - errore (lasciare aperta all'estensione una classe fragile, in nessuna misura estendere la classe in quel modo)
  - difetto (è costruito un po' per ciascuno nel costruttore o in addSet della classe estesa, e in addSet della classe base)
  - malfunzionamento (si rompe l'invariante che collega il contatore al numero di elementi)
- Presentare il codice per ereditarietà, poi discutere dei limiti di extension
- Poi c'e' la cura degli helper methods per evitare dipendenze
- Oppure documentare (item successivo)
- Oppure passare alla composizione, con il wrapper ...
- Mostrare l'implementazione per composizione.

100/66

## TBD: fragile base class reloaded – notes and sketches from the code

- TBD: starting from the Java code, create an UML model and re-discuss all the story

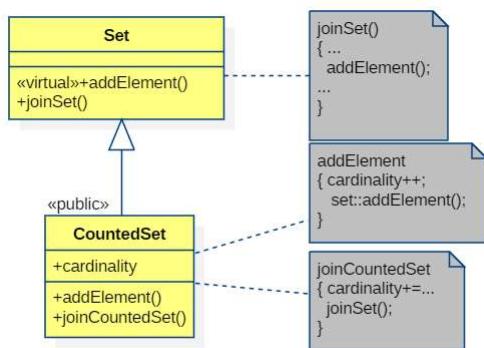
```
EvOct14_FragileBaseClass
src
fragileBaseClassOtt14
FragileBaseClassTester.java
MyCountedSet.java
MyCountedSetByComposition.java
MyCountedSetRobust.java
MySet.java
MySetInterface.java
MySetRobust.java
fragileBaseClassOtt14.rar
```



101/203

## Item 14: Favor composition over inheritance

- TBD move here The fragile base class problem



102/66

## Item 14: Favor composition over inheritance

- Implementation inheritance is a possible way to achieve code reuse
  - Open/closed principle
  - Safe within a package: subclass and superclass under the same control
  - Safe when extending classes specifically designed and documented for extension
- But,
  - inheritance violates encapsulation:  
a subclass depends on the implementation details of its superclass
  - inheriting from ordinary concrete classes across package boundaries, is dangerous
  - Fragile base class
    - *TBD: qui succede anche che sia la classe base a dipendere dalla derivata!*
- ... moreover, a superclass can acquire new methods in subsequent releases
  - If the base class changes, also its contracts can
  - ... how can the derived class still guarantee the substitution principle ?
  - A subclass must evolve in tandem with its superclass

103/66

## Item 14: Favor composition over inheritance

- Extend HashSet to add a counter

```
public class InstrumentedHashSet extends HashSet {
 private int addCount = 0;
 public InstrumentedHashSet() {}
 public InstrumentedHashSet(Collection c) {super(c);}
 public InstrumentedHashSet(int initCap, float loadFactor) {super(initCap, loadFactor);}
 public boolean add(Object o) {addCount++; return super.add(o);}
 public boolean addAll(Collection c) {addCount += c.size(); return super.addAll(c);}
 public int getAddCount() {return addCount;}}
```
- ... add a collection with 3 elements

```
InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
```
- Is this a case of fragile base class?
  - HashSet's addAll method is implemented on top of its add method, although HashSet, quite reasonably, does not document this implementation detail.
  - The rest follows from method lookup ...
- Can we change addAll in the InstrumentedHashSet ?
  - Would still depend on an internal not documented detail of HashSet

104/66

## Item 14: Favor composition over inheritance

- Instead of extending an existing class

- give your new class a private field that references an instance of the existing class.
  - *composition*: the existing class becomes a component of the new one.
  - *forwarding*: each instance method in the new class, invokes the corresponding method on the contained instance of the existing class and returns the results
  - *wrapper*: the new class encapsulates the existing one
- 
- TBD: esempio sullo schema della featurization
  - TBD: menzione del pattern di Dependency Injection
  - TBD: riferimento a ORM per ereditarietà

105/66

## Item 14: Favor composition over inheritance

```
public class InstrumentedSet implements Set {
 // ... leverages the existence of an interface Set, which captures the functionality of HashSet
 private final Set s; // private reference to the composed object
 private int addCount = 0;
 public InstrumentedSet(Set s) {this.s = s;} //reference installed by the constructor
 public boolean add(Object o) {addCount++; return s.add(o);}
 public boolean addAll(Collection c) {addCount += c.size(); return s.addAll(c);}
 public int getAddCount() {return addCount;}
 // Forwarding methods: each and every one :(
 public void clear() {s.clear();}
 public boolean contains(Object o) {return s.contains(o);}
 public boolean isEmpty() {return s.isEmpty();}
 public int size() {return s.size();} public Iterator iterator() {return s.iterator();}
 public boolean remove(Object o) {return s.remove(o);}
 public boolean containsAll(Collection c){ return s.containsAll(c);}
 public boolean removeAll(Collection c){ return s.removeAll(c);}
 public boolean retainAll(Collection c){ return s.retainAll(c);}
 public Object[] toArray() {return s.toArray();}
 public Object[] toArray(Object[] a) {return s.toArray(a);}
 public boolean equals(Object o) {return s.equals(o);}
 public int hashCode() {return s.hashCode();} public String toString() {return s.toString();}
}
```

106/66

## Item 14: Favor composition over inheritance

```
public class InstrumentedSet implements Set {
 // ... leverages the existence of an interface Set, which captures the functionality of HashSet
 private final Set s; // private reference to the composed object
 private int addCount = 0;
 public InstrumentedSet(Set s) {this.s = s;} //reference installed by the constructor
 ...
 public boolean contains(Object o) { return s.contains(o); }
 ...
}
```

- Set is an interface,
  - it is a "fortunate" fact that HashSet is built as an implementation of Set
  - the constructor of InstrumentedSet accepts any implementation of Set

```
Set s1 = new InstrumentedSet(new TreeSet(list));
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```
- can even be used to temporarily wrap an already existing Set

```
static void f(Set s) { // why is this static ? Is only about forwarding, without added state
 InstrumentedSet slnInst = new InstrumentedSet(s);
 ... // Within this method use slnInst instead of s
}
```

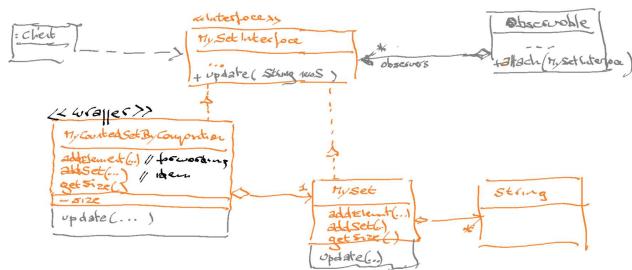
107/66

## Item 14: Favor composition over inheritance

- The SELF problem
  - the wrapped object *is not aware* of its wrapper,  
in a callback scheme it will pass a reference to itself, eluding the wrapper
  - Not suited for use in *callback frameworks*, wherein objects pass self-references to other objects for later invocations
    - *Would it be better moving here the discussion about frameworks versus toolkits?  
and the Hollywood principle of control inversion ?  
Apparently not!*
- performance impact of forwarding or memory footprint of wrapper objects
  - Usually not an issue, and in any case not a problem
- It is a bit tedious to write forwarding methods
  - partially offset by the fact that you have to write only one constructor

108/66

- an occurrence of the SELF problem
  - in the context of MyCountedSetByComposition
  - ... which was used before to illustrate how composition avoids the fragile base class problem
- an object mS:MySet is not aware of being wrapped by some object mCS:MyCountedSet
  - if mS invokes attach() passing "this",
  - ... it will eventually receive a direct notification on its update(),
  - ... it will add the String to the set, but it will not increment the counter in mCS



109/203

#### Item 14: Favor composition over inheritance

- Inheritance is appropriate only if the subclass really is a *subtype* of the superclass
  - "Is every *B* really an *A*?" If you cannot truthfully answer yes, *B* should not extend *A*
  - If the answer is no, it is often the case that *B* should contain a private instance of *A* and expose a smaller and simpler API:  
*A* is not an essential part of *B*, merely a detail of its implementation.
- If you use inheritance where composition is appropriate, you needlessly expose implementation details

110/66

## Item 15: Design and document for inheritance or else prohibit it

- ... a solution by documentation for the fragile base class problem
- the class must document precisely the effects of overriding any method
  - document self-uses of overridable methods:  
which overridable (non-final) methods are invoked,  
in what sequence,  
and how the results of each invocation affect subsequent processing  
(a sufficient menace to avoid self-uses :))
  - document any circumstances under which an overridable method can be invoked  
(e.g. invocations from background threads or static initializers)

*TBD: esercizio: fornire un esempio di come documentare l'implementazione del codice della fragile base class*

111/66

## Item 15: Design and document for inheritance or else prohibit it

- “This implementation.”
  - The description of a method that invokes overridable methods begins with “This implementation.”,  
so as to emphasize that documented behavior may be affected by override
  - E.g. in java.util.AbstractCollection, for public boolean remove(Object o)  
*Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element e such that (o==null ? e==null : o.equals(e)), if the collection contains one or more such elements. Returns true if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).*  
*This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method. Note that this implementation throws an UnsupportedOperationException if the iterator returned by this collection's iterator method does not implement the remove method.*
  - leaves no doubt that overriding the iterator method will affect the behavior of remove, and describes exactly how the behavior of the Iterator returned by the iterator method will affect the behavior of the remove method.
  - Note that since encapsulation is violated, documentation violates the principle that good API documentation describe *what* a given method does and not *how* it does it

112/66

## Item 15: Design and document for inheritance or else prohibit it

- a class may provide hooks into its internal workings in the form of judiciously chosen protected methods
  - E.g.: from java.util.AbstractList:  
`protected void removeRange(int fromIndex, int toIndex)`  
  
*"Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the ArrayList by (toIndex - fromIndex) elements. (If toIndex==fromIndex, this operation has no effect.) This method is called by the clear operation on this list and its sublists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the clear operation on this list and its subLists.*  
*This implementation gets a list iterator positioned before fromIndex and repeatedly calls ListIterator.next followed by ListIterator.remove, until the entire range has been removed. Note: If ListIterator.remove requires linear time, this implementation requires quadratic time."*
  - The protected removeRange method is documented so as to enable override; and thus increase the performance of the clear method

113/66

## Item 15: Design and document for inheritance or else prohibit it

- Which methods should be exposed as protected?
  - each protected method represents a commitment to an implementation detail.
  - but, a missing protected method can render a class unusable for inheritance.
  - no magic bullet !
  - (Also mentioned in Item 12 - minimize accessibility)

114/66

### Item 15: Design and document for inheritance or else prohibit it

- Constructors must not invoke overridable methods, directly or indirectly
    - The superclass constructor runs before the subclass constructor
    - the overriding method will get invoked before the subclass constructor has run.
    - If the overriding method depends on any initialization performed by the subclass constructor, then the method will not behave as expected.
- ```
public class Base {  
    public Base() {m();}  
    public void m() {}  
}  
final class Derived extends Base {  
    private final Date date; // Blank final, set by constructor  
    Derived() {date = new Date();}  
    public void m() {System.out.println(date);}  
}  
■ un client ...  
public static void main(String[] args) {Derived s = new Derived(); s.m();}  
  
■ the method m is invoked by the constructor Base() before the constructor Derived() has a chance to initialize the date field.
```

115/66

Item 15: Design and document for inheritance or else prohibit it

- designing a class for inheritance places substantial limitations on the class.
- A simple cure
 - move the body of each overridable method to a private “helper method” and have each overridable method invoke its private helper method.
- prohibit subclassing in classes not designed and documented for safe subclassing.
 - declare the class final.
 - or, make all constructors private or package-private and add public *static* factories

116/66

- a class declared with the `final` modifier cannot be subclassed.
 - prevents extensions to classes that are not robust enough for this
 - enables compiler optimization
- `final` methods
 - a method declared `final` cannot be overridden.
 - static methods and private methods cannot be overridden either, nor can the methods of a `final` class.
 - If a method cannot be overridden, the compiler may perform certain optimizations

TBD: add motivation on the evil of extension

On the open/closed principle and the fragile base class problem

TBD: this is here out not much attached to its motivation. Should it rather appear when we discuss the fragile base class problem?

- Abstract classes and interfaces
- Default implementations
- Generics, and their substitutability

- to make it easy to work with an array of shapes, it would be helpful if all our shape classes have a common superclass, Shape.

TBD: utile dichiarare oggetti in una classe estesa e tenerli con riferimenti alla classe base, per astrazione e sostituitibilità; ma così facendo non posso invocare metodi specifici delle classi base; e.g. getArea è specifico al caso di un Rectangle o un Circle, ma lo voglio invocare anche su uno Shape.

TBD: circa lavorare per specializzazione :(o per generalizzazione :). Se lavoro per generalizzazione, prima programmo Rectangle, poi Circle e poi creo la superclasse generalizzata Shape; è molto XP. Nell'approccio opposto, prima dichiaro Shape e poi costruisco Rectangle e Circle; è molto upfront
- Abstract Methods
 - An abstract method has no implementation, it is just declared;
 - Any class with an abstract method is automatically abstract itself, and must be declared as such.
 - A class may be declared abstract even if it has no abstract methods.
 - If a subclass of an abstract class does not implement all of the abstract methods it inherits, that subclass is itself abstract.
 - An abstract class cannot be instantiated.
 - A subclass of an abstract class can be instantiated if it implements each of the abstract methods of its superclass

```
public abstract class Shape {  
    public abstract double area();  
    public abstract double circumference();  
}  
  
class Circle extends Shape {  
    protected double r;  
    protected static final double PI = 3.14159265358979323846;  
    public Circle() { r = 1.0; }  
    public Circle(double r) { this.r = r; }  
    public double area() { return PI * r * r; }  
    public double circumference() { return 2 * PI * r; }  
    public double getRadius() { return r; }  
}  
  
class Rectangle extends Shape {  
    protected double w, h;  
    public Rectangle() { w = 0.0; h = 0.0; }  
    public Rectangle(double w, double h) { this.w = w; this.h = h; }  
    public double area() { return w * h; }  
    public double circumference() { return 2 * (w + h); }  
    public double getWidth() { return w; }  
    public double getHeight() { return h; }  
}
```

```
public abstract class Shape {  
    public abstract double area();  
    ... }  
class Circle extends Shape {  
    public double area() { return PI * r * r; }  
    ... }  
class Rectangle extends Shape {  
    public double area() { return w * h; }  
    ... }  
  
Shape[] shapes = new Shape[3]; // Create an array to hold shapes.  
shapes[0] = new Circle(2.0); // Fill in the array...  
shapes[1] = new Rectangle(1.0, 3.0);  
shapes[2] = new Rectangle(4.0, 2.0);  
double total_area = 0;  
for(int i = 0; i < shapes.length; i++)  
    total_area += shapes[i].area(); // Compute the area of the shapes.
```

- subclasses of Shape can be assigned to Shape. No cast is necessary.
- You can invoke the area () and circumference () for Shape objects,

■ Interfaces

- An interface looks a lot like an abstract class, except that it uses the keyword interface instead of the words abstract and class.

```
public interface Drawable {  
    public void setColor(Color c); // a declaration without definition  
    public void setPosition(double x, double y);  
    public void draw(DrawWindow dw);  
}
```

- all the methods declared within an interface are implicitly abstract.

*TBD: Relaxed since the release of Java8:
interfaces can include default method implementations
(e.g. useful for the Composite pattern)*

- any variables declared in an interface must be static and final

- a DrawableRectangle class
that extends the Rectangle class
and implements the Drawable interface

- sometimes called "mix-in"

```
public class DrawableRectangle extends Rectangle implements Drawable {  
    // New instance variables  
    private Color c;  
    private double x, y;  
    // A constructor  
    public DrawableRectangle(double w, double h) { super(w, h); }  
    // Here are implementations of the Drawable methods.  
    // We also inherit all the public methods of Rectangle.  
    public void setColor(Color c) { this.c = c; }  
    public void setPosition(double x, double y) { this.x = x; this.y = y; }  
    public void draw(DrawWindow dw) {  
        dw.drawRect(x, y, w, h, c);  
    }  
}
```

TBD: Is this just a class Adapter?

TBD: more advanced example:

in a Composite, a Leaf inherits from Observable and also from Component

- Using Interfaces

- Assume DrawableCircle and DrawableSquare
are implemented as we implemented DrawableRectangle

```
Shape[] shapes = new Shape[3]; // Create an array to hold shapes  
Drawable[] drawables = new Drawable[3]; // and an array to hold drawables.  
// Create some drawable shapes.  
DrawableCircle dc = new DrawableCircle(1.1);  
DrawableSquare ds = new DrawableSquare(2.5);  
DrawableRectangle dr = new DrawableRectangle(2.3, 4.5);  
// The shapes can be assigned to both arrays.  
shapes[0] = dc; drawables[0] = dc;  
shapes[1] = ds; drawables[1] = ds;  
shapes[2] = dr; drawables[2] = dr;  
// Compute total area and draw the shapes by invoking  
// the Shape and the Drawable abstract methods.  
double total_area = 0;  
for(int i = 0; i < shapes.length; i++) {  
    total_area += shapes[i].area(); // Compute the area of the shapes.  
    drawables[i].setPosition(i*10.0, i*10.0);  
    drawables[i].draw(draw_window); // Assume draw_window defined somewhere.  
}
```

- when a class implements an interface, instances of that class can be assigned to reference variables of the interface type.
- Don't interpret this example to imply that you must assign a `DrawableRectangle` object to a `Drawable` variable before you can invoke the `draw()` method or that you must assign it to a `Shape` variable before you can invoke the `area()` method.
- `DrawableRectangle` defines `draw()` and inherits `area()` from its `Rectangle` superclass, and so you can always invoke these methods.

■ Implementing Multiple Interfaces

```
public class DrawableScalableRectangle extends DrawableRectangle
implements Drawable, Scalable {
// The methods of the Scalable interface must be implemented here.
}
```

■ Constants in Interfaces

- the class that implements the interface "inherits" the constants
- There is no need to prefix them with the name of the interface

```
class A { static final int CONSTANT1 = 3; }
interface B { static final int CONSTANT2 = 4; }
class C implements B {
    void f() {
        int i = A.CONSTANT1; // Have to use the class name here.
        int j = CONSTANT2; // No class name here, because we implement
                           // the interface that defines this constant.
    }
}
```

127/116

■ Extending Interfaces

- Interfaces can have sub-interfaces
- An interface can extend more than one interface at a time:

```
public interface Transformable extends Scalable, Rotateable, Reflectable { }
public interface DrawingObject extends Drawable, Transformable { }
public class Shape implements DrawingObject { ... }
```

128/116

■ Marker Interfaces

- define an interface that is entirely empty. A class can implement this interface to provide additional information about itself.
- The `Cloneable` interface in `java.lang` is an example of "marker interface." It defines no methods, but serves simply to identify the class as one that will allow its internal state to be cloned by the `clone()` method of the `Object` class.
- You can test whether a class implements a marker interface (or any interface) using the `instanceof` operator.

TBD: this is a matter of method, more than language. Should it come later?

TBD: add something on functional interfaces (Single Method Interfaces)

- Although `Object` is a concrete class, it is designed primarily for extension.
- All of its nonfinal methods (`equals`, `hashCode`, `toString`, `clone`, and `finalize`) have explicit general contracts because they are designed to be overridden.
 - It is the responsibility of any class overriding these methods to obey their contracts;

Item 16: Prefer interfaces to abstract classes

<Precondition: consolidate nested classes as a basic java construct>

<Precondition: favor composition over inheritance>

- abstract classes are permitted to contain implementations for some methods, but, to implement an abstract class, a class must inherit from it
 - and, Java permits only single inheritance
- a class implementing an interface can be anywhere in the class hierarchy
- Existing classes can be easily retrofitted to implement a new interface
 - Existing classes cannot, in general, be retrofitted to extend a new abstract class
 - If two classes should implement the same abstract class, this should be placed high up in the type hierarchy so as to be a common ancestor
 - ... and all intermediate descendants should implement the new abstract class as well
- Interfaces are ideal for defining mixins.
 - A *mixin* is a type that a class can implement in addition to its “primary type”
 - E.g. interface Shape features methods about area and perimeter, interface Drawable features methods about setting colors and drawing, each can be associated with an abstract class, used by concrete classes to fulfill the interface contract
 - e.g. Comparable is a mixin interface that allows a class to declare that its instances are ordered with respect to other mutually comparable objects.
 - Abstract classes cannot be used to define mixins due to single inheritance

131/66

Item 16: Prefer interfaces to abstract classes

- Interfaces allow the construction of nonhierarchical type frameworks
 - many concepts don't fall neatly into a rigid hierarchy.

```
public interface Singer {AudioClip Sing(Song s);}
public interface Songwriter {Song compose(boolean hit);}
// some Singer is also a SongWriter
public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```
- Interfaces enable the wrapper class idiom (Item 14) to enhance functionality
 - using abstract classes to define types, the programmer cannot choose composition over inheritance
 - (see the code snippet on fragileBaseClass: use an interface to make the wrapper class substitutable to the wrapped one)

132/66

Item 16: Prefer interfaces to abstract classes

- providing an abstract skeletal implementation class for exported interfaces
 - overcome the downside of interfaces, which cannot provide a partial implementation
 - by convention, call **AbstractInterface** the skeletal implementation of Interface (e.g. `AbstractCollection`, `AbstractSet`, `AbstractList` ...)
 - *TBD: the best example comes at the time of the Composite pattern*

```
// note that AbstractList is an abstract class and implements the List interface
static List intArrayList(final int[] a) { // a static factory returning a List
    if (a == null) throw new NullPointerException();
    // create an AbstractList as instance of an anonymous local class
    // since the enclosing method is static, the local class is also static
    return new AbstractList() {
        public Object get(int i) {return new Integer(a[i]);}
        public int size() {return a.length;}
        public Object set(int i, Object o) {
            int oldVal = a[i];
            a[i] = ((Integer)o).intValue();
            return new Integer(oldVal);
        }
    };
}
```

- Here is where partial implementations in interfaces of Java8 can become relevant

Item 16: Prefer interfaces to abstract classes

- Technique of *simulated multiple inheritance*
 - the class implementing the interface forwards invocations of interface methods to a contained instance of a private inner class that extends the skeletal implementation.
 - closely related to the wrapper class idiom (Item 14)
 - provides most of the benefits of multiple inheritance, while avoiding the pitfalls.

Item 16: Prefer interfaces to abstract classes

- Approach to the definition of a skeletal implementation

- Identify primitive methods of the interface in terms of which the others can be implemented.
 - primitives will be left abstract, and the others will have a concrete implementation
- ```
public abstract class AbstractMapEntry implements Map.Entry {
 public abstract Object getKey(); // primitive method, left abstract
 public abstract Object getValue(); // primitive method, left abstract
 // concrete implementation follow for methods that can be derived from primitive ones
 public Object setValue(Object value) {throw new UnsupportedOperationException();}
 public boolean equals(Object o) {
 if (o == this) return true;
 if (! (o instanceof Map.Entry)) return false;
 Map.Entry arg = (Map.Entry)o;
 return eq(getKey(), arg.getKey()) && eq(getValue(), arg.getValue());}
 private static boolean eq(Object o1, Object o2) {return (o1==null ? o2 == null : o1.equals(o2));}
 public int hashCode() { return (getKey() == null ? 0 : getKey().hashCode()) ^
 (getValue() == null ? 0 : getValue().hashCode());}
}
```

- Recall that skeletal implementations must be designed and documented for inheritance (item 15)

135/66

## Item 16: Prefer interfaces to abstract classes

- Another limit of interfaces over abstract classes

- if a new method is added in an interface, all implementations must be extended
    - If it is added to an abstract class, some default implementation can be provided
    - Thus, abstract classes are more *evolvable* than interfaces
    - interfaces must be designed with care so as not to need to be re-opened
- this limit is overcome by default methods implementation, since Java8

136/66

- With Java 8, interfaces can include
  - default methods
  - static methods

- Java 8 adds interface default methods
  - Also known as Defender Methods or Virtual extension methods
  - declared using the modifier "default"

```
public interface MyInterface {
 void aMethodAsUsual (...);
 default void aMethodWithDefault (...) {...}
}
```
- A class implementing the interface shall give implementation to methods without default

```
public class MyClass implements MyInterface{
 public void aMethodAsUsual(...) {...} // method without default
}
```
- (positive) consequences
  - An interface can be extended without breaking implementation classes (more powerful than accompanying interfaces with base implementation)
  - mitigate the limitation of single inheritance

- a drawback: multiple defaults
  - a diamond problem may occur, as with multiple inheritance
  - two interfaces can implement the same method,
  - and a class can implement both the interfaces
- methods with multiple inherited defaults shall be implemented
  - possibly by upcall to the interface default

```
public interface MyInterface {
 void aMethodAsUsual(...);
 default void aMethodWithDefault (...) {...}
}
public interface AnotherInterface {
 void anotherMethodAsUsual(...);
 default void aMethodWithDefault (...) {...}
}
public class MyClass implements MyInterface, AnotherInterface{
 public void aMethodAsUsual(...) {...} // without default
 public void anotherMethodAsUsual(...) {...}
 public void aMethodWithDefault(...) { // multiple defaults
 MyInterface.aMethodWithDefault(...);
 }
}
```

139/66

```
public interface MyInterface {
 void aMethodAsUsual(...);
 default void aMethodWithDefault (...) {...}
}

public interface AnotherInterface {
 void anotherMethodAsUsual(...);
 default void aMethodWithDefault (...) {...}
}

public class MyClass implements MyInterface,
AnotherInterface{
 public void aMethodAsUsual(...) {...} // without default
 public void anotherMethodAsUsual(...) {...}
 public void aMethodWithDefault(...) { // multiple
 defaults
 MyInterface.aMethodWithDefault(...);
 }
}
```

140/66

- Java 8 adds interface static methods
  - similar to interface default methods
  - but, cannot be overridden
- an interface static method is visible to interface methods
  - ... or it can be invoked using Interface name, as for usual static methods

```
public interface MyInterface {
 void aMethodAsUsual(...);
 default void aMethodWithDefault(...) {...aStaticMethod(...) ...}
 static boolean aStaticMethod(...) {...}
}

... MyInterface.aStaticMethod(...) ...
```

- Remarks
  - Java interface static method is part of interface, we can't use it for implementation class objects.
  - Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
  - Java interface static method helps us in providing security by not allowing implementation classes to override them.
  - We can't define interface static method for Object class methods, we will get compiler error as "This static method cannot hide the instance method from Object". This is because it's not allowed in java, since Object is the base class for all the classes and we can't have one class level static method and another instance method with same signature.
  - We can use java interface static methods to remove utility classes such as Collections and move all of its static methods to the corresponding interface, that would be easy to find and use.

- called **default methods** or also **defender methods**
  - added since Java 8 (2014)
- intended for backward compatibility in the extension of interfaces

```
public interface oldInterface {
 public void existingMethod();
 default public void newDefaultMethod() {
 System.out.println("New default method"
 " is added in interface");
 }
}

public class oldInterfaceImpl implements oldInterface {
 public void existingMethod() {
 // existing implementation is here...
 }
 // no implementation needed for newDefaultMethod()
}
```

143/116

- A notable case:
  - Java 8 adds `forEach()` method in the collection library
  - and provides it with a default implementation  
(which also uses lambda expressions, also added since Java 8)

```
public interface Iterable<T> {
 public default void forEach(Consumer<? super T> consumer)
 {
 for (T t : this) {consumer.accept(t);}
 }
}
```

- of course, default methods cannot use instance variables
  - ... but, they can access instance variables passed by arguments

144/116

- multiple interface may lead to ambiguity

- a class implementing 2 interfaces with default implementations for the same method name will produce a compile error (`«java: class Impl inherits unrelated defaults for defaultMethod() from types InterfaceA and InterfaceB»`)

```
public interface InterfaceA {
 default void defaultMethod() {
 System.out.println("Interface A default method");
 }
}

public interface InterfaceB {
 default void defaultMethod() {
 System.out.println("Interface B default method");
 }
}

public class Impl implements InterfaceA, InterfaceB { }
```

145/116

- ... to solve multiple interface ambiguity

- either re-implement the method

```
public class Impl implements InterfaceA, InterfaceB {
 public void defaultMethod() {
 }
}
```

- ... or explicitly refer the implementation to bind

```
public class Impl implements InterfaceA, InterfaceB {
 public void defaultMethod(){
 // existing code here..
 InterfaceA.super.defaultMethod();
 }
}
```

146/116

## Java Basics & Idioms – Part III

- Visibility
- Packages, class and field visibility modifiers
- Defensive copies
- Immutable objects
- ... other idioms
- Singleton

- *TBD: this is here to precede the introduction of visibility levels*

- In java, classes are collected in packages

- a package is a set of classes, and each class belongs to 1 package
- provides a mechanism of name scoping and supports visibility control
- The classes defined in a file are associated to a package using the package statement

```
package shapes;
// Specifies a package for the class(es) in this file
...
```

- the name of a package is (by discipline) lower case
- If the package statement appears, it must be the first in the file
- If package statement is omitted, classes are be associated to the unnamed default package

- Some packages of the Java API

- java.io: Classes for all kinds of input and output
- java.lang: Classes for the core language
- java.util: Classes for useful data types
- ...
- java.applet: Classes for implementing applets
- java.awt: Classes for graphics, text, windows, and GUIs
- java.awt.image: Classes for image processing
- java.awt.peer: Interfaces for a platform-independent GUI toolkit
- java.net: Classes for networking

- Cmp to standard libraries of c

- Classes vs libraries
- larger grain in the functional perspective
- Expands the (open)standard base

- <http://docs.oracle.com/javase/8/docs/api/>

Enrico Vicario  
SWE – AA22/23

<http://docs.oracle.com/javase/7/docs/api/>

| Modifier and Type | Field and Description                                                                 |
|-------------------|---------------------------------------------------------------------------------------|
| static Boolean    | FALSE<br>The Boolean object corresponding to the primitive value <code>false</code> . |
| static Boolean    | TRUE<br>The Boolean object corresponding to the primitive value <code>true</code> .   |

151/116

Enrico Vicario  
SWE – AA22/23

## Packages

- **java.lang: classes fundamental to the language**
  - *Object*: the root of the class hierarchy,
  - *Class*: instances of which represent classes at run time
  - *Boolean, Character, Integer, Long, Float, Double, Void*:
    - classes wrapping a value of a primitive type into an object,
    - allowing storage as a reference type
    - conversions and standard operations (e.g. `hashCode()`, `equals()`)
    - *Void* class is non-instantiable,  
holds a reference to a *Class* object representing the type *void*.
  - *Math*: provides commonly used mathematical functions (e.g sine, cosine, sqrt)
  - *String, StringBuffer, and StringBuilder*: common operations on Strings.
  - *ClassLoader, Process, ProcessBuilder, Runtime, SecurityManager, and System*: "system operations" managing dynamic loading of classes, creation of external processes, host environment inquiries (e.g. time of day), enforcement of security policies.
  - *Throwable*: objects that may be thrown by the `throw` statement;  
*Error* and *Exception* are subclasses of *Throwable*.
- **java.io: input output**
  - system input and output through data streams,
  - serialization and file system

152/116

| Enrico Vicario<br>SWE – AA22/23 | Packages                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                 | <ul style="list-style-type: none"> <li>■ <b>java.util:</b> the collections framework and legacy collections, event model, date and time facilities, internationalization, miscellaneous utility classes (string tokenizer, random-numbers, bit array).</li> <li>■ <b>Collections framework:</b> <ul style="list-style-type: none"> <li>▪ <i>Collection interfaces:</i> interfaces of different types of collections, such as sets, lists, and maps.</li> <li>▪ <i>General-purpose implementations:</i> primary implementations of interfaces.</li> <li>▪ <i>Legacy implementations:</i> collection classes from earlier releases (Vector and Hashtable) retrofitted to implement the collection interfaces</li> <li>▪ <i>Special-purpose implementations:</i> Implementations designed for use in special situations (nonstandard performance, usage restrictions, behavior)</li> <li>▪ <i>Concurrent implementations:</i> Implementations for highly concurrent use</li> <li>▪ <i>Wrapper implementations:</i> Add functionality, such as synchronization</li> <li>▪ <i>Convenience implementations:</i> High-performance "mini-implementations"</li> <li>▪ <i>Abstract implementations:</i> Partial implementations of the collection interfaces to facilitate custom implementations.</li> <li>▪ <i>Algorithms:</i> Static methods implementing useful operations (e.g. sorting)</li> <li>▪ <i>Infrastructure:</i> Interfaces providing essential support for collection interfaces.</li> <li>▪ <i>Array Utilities:</i> Utility functions for arrays</li> </ul> </li> </ul> |

| Enrico Vicario<br>SWE – AA22/23 | Visibility                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                 | <ul style="list-style-type: none"> <li>■ <b>Information hiding</b> <ul style="list-style-type: none"> <li>▪ is about exposing an abstraction that hides details of the implementation</li> <li>▪ a general concept of software engineering, not only for programming or java</li> <li>▪ protects the implementation from ignorant/malicious access</li> <li>▪ Relieves the client from knowing structural details</li> <li>▪ Relieves the developer from documenting internal assumptions</li> <li>▪ Provides a functional and de-cluttered perspective</li> <li>▪ Enables evolution of the implementation be decoupled from abstraction</li> <li>▪ Hiding is about visibility</li> </ul> </li> <li>■ <b>3 Scopes of visibility in Java</b> <ul style="list-style-type: none"> <li>▪ the body of a method, a class, a package</li> <li>▪ A package is a set of classes, and each class belongs to 1 package</li> </ul> </li> <li>■ <b>Access (visibility) modifiers (of the declaration)</b> <ul style="list-style-type: none"> <li>▪ A class can be <code>public package</code></li> <li>▪ A class member can be <code>public protected package private</code></li> </ul> </li> </ul> |

■ visibility modifier of the class declaration/definition

```
package shapes;
public class Circle { ... }
```

- can be public or package (is package if omitted)
- determines the visibility scope of the class

■ A public class is visible everywhere

- i.e. any other class can refer to the class name
- does not necessarily imply that any class can create an instance
  - E.g. the constructor might be private or protected (singleton)
  - about the difference between managing references and creating objects

■ A package class is visible only from within its package

■ What would be a private or protected class for?

- Something like that can be obtained through inner classes

■ visibility modifier of variables and methods of a class

```
package shapes;
public class Circle {
 ...
 protected double r; // hidden, but visible to subclasses.
 public Circle(double x, double y, double r) {...}
 int aPackageVisibleMethod(...) {...}
}
```

- each variable and method can be public| protected| package| private
- when not specified the visibility is package

## Class members visibility modifiers

- **Private**
  - A **private** member is visible only in methods defined within its class (or in classes defined within the class)
  - Private members are not visible within subclasses, and are *not* inherited
  - (non-private) methods that invoke **private** methods internally are inherited and may be invoked by subclasses)
  - Every object has its copy of all fields of all superclasses, including **private** fields
- **Package**
  - it is visible only within the classes that are part of the same package.
  - package visibility is more restrictive than **protected**, but less restrictive than **private**.
  - A subclass inherits the **protected** members of its superclass, but it can only access those members through instances of itself, not directly in instances of the superclass. FixMe: vedi la nota
- **Protected**
  - A **protected** member is visible within the class where it is defined, within all subclasses of the class, and within all classes in the same package
- **Public**
  - Public members are visible wherever their class is

157/116

## Class members visibility modifiers

| Accessible to:                  | public | protected | package | private |
|---------------------------------|--------|-----------|---------|---------|
| Same class                      | yes    | yes       | yes     | yes     |
| Class in same package           | yes    | yes       | yes     | no      |
| Subclass in different package   | yes    | yes       | no      | no      |
| Non-subclass, different package | yes    | no        | no      | no      |

- **public**: for methods and constants that form part of the public API of the class
- **protected**: for fields and methods that aren't necessary to use the class, but that may be of interest to anyone creating a subclass as part of a different package
- **package**: for fields and methods to be hidden outside of the package, but which you want cooperating classes within the same package to have access to.
- classes in the same package are assumed to know about, and trust, each other.
- you can't take advantage of package visibility unless you use the **package** statement to group your related classes into packages.
- **private**: for fields and methods used inside the class to be hidden everywhere else.

158/116

■ Encapsulation

- Is about coupling together data and functions
- here intended as: hide data and provide methods to access them
- Can be a way to do information hiding

```
public class Circle {
 protected double x, y; // hidden, but visible to subclasses.
 protected double r; // hidden, but visible to subclasses.
 private static final double MAXR = 100.0; // (constant).
 ...
 // Public data access methods
 public void moveto(double x, double y) { this.x=x; this.y = y; }
 public void move(double dx, double dy) { x += dx; y += dy; }
 public void setRadius(double r){this.r=(check_rad (r))?r:MAXR;}
 // Declare trivial methods final to avoid dynamic method lookup
 public final double getX() { return x; }
 public final double getY() { return y; }
 public final double getRadius() { return r; };
}
```

```
package shapes; // Specify a package for the class.
public class Circle { // Note that the class is still public!
 protected double x, y; // hidden, but visible to subclasses.
 protected double r; // hidden, but visible to subclasses.
 private static final double MAXR = 100.0; // (constant).
 private boolean check_rad(double r) { return (r <= MAXR); }
 // Public constructors
 public Circle(double x, double y, double r) {this.x=x; this.y=y;
 if (check_rad (r)) this.r = r; else this.r = MAXR; }
 public Circle(double r) { this(0.0, 0.0, r); }
 public Circle() { this(0.0, 0.0, 1.0); }
 // Public data access methods
 public void moveto(double x, double y) { this.x=x; this.y = y; }
 public void move(double dx, double dy) { x += dx; y += dy; }
 public void setRadius(double r){this.r=(check_rad (r))?r:MAXR;}
 // Declare trivial methods final to avoid dynamic method lookup
 public final double getX() { return x; }
 public final double getY() { return y; }
 public final double getRadius() { return r; };
}
```

## Item #12: Minimize the accessibility of classes and members

- Accessibility modifiers control visibility and access of classes, interfaces, and members
  - Classes and interfaces can be `public` | `(package-private)`
  - Members can be `public` | `protected` | `(package-private)` | `private`
- Rule of thumb: make each class and member as inaccessible as possible
- Why? information hiding and encapsulation produce decoupling
  - Eases parallel development, and maintenance, and performance tuning
  - Increases SW reuse
  - Mitigates the risk when building large systems:  
individual modules may prove successful,  
even if the system for which they were built does not
  - Avoids the burden of documentation
  - ... could say more

161/66

## Item 12: Minimize the accessibility of classes and members

- For members of public classes, a huge increase in accessibility occurs when the access level goes from `package-private` to `protected`.
  - A protected member is part of the class's exported API,  
it must be supported forever,  
and represents a public commitment to an implementation detail
  - There are not many cases where protected members are in fact essential  
*TBD: A good example is `notify()` in the Observer pattern;  
Observable is in a standard package, while concreteSubject is a class made  
for its own business*
- In the override of a method, accessibility cannot be lowered
  - Is a corollary of the substitution principle:  
a member of a subclass must fit any request for any member of the superclass
  - As a further corollary:  
when a class implements an interface, all the methods of the interface must be `public`
- Instance fields should never be `public`
  - If an instance field is `nonfinal`, or is a `final` reference to a mutable object,  
then by making the field `public`, you give up the ability to limit its values,
  - ... and to enforce invariants
  - classes with `public` mutable fields are not thread-safe

162/66

## Short aside on class Observable and interface Observer - 1/2

- *TBD: questo serve per potere discutere della visibilità di attach, detach, notify... e anche per discutere di librerie standard  
ma conviene discuterne dopo, più avanti con i patterns. E col codice*

### Observer class

- A class can implement the Observer interface when it wants to be informed of changes in observable objects

```
package java.util;
public interface Observer
{ void update(Observable o, Object arg);
 // is called whenever the observed object is changed.
 // Observable o is for pull-mode, Object arg for push-mode}
```

### Observable class

- represents an observable object, or "data" in the model-view paradigm.
- It can be subclassed to represent an object that has to be observed.
- An observable object can have one or more observers.  
An observer may be any object that implements interface **Observer**.
- After an observable instance changes, a call to the notifyObservers method causes all its observers to be notified of the change by a call to their **update** method.  
The order in which notifications will be delivered is unspecified. The default implementation provided in the Observable class will notify Observers in the order in which they registered interest, but subclasses may change this order.

```
package java.util;
public class Observable extends Object
```

163/66

## Short aside on class Observable and interface Observer - 2/2

```
package java.util;
public class Observable extends Object

public void addObserver(Observer o)
// adds an observer to the set (not multset !) of observers
public void deleteObserver(Observer o)
// Deletes an observer from the set of observers

public void notifyObservers(Object arg)
// If hasChanged()==true, notifies all the observers by
// invoking
// their update() with two arguments: this observable object
// and
// the arg argument. Arg supports the push mode.
public void notifyObservers()
// operates as notifyObservers(null), thus supports only pull-
// mode

protected void setChanged()
// Marks this Observable so that hasChanged() will return true
protected void clearChanged()
// Marks this object so that hasChanged() will return false
// This method is called automatically by notifyObservers
public boolean hasChanged()
// Tests if this object has changed.
```

164/66

## Item #24: make defensive copies when needed

- In c you can pass data structures by value
  - reduces efficiency and is never used
  - but can serve to ensure confinement of side effects
- in java you pass by value
  - but you have only references to pass
  - and since we desire what we don't have, we want a way to emulate pass-by-value of objects
  - call this defensive copy
- example of a wrapper that composes a wrapped object
  - assumes that no-one else will ever produce any side effect on it
  - yet, receives the object from an external factory
  - and delegates to some outer object a processing on the values of the wrapped object.
- requires that defensive copies are made
  - at creation make a copy of the passed object (don't accept candies from unknown people);
  - for processing, pass a reference to a copy of the wrapped object.
    - *TBD: refer to code on the web page.*

*TBD: add UML or code*

165/66

## Item 24: make defensive copies when needed

- An apparently immutable class, enforcing start<=end
- ```
public final class Period {  
    private final Date start;  
    private final Date end;  
    /**  
     * @throws IllegalArgumentException if start is after end  
     */  
    public Period(Date start, Date end) {  
        if (start.compareTo(end) > 0)  
            throw new IllegalArgumentException(start + " after " + end);  
        this.start = start;  
        this.end = end;  
    }  
    public Date start() {return start;}  
    public Date end() {return end;}  
    ...  
}
```
- The invariant can be broken by exploiting the fact that Date is mutable:
- ```
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78); // Modifies internals of p!
```

166/66

## Item 24: make defensive copies when needed

- To protect the internals of a Period instance from this sort of attack,
  - it is essential to make a defensive copy of each mutable parameter
  - and to use the copies as components of the Period instance in place of the originals:

```
public Period(Date start, Date end) {
 this.start = new Date(start.getTime());
 this.end = new Date(end.getTime());
 if (this.start.compareTo(this.end) > 0)
 throw new IllegalArgumentException(start + " after " + end);
}
```
- defensive copies are made before checking the validity of the parameters
  - validity check is performed on the copies rather than on the originals.
  - protects the class against changes to the parameters from another thread
- we did not use Date's clone method to make the defensive copies.
  - Since Date is nonfinal,  
the clone method could return an instance of an untrusted subclass

167/66

## Item 24: make defensive copies when needed

- ... Another vulnerability
  - End() returns the reference to the internal copy

```
public final class Period {
 private final Date start;
 private final Date end;

 ...
 public Date start() {return start;}
 public Date end() {return end;}
 ...
}
```
  - And this uses the vulnerability to break the invariant

```
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78); // Modifies internals of p!
```
  - Overcome the limit, by returning references to copies

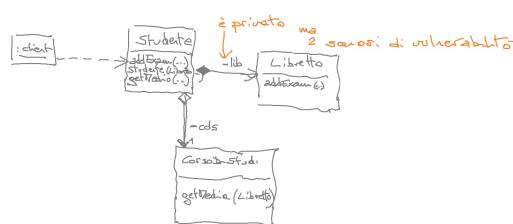
```
public Date start() { return new Date(start.getTime());}
public Date end() {return new Date(end.getTime());}
```

168/66

### Item #13: favor immutability

- An immutable class is a class whose instances cannot be modified.
  - The state of each instance is set on creation and is fixed for the object lifetime
  - E.g. String, boxed primitive classes Boolean, BigInteger, BigDecimal, ...
- Immutable objects are simple, thread-safe, and can be shared
- An immutable class can provide static factories
  - frequently requested instances can be cached, to avoid creating a new one on each call
  - (see for instance the example of the Calendar in Item 4)

169/66



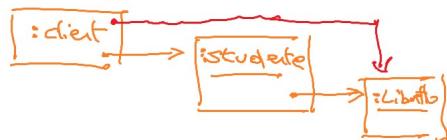
① // nel client

una comune container  
cosa è Studente e Libretto

Libretto lib = new Libretto(..);

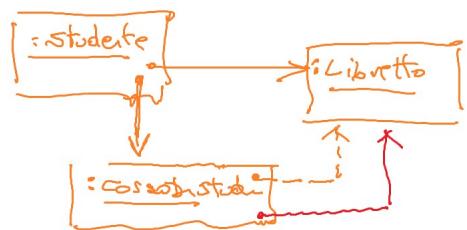
Studente st = new Studente(lib);

...  
lib.addExam(..)



170/203

② // in Studente per estrarre un  
// riferimento al Libretto  
public float getMedia ()  
{  
 return cds.getMedia(lib);  
}  
poi in qualche metodo in corso di studi  
vere usato il riferimento lib  
per un nuovo additivo



La soluzione è:

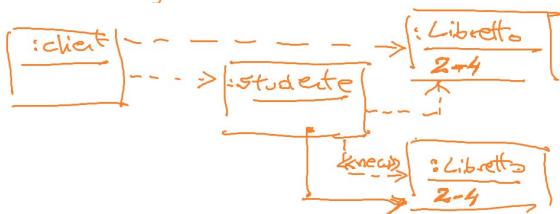
in ①: il costruttore di Studente  
mette il riferimento  
a uno oggetto del Libretto ricevuto

```
class Studente
{ private Libretto lib;
```

```
 Studente (Libretto lib, ...)
```

```
 { // this.lib = lib;
 this.lib = new Libretto (lib);
 }
```

```
 ...
```



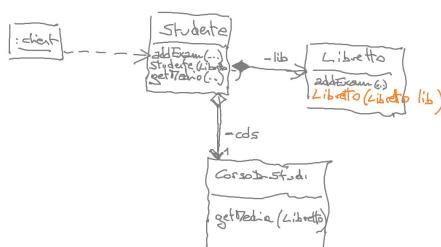
in ② : studente passa el: corso di studi  
uno copy del suo libretto

// in studente

public float getMedia() {

// return cds.getMedia(lib);

return cds.getMedia(new Libretto(lib));



173/203

### Item 13: favor immutability

- Five conditions to make a class immutable
  - don't provide any methods that modify the object's state (known as mutators).
  - ensure that the class can't be extended:  
prevents subclasses from behaving as if the object's state had changed
  - make all fields final
  - make all fields private
  - ensure exclusive access to any mutable components:  
if the class has any reference to mutable objects, ensure that clients cannot get them;  
make defensive copies in constructors, accessors, and readObject methods
- can be relaxed by allowing mutable fields  
that do not affect external behavior
  - E.g. for caching
- An alternative: make constructors private and add static factory methods
 

```
public class Complex {
 private final double re;
 private final double im;
 private Complex(double re, double im) {this.re = re; this.im = im;}
 public static Complex valueOf(double re, double im) { return new Complex(re, im);}
 ...
}
```

174/66

## Item #4: Avoid creating duplicate objects

- an item of effective Java on controlling instances at creation
  - an educated exercise (and practice) on:  
static construction, final classes, library types, ...
- it is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed.
  - An example was the static constructor Boolean.valueOf() in item 1
  - In general, this applies to any *immutable object* (see more in item 15)

*NB: se due oggetti sono immutabili e a un certo punto sono uguali,  
da li' in poi e' inutile distinguergli.*

*TBD: Il caso di Boolean conviene semmai introdurlo qui*

175/66

## Item 4: Avoid creating duplicate objects

- Note: this originally appears as  
*Item 1: Consider static factory methods instead of constructors*
- An example on controlling instances at creation
  - quite involved, yet useful to learn various other things
  - mixes two concepts: using a static factory method, and controlling instances
- ... requires some background on Boolean
  - java.lang.Boolean is a class  
that embeds the boolean type into an immutable object
  - valueOf is a method of Boolean, which receives a boolean value b,  
and returns a reference to a Boolean with the same value as b

```
static Boolean valueOf(boolean b)
```
  - Boolean.TRUE and Boolean.FALSE are static references  
to two immutable Boolean objects, created when the class Boolean is loaded

```
public static final Boolean TRUE
public static final Boolean FALSE
```
  - Learn more on Boolean at <http://docs.oracle.com/javase/8/docs/api/>

176/66

## A brief aside on documentation on standard Classes (1/2)

- <http://docs.oracle.com/javase/7/docs/api/>
- Boolean is a (final) class in the package java.lang

```
public final class Boolean
extends Object
implements Serializable, Comparable<Boolean>
```

The Boolean class wraps a value of the primitive type boolean in an object. An object of type Boolean contains a single field whose type is boolean.  
In addition, this class provides many methods for converting a boolean to a String and a String to a boolean, as well as other constants and methods useful when dealing with a boolean.

- Boolean features 2 static members TRUE and FALSE

- references to Boolean objects embedding true and false values
- TRUE and FALSE are final

### Fields

| Modifier and Type     | Field and Description                                                   |
|-----------------------|-------------------------------------------------------------------------|
| static Boolean        | FALSE<br>The Boolean object corresponding to the primitive value false. |
| static Boolean        | TRUE<br>The Boolean object corresponding to the primitive value true.   |
| static Class<Boolean> | TYPE<br>The Class object representing the primitive type boolean.       |

### Class Boolean

java.lang.Object  
java.lang.Boolean

All Implemented Interfaces:  
Serializable, Comparable<Boolean>

### TRUE

public static final Boolean TRUE  
The Boolean object corresponding to the primitive value true.

### FALSE

public static final Boolean FALSE  
The Boolean object corresponding to the primitive value false.

... and a static method valueOf

### valueOf

public static Boolean valueOf(boolean b)

Returns a Boolean instance representing the specified boolean value. If the specified boolean value is true, this method returns Boolean.TRUE; if it is false, this method returns Boolean.FALSE. If a new Boolean instance is not required, this method should generally be used in preference to the constructor Boolean(boolean), as this method is likely to yield significantly better space and time performance.

177/66

## A brief aside on documentation on standard Classes (2/2)

- Other facts about Constructors

### Constructors

#### Constructor and Description

|                        |                                                                                                                                              |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Boolean(boolean value) | Allocates a Boolean object representing the value argument.                                                                                  |
| Boolean(String s)      | Allocates a Boolean object representing the value true if the string argument is not null and is equal, ignoring case, to the string "true". |

- ... and other Methods

### Methods

#### Modifier and Type

| Modifier and Type | Method and Description                                                                                                                                     |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean           | booleanValue()<br>Returns the value of this Boolean object as a boolean primitive.                                                                         |
| static int        | compare(boolean x, boolean y)<br>Compares two boolean values.                                                                                              |
| int               | compareTo(Boolean b)<br>Compares this Boolean instance with another.                                                                                       |
| boolean           | equals(Object obj)<br>Returns true if and only if the arguments is not null and is a Boolean object that represents the same boolean value as this object. |
| static boolean    | getBoolean(String name)<br>Returns true if and only if the system property named by the argument exists and is equal to the string "true".                 |
| int               | hashCode()<br>Returns a hash code for this Boolean object.                                                                                                 |
| static boolean    | parseBoolean(String s)<br>Parses the string argument as a boolean.                                                                                         |
| String            | toString()<br>Returns a String object representing this Boolean's value.                                                                                   |
| static String     | toString(boolean b)<br>Returns a String object representing the specified boolean.                                                                         |
| static Boolean    | valueOf(boolean b)<br>Returns a Boolean instance representing the specified boolean value.                                                                 |
| static Boolean    | valueOf(String s)<br>Returns a Boolean with a value represented by the specified string.                                                                   |

#### Methods inherited from class java.lang.Object

clone, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

178/66

#### Item 4: Avoid creating duplicate objects

- // in the definition of Boolean class

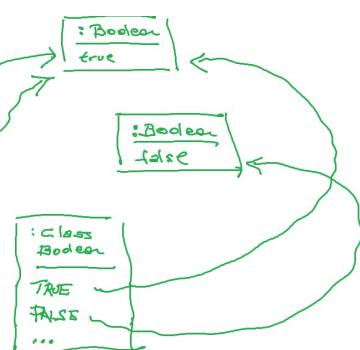
```
public class Boolean extends ... implements ...
{ ...
 public static Boolean valueOf(boolean b) // a static factory method in
 Boolean
 // returns a Boolean with boolean
 value
 }
 // in the client
 boolean b1, b2,b3; // three primitive boolean variables
 Boolean myB1, myB2, myB3; // three references to Boolean objects
 ... // do something that gives value to b1, b2, b3
 myB1 = Boolean.valueOf(b1);
 myB2 = Boolean.valueOf(b2);
 myB3 = Boolean.valueOf(b3);
 ...
```

- 3 references but only two objects

TBD: un disegno: ci sono  $b1$ ,  $b2$ , e i riferimenti  $myB1$  e  $myB2$  inizialmente nulli; e ci sono 2 oggetti Boolean, con valore true e false, creati dal classloader

179/66

```
boolean b;
 ...
 b = TRUE;
Boolean b1 = valueOf(b)
 b1
Boolean b2 = valueOf(b)
 b2
if(b1 == b2)
```



180/203

## Item 4: Avoid creating duplicate objects

### ■ The real story behind

- At the loading of the Boolean class, two objects of type Boolean are created, with embedded values true and false
- Whenever valueOf is invoked, no new objects are created
- A reference to one of the two Boolean objects is returned depending on the value of the formal parameter b (note that the implementation in principle could be not known)

### ■ TBD: draw variables in the programmer and memory spaces

- In the programmer's space: primitives b1,b2,b3 and references myB1,myB2,myB3
- In the managed memory space: only 2 immutable Boolean,
- Multiple references share the same object

*TBD: Since no two equal instances will ever exist, a.equals(b) iff a==b, which is more efficient to check than a.equals(b) method*

181/66

## Item 4: Avoid creating duplicate objects

*NOTE: skip this slide!*

*TBD: la nota sulle stringhe deve venire dopo, non aggiunge nulla se non dirci come nell'implementazione di Java e' stato applicato il concetto.*  
`String s = new String("stringette"); // DON'T DO THIS!`

- this creates a new String instance each time the statement is executed
- each instance will have its own identity (and memory footprint), but all the instances will always behave in the same manner as String is immutable
- Yet, all instances are distinct but equal objects, neither they can be muted
- A better statement using a single String instance:

`String s = "stringette"; // DO THIS!`

- Reduces the memory footprint, facilitates equivalence check, simplifies the picture

182/66

#### Item 4: Avoid creating duplicate objects

- ... reuse is not only for immutable objects, but also for objects that are not muted in their lifecycle
  - example: isBabyBoomer() checks if the birthdate falls in a given interval
  - a new Calendar, TimeZone, and two Date are created, at each invocation

*TBD: Si puo' esemplificare meglio. Qui in realta' l'antipattern e' mettere a livello di istanza una variabile (costante) che dovrebbe stare a livello di classe. L'esempio e' comunque ricco: come costruire un oggetto con un metodo statico e passarne il riferimento al costruttore statico di un altro oggetto, ....*

```
public class Person {
 private final Date birthDate;
 // ...
 public boolean isBabyBoomer() {
 // Unnecessary allocation of expensive object
 Calendar gmtCal =Calendar.getInstance(TimeZone.getTimeZone("GMT"));
 // note that this exemplifies twice the static factory method (item 1)
 // why call this getTimeZone and not getInstance ?
 gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
 Date boomStart = gmtCal.getTime();
 gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
 Date boomEnd = gmtCal.getTime();
 return birthDate.compareTo(boomStart) >= 0 &&
 birthDate.compareTo(boomEnd) < 0;}}}
```

Yet, start and end dates of the boom are always the same, and so is the TimeZone, ... and they could thus be reused across subsequent invocations

183/66

#### Item 4: Avoid creating duplicate objects

- ... a better solution: reuse a single instance across subsequent invocations
  - make reused components be shared at the class level
  - to reduce the memory footprint and the garbage collector workload, and to make the intent more evident

```
class Person {
 private final Date birthDate;
 //...
 private static final Date BOOM_START;
 private static final Date BOOM_END;
 static {
 // this is a static initializer (see java basics)
 Calendar gmtCal =Calendar.getInstance(TimeZone.getTimeZone("GMT"));
 gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
 BOOM_START = gmtCal.getTime();
 gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
 BOOM_END = gmtCal.getTime();
 }
 public boolean isBabyBoomer() { // now relying on static (and thus single) variables
 return birthDate.compareTo(BOOM_START) >= 0 &&
 birthDate.compareTo(BOOM_END) < 0;
 }
}
```

184/66

## Item 4: Avoid creating duplicate objects

- A last but not least example: the Adapter pattern
  - Sometimes an Adapter has no state
  - In this case there is no use in having multiple instances
    - *TBD: not so often; can be when the adaptee is itself a single instance object detto così è molto questionabile!*
- A remark of prudence:
  - there will be later an opposite and more relevant issue for avoiding reuse and rather creating "defensive copies" (item 24)
    - *TBD: o forse spostando slides ora lo abbiamo già visto*

185/66

## Item 2: Enforce the singleton property with a private constructor

- an educated exercise on private constructors and static methods or fields
- A singleton is a class that is instantiated exactly once [GOF]
  - Two implementation schemes, both keeping the constructor private and exporting a public static member to provide access to the sole instance
    - *TBD: meglio partire subito con l'implementazione fatta bene (ora nel seguito), in cui l'istanza e' creata da getInstance alla prima invocazione e poi si mantiene un riferimento privato (e statico perche' deve essere usato da getInstance).*
- Singleton with public final field

```
public class Elvis {
 public static final Elvis INSTANCE = new Elvis();
 private Elvis() { ... }
 public void leaveTheBuilding() { ... }
}
// client code
Elvis.INSTANCE.leaveTheBuilding();
```

  - the private constructor is called only once, at the initialization of Elvis.INSTANCE
  - The instance is created when the class is loaded
  - The lack of a public or protected constructor guarantees a "monoelvistic" universe
  - Elvis.INSTANCE becomes a kind of global variable

186/66

## Item 2: Enforce the singleton property with a private constructor

- Singleton with static factory

```
public class Elvis {
 private static final Elvis INSTANCE = new Elvis();
 private Elvis() { ... }
 public static Elvis getInstance() { return INSTANCE; }
 public void leaveTheBuilding() { ... }
}
// client code
Elvis.getInstance().leaveTheBuilding()
```

- All calls to Elvis.getInstance return the same object reference
- The instance is still created at the time of class loading

187/66

## Item 2: Enforce the singleton property with a private constructor

- ... some variants on the Singleton with static factory method

*TBD: questo è la pratica, e conviene presentarlo fino dall'inizio*

- Deferred creation

```
public class Elvis {
 private static Elvis INSTANCE = null; //TBD: non e' maiuscolo!
 private Elvis() { ... }
 public static Elvis getInstance() {
 if(INSTANCE==null) INSTANCE=new Elvis();
 return INSTANCE; }
 public void leaveTheBuilding() { ... }
}
```

- Deferred creation with ownership

```
public class Elvis {
 private static Elvis INSTANCE = null;
 private Object owner = null;
 private Elvis() { ... }
 public static Elvis getInstance(Object owner) {
 if(INSTANCE==null)
 {
 INSTANCE=new Elvis();
 this.owner=owner; }
 return INSTANCE; }
 public void leaveTheBuilding() { ... }
}
```

188/66

### Item 3: Enforce noninstantiability with a private constructor

TBD: non un major item.

Vale la pena di citarlo in riferimento a Math  
ma non merita troppo spazio.

- ... in previous items:
  - Item 1: using static factory methods (structural perspective)
    - to allow multiple constructors with significant names, also with the same signature,
    - and to allow control of instance creation (both in functional perspective)
  - Item 2: controlling creation so as to have a single instance (functional perspective)
    - possibly based on a static factory method (structural perspective)
- this item: controlling creation so as to have no instances (functional perspective)
  - question 1: What can this be for?
  - question 2: How can this be enforced? (structural perspective)
- Question 1: what can this be for?
  - a class can be made only of static methods and fields,  
and in this case there is no use in creating an instance
  - java.lang.Math collects constants and methods that don't refer to any instance context  
(<http://docs.oracle.com/javase/7/docs/api/>)

189/66

### Item 3: Enforce noninstantiability with a private constructor

- Question 2: how can non instantiability be enforced?
  - include a private constructor
- **// Noninstantiable utility class**

```
public class UtilityClass {
 // Suppress default constructor for noninstantiability
 private UtilityClass() { // this will never be invoked }
 ... // Remainder omitted
}
```
- The private constructor prevents subClassing
  - what if the constructor is declared protected?  
*Any class in the package will be allowed to create an instance :(*  
*"it can work" se nel package c'e' disciplina, ma questo puo costare molti tests*
- What if we enforce noninstantiability by making a class abstract?
  - the class could be subclassed and the subclass instantiated

TBD: What if we make it also final?

*Apparently, this might serve to obtain the same effect of a private constructor,  
preventing class instances and subclassing.*

*Could such a class be instantiated through an anonymous class in the constructor?  
(see Java Final abstract class su stackoverflow)*

190/66

### Item #23: check parameters for validity

- Most methods (and constructors) have restrictions on what values may be passed into their parameters.
  - non-negative index values, bounded indexes, non-null references, ...
- clearly document all such restrictions and enforce them with checks at the beginning of the method body
  - Unit tests can be a good means to document usage conditions
- for public methods, use the Javadoc `@throws` tag to document the exception
  - `IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointerException`

```
/*
 * Returns a BigInteger whose value is (this mod m). This method
 * differs from the remainder method in that it always returns a
 * non-negative BigInteger.
 * @param m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
 if (m.signum() <= 0)
 throw new ArithmeticException("Modulus <= 0: " + m);
 ... // Do the computation
}
```

191/66

- *TBD: Question: what is the advantage of a Singleton wrt a class made of static methods and variables*
  - un singleton puo' avere variabili e metodi privati?

192/203

### Item 23: check parameters for validity

- For public methods, check for validity is necessarily up to the callee
  - For package-private, it could also be a responsibility of the caller
- It is particularly important to check the validity of parameters that are stored away for later use
  - on dataflow testing on def-use paths
  - ... and also on Bohrbugs vs Mandelbugs
- Constructors represent a special case of the principle that you should check the validity of parameters that are to be stored away for later use.
  - It is critical to check the validity of constructor parameters to prevent the construction of an object that does not initially set the expected class invariants.

193/66

### Item #22: Replace function pointers with classes and interfaces

- C supports *function pointers*, typically used to allow the caller of a function to specialize its behavior by passing in a pointer to a second function, sometimes referred to as a callback
  - E.g. the operation to be repeated in the visit of a list, the comparator passed to `qsort`
  - A kind of strategy pattern
- Java attains the same functionality using the idiom of function objects
  - objects whose methods perform operations on other objects, passed explicitly

```
class StringLengthComparator {
 public int compare(String s1, String s2) {return s1.length() - s2.length();}
}
```
- StringLengthComparator class is stateless
  - Can be a singleton to save on unnecessary object creation costs (Items 4,2)

```
class StringLengthComparator {
 private StringLengthComparator() {} // private constructor
 public static final StringLengthComparator INSTANCE = // static final factory
 new StringLengthComparator();
 public int compare(String s1, String s2) {return s1.length() - s2.length();}
}
```

194/66

## Item 22: Replace function pointers with classes and interfaces

- This all can be cast into a full fledged strategy pattern
- Define an interface (the AbstractStrategy)  

```
public interface Comparator {public int compare(Object o1, Object o2); }
```
- Let StringLengthComparator be a possible implementation of Comparator (one out of many possible ConcreteStrategy)  

```
class StringLengthComparator implements Comparator{
 private StringLengthComparator() {} // private constructor
 public static final StringLengthComparator INSTANCE = // static final factory
 new StringLengthComparator();
 public int compare(String s1, String s2) {return s1.length() - s2.length();}
}
```

195/66

## Item 22: Replace function pointers with classes and interfaces

- <Precondition: consolidate nested classes>
- Concrete strategy classes are often declared using anonymous classes (Item 18)  

```
// a statement invokes Arrays.sort passing the reference to an array of String
// and the constructor of an implementation of Comparator
// that is defined as a nested local anonymous class
// that defines the implementation for the Comparator.compare method
Arrays.sort(stringArray, new Comparator() {
 // anonymous class local to the constructor invocation
 // sets up the concrete strategy for comparison
 public int compare(Object o1, Object o2) {
 String s1 = (String)o1;
 String s2 = (String)o2;
 return s1.length() - s2.length();
 } // end of the local anonymous class, and of the enclosing expression and
 // statement
```

196/66

- Generics
- Exception handling

- A *generic type* is a generic class or interface parameterized over types.
- a basic solution using Object
  - much like a void pointer in c
  - downcasting will be needed in the client code

```
public class Box {
 private Object object;
 public void set(Object object) { this.object = object; }
 public Object get() { return object; } }
```

- a solution with parametric type T

```
public class Box<T> {
 private T t;
 public void set(T t) { this.t = t; }
 public T get() { return t; } }
```

<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

## Generics – a notable example

java.util

### Class ArrayList<E>

```
java.lang.Object
 java.util.AbstractCollection<E>
 java.util.AbstractList<E>
 java.util.ArrayList<E>
```

#### All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

#### Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

199/116

## Generics

### Methods

| Modifier and Type | Method and Description                                                                                                                                                                                               |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean           | <code>add(E e)</code><br>Appends the specified element to the end of this list.                                                                                                                                      |
| void              | <code>add(int index, E element)</code><br>Inserts the specified element at the specified position in this list.                                                                                                      |
| boolean           | <code>addAll(Collection&lt;? extends E&gt; c)</code><br>Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| boolean           | <code>addAll(int index, Collection&lt;? extends E&gt; c)</code><br>Inserts all of the elements in the specified collection into this list, starting at the specified position.                                       |
| void              | <code>clear()</code><br>Removes all of the elements from this list.                                                                                                                                                  |
| Object            | <code>clone()</code><br>Returns a shallow copy of this ArrayList instance.                                                                                                                                           |
| boolean           | <code>contains(Object o)</code><br>Returns true if this list contains the specified element.                                                                                                                         |

### ArrayList

`public ArrayList(Collection<? extends E> c)`

E Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

#### Parameters:

c - the collection whose elements are to be placed into this list

#### Throws:

NullPointerException - If the specified collection is null

200/116

## Generics – definition - upper bound

- upper bounded generic parameter
  - ensures that the parameter implements an interface or extends a class
  - ... the implementation can thus use methods in the bound interface or class

```
public class ListOfNumber<T extends Number> {
 public Number sum() {
 // ... an operation that makes sense on Number ...
 }
```

- multiple bounds can be specified, only the first can be a class

```
public class
ListOfNumber<T extends Integer & Number & Prime >
{ ... }
```

201/203

## Generics – declaration and invocation

- the parametric type must be determined at the creation of an object
  - ... but it can be left undetermined at declaration of a reference variable (*wildcard ?*)

```
public class ListOfNumberFactory {
 public static ListOfNumber<Double> getList() {
 return new ListOfNumber<Double>();
 }
 // in some client:
 ListOfNumber<?> list = ListOfNumberFactory.getList();
 System.out.println(list.sum());
```

```
public class ListOfNumber<T> { ... }
// in some client:
ListOfNumber<?> list = new ListOfNumber<Double>();
System.out.println(list.sum());
```

202/203

## declaration - wildcard bounds

- wildcards in declarations can be upper bounded
  - public class ListOfNumber <T extends Number> implements Iterable<T> {  
    public Number sum() { // ... compute the sum ...}  
    @Override public Iterator<T> iterator() { // ... return an iterator ...}  
}

// in some client:

```
ListOfNumber<? extends Number> list = ListOfNumberFactory.getList();
for (Number number: list) { // ... do something with the Number ...}
```

- wildcards can also be lower bounded
  - ... the argument type must be the same or a superclass of the bound
  - public class IntegerListFactory {  
    public static MyList<? super Integer> getList() {  
        // ... return MyList<Integer>, MyList<Number>, MyList<Object>, etc....  
    }  
}
  - MyList<? super Integer> integerList = IntegerListFactory.getList();  
integerList.addElement(new Integer(42));

■ (TBD: is this really useful for some relevant idiom?)

203/203

## Generic Methods and Type Inference

- a Generic Method introduces type parameters
  - ... with scope limited to the method where it is declared
- at invocation the type is determined by the actual parameter passed (type inference)

204/203

- Integer is a subtype of Number but a Set<Integer> is not a subtype of a Set<Number>
  - in a Set<Integer> you cannot add a Float, which is allowed on a Set<Number>
  - the Liskov substitution principle would not be satisfied

- Exception is an event that disrupt the normal flow of a program
  - E.g. some broken invariant, a null reference, no memory available, a locked file, a db query returning an empty result set, or multiple results, ...
    - *TBD: about known unknowns and unknown unknowns ...*
- Exception handling is about providing a gentle management
  - takes a relevant effort on various levels
  - can be supported at the language level
- In the C language
  - use the returned value of functions to encode failures
  - and propagate failures up along the call stack until reaching a level that can assume the responsibility of management
  - (... often something like: `printf(...); exit(-1);`)
- Java provides a more advanced support
  - typed exceptions, throw statement, try-catch statement

207/116

- when an error occurs within a method
  - the method creates a typed exception object carrying information useful for error handling and hands it off ("throws") to the runtime environment
  - the runtime system searches a method that can handle ("catch") the error going back up along the call stack
  - exceptions propagate up through the lexical block structure of a Java method, and then up the method call stack.
  - If an exception is not caught by the block of code that throws it, it propagates to the next higher enclosing block of code.
  - ... up to the `main()` method, where the Java interpreter exits

208/116

- An exception in Java is an instance of a subclass of `java.lang.Throwable`.
- `Throwable` has two standard subclasses: `java.lang.Error` and `java.lang.Exception`.
  - Exceptions that are subclasses of `Error`
    - linkage problems related to dynamic loading,  
virtual machine problems such as running out of memory.
    - Should be considered unrecoverable, and should not be caught.
  - Exceptions that are subclasses of `Exception`
    - conditions that may be caught and recovered from.
    - They include such exceptions as `java.io.EOFException`, which signals the end of a file and `java.lang.ArrayAccessOutOfBoundsException`, which indicates that a program has tried to read past the end of an array.
- `Throwable` includes a `String` used to store a human-readable message
  - set when the exception is created, by passing an argument to the constructor method, and read with `Throwable.getMessage()`
  - Most exceptions contain only this single message, but a few add other data.
  - `java.io.InterruptedIOException`, adds `public int bytesTransferred`, specifying how much of the I/O was completed

- `try` establishes a block that is to have exceptions and abnormal exits handled
  - (abnormal exits: `break`, `continue`, `return`, or exception propagation)
- The `try` block is followed by zero or more `catch` clauses
  - each with an argument of type `Throwable` or a subclass
  - When an exception occurs, the first `catch` with matching argument is invoked (the argument matches the type of the exception object, or it is a superclass)
  - This `catch` argument is valid only within the `catch` block, and refers to the actual exception object that was thrown.
- `catch` clauses are optionally followed by a `finally` block
  - The statements of a `finally` block are guaranteed to be executed, regardless of how the code in the `try` block exits.
  - If there is not a local `catch` block to handle the exception, control transfers first to the `finally` block, and then propagates up

```
try {
 // Normally this code runs from the top of the block to the bottom
 // without problems. But it sometimes may raise exceptions or
 // exit the block via a break, continue, or return statement.
}
catch(SomeException e1) {
 // Handle an exception object e1 of type SomeException or a subclass.
}
catch(AnotherException e2) {
 // Handle an exception object e2 of type AnotherException or a subclass.
}
finally {
 // Always execute this code, after we leave the try clause,
}
```

- Not the same semantics as that of switch case in the c language
  - only one catch clause is executed

211/116

- any method that can cause a "normal exception" to occur must either catch the exception or specify the type of the exception with a throws clause

```
public void open_file() throws IOException {
 // a body that might generate an uncaught java.io.IOException
}
// ...
public void myfunc(int arg) throws MyException1, MyException2 {
//...
}
```

- the exception specified in a throws may be a superclass of the exception thrown
- if a method throws exceptions a, b, and c, all of which are subclasses of d, the throws clause may specify all of a, b, and c, or it may simply specify d.

212/116

■ Defining and Generating Exceptions

- You can signal your own exceptions with the `throw` statement.
- Often, exception objects are allocated in the same statement that they are thrown in:

```
throw new MyException("my exceptional condition occurred.");
```

- When an exception is thrown, normal program execution stops and the interpreter looks for a `catch` clause that can handle the exception.
- Execution propagates up through enclosing statements and through invoking functions until such a handler is found.
- Any `finally` blocks that are passed during this propagation are executed.
- you can use one of the exception classes already defined by Java API.
- Often, though, you will want to define and throw your own exception types.

■ *TBD: add more (perhaps later)  
on patterns and antipatterns in the usage of exceptions*

- Program structure
- Comments
- Name space
- Unicode
- Nested classes

- a Java program is made of or more class definitions in .java files
  - each compiled into a .class file of Java Virtual Machine object code
  - one class defines a `main()` method which is where the program starts
  - `main()` is a public and static method,  
so that it can be invoked by the JVM before any object has been created

```
public static void main(String argv[])
```

- `main()` may receive actual parameters as an array of strings,  
conventionally named `argv[]`

```
public class echo {
 public static void main(String argv[]) {
 for(int i=0; i < argv.length; i++)
 System.out.print(argv[i] + " ");
 System.out.print("\n");
 System.exit(0);
 }
}
```

- `main()` is declared void, so no return value is provided

- *TBD: skip this slide*

- The Java API does not allow a Java program  
to read operating system environment variables  
because they are platform-dependent.

- A Java program can look up the value of a named property

```
String homedir = System.getProperty("user.home");
String debug = System.getProperty("myapp.debug");
```

- The Java interpreter automatically defines a number of standard system  
properties when it starts up.

You can insert additional property definitions into the list by specifying the `-D` option to the interpreter:

```
%java -Dmyapp.debug=true myapp
```

- **/\* text \*/**
  - the compiler ignores the block from /\* until the next \*/ also across multiple lines
  - since Java does not support pre-processor directives like #if 0, /\* \*/ is sometimes used to comment out a block ... with the risk of nesting
- **// text**
  - the compiler ignores the text from // until the end of the line
- **/\*\* text \*/**
  - as for /\* \*/ but enables processing of tags in the comment be performed by the javadoc JDK tool
  - E.g. @author, @version, @param, @return, @since, @exception, @throws, @overrides, ...

217/116

- A Javadoc comment is embedded within delimiters /\*\* and \*/
  - Supports automated generation of documentation, with possible HTML
  - Appears next to the commented items, without any separating newline
  - Uses predefined tags @tagName
  - @author, @version, @since, @see, @param, @return, ...
- applied to document Classes

```
/**
 * @author Elvis Presley
 * @version 1.2
 * @since 2013-10-09
 */
public class AClassName{ // class body }
```
- ... and variables ...

```
/**
 * Description of the variable here.
 */
private int debug = 0;
```

218/116

### ... and methods

```
/**
 * one line description.
 * <p>
 * optional longer description, with HTML separated paragraphs
 *
 * @param variable Description
 * @param variable Description
 * @return Description
 */
public int aMethodName (...) { // method body }
```

219/116

- the c language roadmap
  - Types and values
    - Elementary types,
    - user defined types (struct), type definition
  - Variables
    - Declaration and reference
    - Pointers, Arrays, static or dynamic allocation
  - Expressions
    - Operators, side-effects and returned values,
    - functions, binding technique, definition declaration and reference
  - Statements
    - Program structure
- ... makes a 3 parties game, that applies to Java too
  - variables encode values (in types)
  - expressions return a value and produce side effects on variable
    - variables affect returned values and side-effects
  - statements control the flow of execution of expressions
    - expressions returned values affect statements

220/116

- As in the c language
  - an expression is a combination of references to variables and constants through operators
  - the semantics of an expression is made of the returned value (in some type) and the side effects produced
  - the invocation of a method, i.e. the reference to a method, is an expression, which in fact returns a value and may have side-effects
- Java operators are basically the same as in the c language, except for
  - instanceof operator
    - returns true iff the object on its left-hand side is not null and is an instance of the class or implementation of the interface specified on its right-hand side.
  - the expression chaining comma operator is suppressed
  - the + operator applied to String values concatenates them
    - If only one operand of + is a String, the other one is converted to a string.
    - The conversion is done automatically for primitive types, and by calling the `toString()` method of non-primitive types.
  - *TBD: add here something on lambda expressions introduced since Java8*

| Prec. | Operator    | Operand Type      | Ass                               | Operation Performed                    |
|-------|-------------|-------------------|-----------------------------------|----------------------------------------|
| 1     | ++          | arithmetic        | R                                 | pre-or-post increment                  |
|       | --          | arithmetic        | R                                 | pre-or-post decrement                  |
|       | +, -        | arithmetic        | R                                 | unary plus, unary minus                |
|       | ~, ~        | integral          | R                                 | bitwise complement (unary)             |
|       | !           | boolean           | R                                 | logical complement (unary)             |
|       | (type)      | any               | R                                 | cast                                   |
| 2     | *, /, %     | arithmetic        | L                                 | multiplication, division, remainder    |
| 3     | +, -        | arithmetic        | L                                 | addition, subtraction                  |
|       | +           | String            | L                                 | string concatenation                   |
| 4     | <<          | integral   L      | left shift                        |                                        |
|       | >>          | integral   L      | right shift with sign extension   |                                        |
|       | >>>         | integral   L      | right shift with zero extension   |                                        |
| 5     | <, <=       | arithmetic        | L                                 | less than, less than or equal          |
|       | >, >=       | arithmetic        | L                                 | greater than, greater than or equal    |
|       | instanceof  | object, type      | L                                 | type comparison                        |
| 6     | ==          | primitive   L     | equal (have identical values)     |                                        |
|       | !=          | primitive   L     | not equal (have different values) |                                        |
|       | ==          | object            | L                                 | equal (refer to same object)           |
|       | !=          | object            | L                                 | not equal (refer to different objects) |
| 7     | &           | integral   L      | bitwise AND                       |                                        |
|       | &           | boolean   L       | boolean AND                       |                                        |
| 8     | ^           | integral   L      | bitwise XOR                       |                                        |
|       | ^           | boolean   L       | boolean XOR                       |                                        |
| 9     |             | integral   L      | bitwise OR                        |                                        |
|       |             | boolean   L       | boolean OR                        |                                        |
| 10    | &&          | boolean   L       | conditional AND                   |                                        |
| 11    |             | boolean   L       | conditional OR                    |                                        |
| 12    | ?:          | boolean, any, any | R                                 | conditional (ternary)operator          |
| 13    | =           | variable, any     | R                                 | assignment                             |
|       | *=, /=,     | variable, any     | R                                 | assignment with operation              |
|       | %=, +=, -=, |                   |                                   |                                        |
|       | <=, >=,     |                   |                                   |                                        |
|       | >>=,        |                   |                                   |                                        |
|       | &=, ^=,  =  |                   |                                   |                                        |

■ **for loop**

- to surrogate the lack of the c comma operator, multiple comma-separated expressions are allowed in the initialization and increment sections (not in the test section),

```
int i; String s;
for(i=0, s="testing"; (i<10) && (s.length()>= 1); i++, s=s.substring(1))
{ System.out.println(s); }
```

- variables can be declared in the initialization section of the loop and will have the loop as their scope and lifetime

```
for(int i = 0; i < my_array.length; i++)
System.out.println("a[" + i + "] = " + my_array[i]);
// one more:
int j = -3; // this j remains unchanged.
for(int i=0, j=10; i < j; i++, j--) System.out.println("k = " + i*j);
```

■ **"enhanced" for loop**

- designed and recommended for iteration through Collections and arrays

```
class EnhancedForDemo {
 public static void main(String[] args){
 int[] numbers =
 {1,2,3,4,5,6,7,8,9,10};
 for (int item : numbers) {
 System.out.println("Count is: " + item);
 }
 }
}
```

- **TBD: skip this slide – this is about the evil**
- **labelled break and continue**
  - may optionally be followed by a label that specifies an enclosing loop (for `continue`) or any enclosing statement (for `break`).
  - After the `break` statement is executed, any required `finally` clauses are executed, and control resumes at the statement following the terminated statement.
- ```
test: if (check(i)) {
    try {
        for(int j=0; j < 10; j++) {
            if (j > i) break;           // terminate just this loop
            if (a[i][j] == null) break test; // do the finally clause and
                                            // terminate the if statement.
        }
        finally{ cleanup(a, i, j); }
    }
}
```
- **no goto**
 - `goto` is a reserved word, but not currently a statement of the language

- **TBD: skip this slide – not essential at this stage**
- **The synchronized Statement**
 - Java is a multithreaded system
 - "Critical sections" of code that must not be executed simultaneously
 - `synchronized (expression) statement`
 - `expression` is an expression that must resolve to an object or an array.
 - The `statement` is the code of the critical section,
 - Note that you do not have to use the `synchronized` statement unless your program creates multiple threads that share data
 - ```
public static void SortIntArray(int[] a) {
 // Sort the array a. This is synchronized so that some other
 // thread can't change elements of the array while we're sorting it.
 // At least not other threads that protect their changes to the
 // array with synchronized.
 synchronized (a) {
 // do the array sort here.
 }
}
```
  - The `synchronized` keyword is more often used as a method modifier in Java. When applied to a method, it indicates that the entire method is a critical section

- *TBD: this has probably already appeared before, with the package statement*

- The import statement

- allows us to refer to classes by abbreviated names.
- import statements must appear after the package statement, if any, and before any other statements in a Java file.

```
package games.tetris;
import java.applet.*;
import java.awt.*;
```

227/116

- No Global Variables

- every variable and method is declared within a class
- Also, every class is part of a *package*.
- every Java variable or method may be referred to by its fully qualified name which consists of the package name, the class name, and the field name, all separated by periods.
- Package names are themselves usually composed of multiple period-separated components

```
david.games.tetris.SoundEffects.play()
```

- Packages, Classes, and Directory Structure

- every compiled class is stored in a separate file.  
The name of this file must be the same as the name of the class, with the extension *.class* added.
- E.g., if the fully qualified name of a class is `david.games.tetris.SoundEffects`, the full path of the class file must be `david/games/tetris/SoundEffects.class`.\*

228/116

## The Name Space: Packages, Classes, and Fields

- **A file of Java source code should have the extension `.java`.**

- It consists of one or more class definitions.
- If more than one class is defined in a `.java` file, only one of the classes may be declared `public`, and that class must have the same name as the source file
- If a source file contains more than one class definition, those classes are compiled into multiple `.class` files.

229/116

## The Name Space: Packages, Classes, and Fields

- **The Java Class Path**

- The Java interpreter looks up its system classes relative to the directories specified by the `-classpath` argument

```
setenv CLASSPATH .:~/classes:/usr/local/classes
```

- **Globally Unique Package Names**

- The Java designers have proposed an Internet-wide unique package naming scheme that is based on the domain name

230/116

## The Name Space: Packages, Classes, and Fields

### ■ The import Statement

- makes Java classes available to the current class under an abbreviated name.
- `import` doesn't actually make the class available or "read it in"; it simply saves you typing and makes your code more legible.
- Any number of `import` statements may appear in a Java program.
- They must appear, however, after the optional `package` statement, and before the first class or interface definition in the file.

```
import java.awt.image;
```

- allows `java.awt.image.ImageFilter` to be called `image.ImageFilter`:

```
import java.util.Hashtable;
```

- allows you to type `Hashtable` instead of `java.util.Hashtable`:

```
import java.lang.*;
```

makes the core classes of the language available by their unqualified class names

231/116

## No Preprocessor

### ■ Java does not include any kind of preprocessor

- rules out `#define`, `#include`, and `#ifdef`

### ■ Defining Constants

- Any variable declared `final` in Java is a constant (its value must be specified with an initializer when it is declared),

- The variable `java.lang.Math.PI` is an example:

```
public final class Math {
 ...
 public static final double PI = 3.14159.....;
 ...
}
```

- C convention of using CAPITAL letters for constants is also a Java convention
- Java constants have globally unique hierarchical names, while constants defined with the C preprocessor always run the risk of a collision
- Java constants are strongly typed

232/116

■ Defining Macros

- Java has no equivalent to this sort of macro, but compiler technology has advanced to a point where macros are rarely necessary any more.
- A good Java compiler should automatically be able to "inline" short Java methods where appropriate.

■ Including Files

- Java does not have a `#include` directive when the Java compiler needs to read in a specified class file, it knows exactly where to find it
- Java does not make the distinction between *declaring* a variable or procedure and *defining* it that C does.
- Java does have an `import` statement, which is superficially similar to the C preprocessor `#include` directive.
- since the compiler implicitly imports all the classes of the `java.lang` package, we can refer to the constant `java.lang.Math.PI` by `Math.PI`.

■ Conditional Compilation

- Java does not have any form of the C `#ifdef` or `#if` directives to perform conditional compilation.
- a good Java compiler does not compile code if it can prove that it will never execute
- placing code within an `if (false)` block is equivalent to surrounding it with `#if 0` and `#endif` in C.  
`private static final boolean DEBUG = false;`
- with such a constant defined, any code within an `if (DEBUG)` block is not actually compiled into the class file.

- Java characters, strings, and identifiers (e.g., variable, method, and class names) are composed of 16-bit Unicode characters.
    - The Unicode character set is compatible with ASCII
    - the first 256 characters (0x0000 to 0x00FF) are identical to the ISO8859-1 (Latin-1) characters 0x00 to 0xFF.
  - A character may be represented with the Unicode escape sequence \uxxxx
  - Java also supports all of the standard C character escape sequences, such as \n, \t, and \xxx (where xxx is three octal digits).
  - Java does not support line continuation with \ at the end of a line.
    - Long strings must either be specified on a single long line, or they must be created from shorter strings using the string concatenation (+)
- *TBD: add here or later some snippets about effective usage of exceptions*

- Nested class: a class defined within another class
  - can be either static and non-static (in the latter case called *inner class*)

```
class OuterClass { ... // outer top-level class
 static class StaticNestedClass { ... // static nested class
 }
 class InnerClass { ... // (non static nested) inner class
 }
}
```
- For what?
  - group "helper classes" that are only used in one place (A kind of private class)
  - lead to more readable and maintainable code: place code closer to where it is used
  - increase encapsulation:
    - e.g. Some A members should be private, but need to be accessed from a class B; if B becomes a nested class of A, A's members can be declared private, and B is hidden from the outside world.

- a static nested class is associated with its outer class, not with an instance

```
class OuterClass { ... // outer top-level class
 static class StaticNestedClass { ... } // static nested class
}
```

- Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

237/116

- an inner class is associated with an instance of its enclosing class
  - an instance of an inner class exists within an instance of the outer class
  - and has direct access to that object's methods and fields.
- class OuterClass { ... // outer enclosing class
 class InnerClass { ... // inner class
 }
}
- an inner object can be instantiated only after an instance of its enclosing class

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

  - *Viceversa, the inner instance is not always necessarily created*
- inner classes can be either *Local Classes*
  - defined in a *block*, which is a group of zero or more statements in a compound
  - typically, in the body of a method.
  - (*much like a local type for local variables*)
- ... or *Anonymous Classes*
  - like local classes except that they do not have a name.

238/116

- Example: a local class defined in the body of a method definition

```
public class LocalClassExample {
 static String regularExpression = "[^0-9]";
 public static void validatePhoneNumber(String phoneNumber1, String phoneNumber2) {
 // ...
 class PhoneNumber {
 // a class local to the definition of a (static) method
 String formattedPhoneNumber = null;
 PhoneNumber(String phoneNumber) {
 // local class constructor
 String currentNumber = phoneNumber.replaceAll(regularExpression,
 "");
 if (currentNumber.length() == numberLength)
 formattedPhoneNumber = currentNumber;
 else
 formattedPhoneNumber = null; }
 public String getNumber() { return formattedPhoneNumber; } // local class method }
 // ... outer method definition continues, and creates an instance of the local class
 PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
 // ...
 // a method of the local class is invoked on its instance
 if (myNumber1.getNumber() == null) System.out.println("First number is invalid");
 // ...
 }
}
```

*Not much used a class defined within a method:  
it could rather be defined in the class not in the method !  
Yet, useful for introducing concepts*

239/116

- A "real" nested class: nested but not in a method

- Making this static is an accident, could work the same without, with different concept
- Class PhoneNumber is made static because it does not need to access instance variables or methods of its enclosing class

```
public class LocalClassExample {
 static String regularExpression = "[^0-9]";
 private static class PhoneNumber { // note private and static
 String formattedPhoneNumber = null;
 PhoneNumber(String phoneNumber) {
 // local class constructor
 String currentNumber = phoneNumber.replaceAll(regularExpression,
 "");
 if (currentNumber.length() == numberLength)
 formattedPhoneNumber = currentNumber;
 else
 formattedPhoneNumber = null; }
 public String getNumber() { return formattedPhoneNumber; } // local class method }
 }
 public static void validatePhoneNumber(String phoneNumber1, String phoneNumber2) {
 // ... outer method using the nested class
 PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
 // ...
 // a method of the local class is invoked on its instance
 if (myNumber1.getNumber() == null) System.out.println("First number is invalid");
 // ...
 }
}
```

240/116

## Local Classes: access and declaration limitations

- Local classes in static methods, can only refer to static members of the enclosing class
  - e.g. local class PhoneNumber defined in the static method validatePhoneNumber, can access the member variable regularExpression only because this is static

```
public class LocalClassExample {
 static String regularExpression = "[^0-9]";
 public static void validatePhoneNumber(String phoneNumber1, String phoneNumber2) {
 // ...
 class PhoneNumber {
 // a class local to the definition of a method
 String formattedPhoneNumber = null;
 PhoneNumber(String phoneNumber) {
 // local class constructor
 String currentNumber = phoneNumber.replaceAll(regularExpression,
 "");
 ...
 }
 }
 ...
 }
}
```
- A local class cannot itself define most types of static members, because it does not exist before the instance
  - cannot declare an interface inside a block; interfaces are inherently static.
  - cannot declare static initializers in a local class
  - can have static members provided that they are constant variables.
- ... moreover
  - a local class has access to local variables if they are declared final.
  - since Java SE8, a local class defined in a method can access the method's parameters.

241/116

## Anonymous Classes

- anonymous classes appear as expressions, and do not introduce a name
  - extends the invocation of a constructor with a class definition contained in a block

```
public class HelloWorldAnonymousClasses { // enclosing class
 interface HelloWorld { public void greet(); // an interface local to the class
 public void greetSomeone(String someone); }
 public void sayHello() {
 class EnglishGreeting implements HelloWorld { // a class local to a method
 String name = "world";
 public void greet() { greetSomeone("world"); }
 public void greetSomeone(String someone) { name = someone;
 System.out.println("Hello " + name); }
 } // end of the local class def
 HelloWorld englishGreeting = new EnglishGreeting();
 HelloWorld frenchGreeting = new HelloWorld()
 // anonymous local class extending a constructor invocation expression
 String name = "tout le monde";
 public void greet() { greetSomeone("tout le monde"); }
 public void greetSomeone(String someone) {name = someone;
 System.out.println("Salut " + name); }
 }; // end of the anonymous class definition, and of the enclosing statement
 ...
 }
}
```

242/116

## Anonymous Classes: access and declaration limitations

- An anonymous class
  - has access to the members of its enclosing class.
  - can access local variables in its enclosing scope provided that they are declared final
- An anonymous class
  - cannot declare static initializers or member interfaces
  - can have static members if they are constant variables.
  - Cannot declare a constructor
  - can declare Fields, Extra methods, Instance initializers, Local classes

243/116

## More on shadowing with nested classes

- Shadowing occurs whenever a name declared in a scope equal to another in the enclosing scope
- ```
public class ShadowTest {                                // enclosing class
    public int x = 0;
    class FirstLevel {                                 // inner class (non static, named)
        public int x = 1;
        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }
    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
// produced output:
//      x = 23      this.x = 1      ShadowTest.this.x = 0
```
- may happen for the same reason why formal parameters often have the same name of local variables that they carry.

244/116

Item #18: Favor static member classes over nonstatic

- <Precondition: consolidate nested classes as a basic java construct>
- A *nested class* is a class defined within another class.
 - should exist only to serve its enclosing class
- four kinds of nested classes:
 - *static member classes*,
 - *Inner classes*
 - *Non-static member classes*
 - *anonymous classes*
 - *local classes*

245/66

Item 18: Favor static member classes over nonstatic

- Use a *non-static member class* to define an adapter

```
public class MySet extends AbstractSet {  
    private class MyIterator implements Iterator {  
        // methods here access instance variables and methods of the enclosing  
        // class  
        ...  
    }  
    ... // Bulk of the class omitted  
    public Iterator iterator() {return new MyIterator();}  
}
```
- Use a *static member class*
whenever it does not require access to an enclosing instance,
 - avoids each instance to contain an extraneous reference to the enclosing object,
with loss of time and space
 - allows allocation even when no enclosing instance exists
 - Make the static class private
to represent components of the object represented by the enclosing class.

246/66

Item 18: Favor static member classes over nonstatic

- Use *anonymous classes* only if there is no need to refer to them after they are instantiated
 - typically implement only methods in their interface or superclass
 - They do not declare any new methods (*with a new name*), as there is no nameable type to access new methods.
 - Because they occur in the midst of expressions, they should be very short, perhaps twenty lines or less, for readability
- One common use of an anonymous class is to create a *function object*
 - E.g. a Comparator instantiated in a subclass/implementation of class/interface Comparator that overrides/implements the compare method (java.util.Comparator is an interface, but, with a class would be the same)

```
Arrays.sort(args, new Comparator() {  
    public int compare(Object o1, Object o2) {  
        return ((String)o1).length() - ((String)o2).length();  
    } // closes the anonymous class extension of constructor invocation  
}); // closes the embedding statement
```
- other common uses of anonymous classes
 - to create a *process object*, such as a Thread, Runnable, or TimerTask instance.
 - within a static factory method (see Item 16).
 - A command pattern

247/66

Item 18: Favor static member classes over nonstatic

- use *anonymous classes* to give different implementations for abstract methods
 - example: a calculator with different implementations of an eval operation (note doubly nested classes)

```
public class Calculator {  
    public static abstract class Operation {  
        private final String name;  
        Operation(String name) { this.name = name; }  
        public String toString() { return this.name; }  
        abstract double eval(double x, double y); // abstract method, implemented in nested classes  
        public static final Operation PLUS = new Operation("+") { // doubly nested anonymous class  
            double eval(double x, double y) { return x + y; } };  
        public static final Operation MINUS = new Operation("-") { // doubly nested anonymous class  
            double eval(double x, double y) { return x - y; } };  
    } // end of nested class  
    // calculate() will bind different implementations depending on the subtype of op  
    public double calculate(double x, Operation op, double y) {return op.eval(x, y);} }  
  
Calculator c= new Calculator();  
double z = c.calculate(getX(...), Calculator.Operation.PLUS,getY(...));
```

248/66

Item 18: Favor static member classes over nonstatic

- *Local classes* are probably the least frequently used kind of nested class.
- may be declared anywhere that a local variable may be declared and obeys the same scoping rules.
- Local classes have some attributes in common with each of the other three kinds of nested classes.
 - Like member classes, they have names and may be used repeatedly.
 - Like anonymous classes, they have enclosing instances if and only if they are used in a nonstatic context.
 - Like anonymous classes, they should be short so as not to harm readability

Software Engineering - A.A. 20/21

UML diagrams for the implementation perspective:
Class, Object, Sequence and Collaboration Diagrams
(with emphasis on notation more than method)



Enrico Vicario

Dipartimento di Ingegneria dell'Informazione
Laboratorio Tecnologia del Software – stlab.dinfo.unifi.it
Università di Firenze

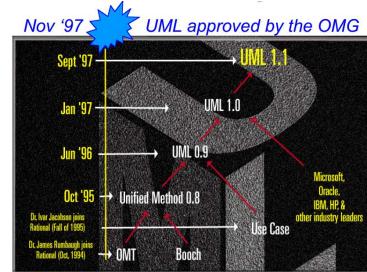
enrico.vicario@unifi.it, stlab.dinfo.unifi.it/vicario

1/42

- Class diagram
- Object diagram
- UML profiles
- Sequence and Collaboration diagrams
- Tools
- Implementation, specification and conceptual perspectives

2/42

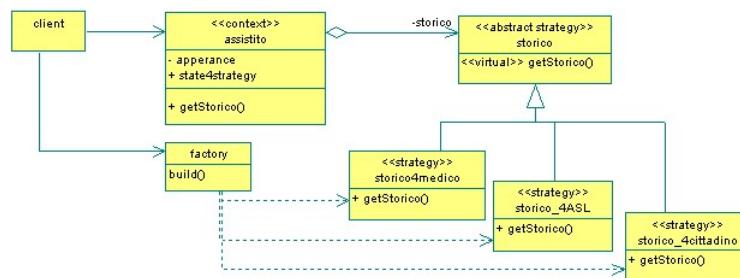
- UML: Unified Modeling Language
 - Language: a notation, later accompanied by a method of use
 - Modeling: suitable for abstraction, mainly Object Oriented
 - Unified: a suite of diagrams from different approaches, some with more prominent role (core diagrams)



- Class diagram
 - a core diagram in the UML suite
 - natively intended as a way to capture an OO implementation, but equally relevant in other levels of abstraction and for different intents
- We start here with the implementation perspective
 - not necessarily the most relevant, but the easiest one at this point

3/42

- Class Diagrams serve to capture the organization of "classes"
 - allocation and partition of *responsibilities* among classes (object types)
 - relations of use, creation, association, aggregation, composition, specialization/generalization,
 - *TBD: apologies for naming, often not in Java style in these slides!*



- Provides a static view
 - open to several potential instantiations and behaviors
 - (more answers later with object diagrams and sequence diagrams)

4/42

- Name + attributes + methods of a class
 - intended as a construct of some OO language

Stack
TOS Size
init() push() pop() isEmpty() isFull()

- Visibility (access modifiers) of attributes and methods
 - + public: in any class
 - # protected: within the package and in sub-classes
 - ~ package-private: within the package
 - - private: within the class

Stack
-TOS -Size
+init() +push() +pop() -isEmpty() -isFull()

5/42

- Types of attributes, signature of parameters and returned value
 - Pascal-like notation (sigh!)

Stack
-TOS -Size
+init(size: int); void +push(value: int); bool +pop(ptr: int*); bool -isEmpty(); bool -isFull(); bool

- virtual methods (in Java, abstract methods)
 - operation()
 - <>abstract>> operation()
 - operation()=0
 - operation();

Stack
-TOS -Size +init(size: int); void +push(value: int); bool +pop(ptr: int*); bool -isEmpty(); bool -isFull(); bool +Operation(); void

medico_curante
- codice_regionale
+ elenco_assistiti

aggiungi_assistito()
<>virtual>> + fornisce_orario()

6/42

Class stereotypes

- **Stereotypes**

- add specialized semantics according to some profile
- can be represented by <<...>>, or by some graphical notation
- (more later)

```
<<persistente>>
medico_curante
- codice : codice_regionale
+ elenco : elenco_assistiti

# aggiungi_assistito()
+ fornisce_orario()
```

- To be used with judice (and incompleteness)

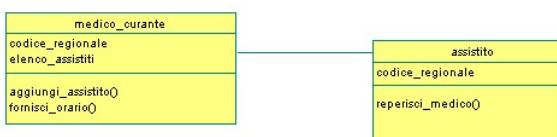
- the final goal is to communicate among developers
- within some context: a lab, an enterprise, a set of enterprises involved in a common project, ...

7/42

Association

- Documents a structural relation between two classes

- the objects of one class maintain a reference to some object in the other class

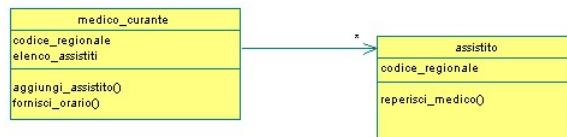


- Navigation: which class has a reference to the other one

- no arrows means that the association is bi-directional !

- Multiplicity

- all variants, but most useful to distinguish between one and many (*)

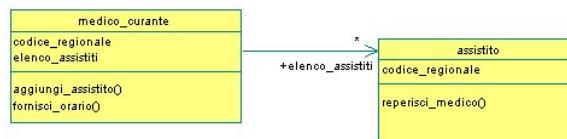


8/42

Association

■ Roles

- document the role played by an association end
basically, the name of the reference in the source class



■ Association name

- less useful than the role

9/42

Associations of subtype <<uses>> and <<creates>>

■ Creates

- documents the situation where methods of one class create instances of objects in another class

■ Uses

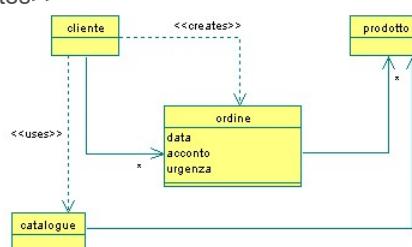
- documents the situation where methods of one class refer to methods or attributes of another class
- different meaning wrt an association:
uses does not correspond to a class variable,
but rather to some parameter passed in some method invocation
- is about dependence

■ Dashed line

- stereotypes <<uses>> and <<creates>>
- <<creates>> usually suppressed
- (yet, suppressing <<creates>>
may be more convenient)

■ uses and creates

- convey key information in the architectural perspective



10/42

Associazione e uso

- L'associazione documenta una relazione strutturale tra due tipi
 - il source object ha nel suo stato un field che referenzia il target
 - il field esiste in tutto il lifecycle del source
 - (il field che realizza l'associazione non si rappresenta nella source class)

```

classDiagram
    class medico_curante {
        codice_regionale
        elenco_assistiti
        aggiungi_assistito()
        fornisce_orario()
    }
    class assistito {
        codice_regionale
        reperiscono_medico()
    }
    medico_curante "1" -- "*" assistito
  
```
- la relazione d'uso documenta una associazione temporanea
 - un metodo nel source object usa il target object, attraverso un riferimento che esiste nel lifetime del metodo, e.g. utilizzando un parametro attuale
 - spesso omessa nel modello per il bene della semplicità
- e.g. il link tra un medico e i suoi assistiti è una associazione
 - il link tra un medico e gli assistiti che si sono iscritti per le visite oggi è probabile che sia una relazione di uso

11/42

Aggregation and composition

- Aggregation and composition specialize the semantics of association
 - stereotypes provided with graphical appearance
 - emphasizing the concept of inclusion of a part within a whole

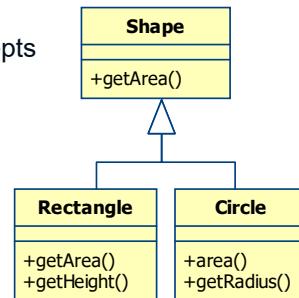
```

classDiagram
    class Libretto
    class Studente
    class CorsoDiLaurea
    class Scuola
    Libretto --> Studente
    Studente --> CorsoDiLaurea
    CorsoDiLaurea --> Scuola
  
```
- Aggregation documents a relation “Is part of”
 - the same part can be aggregated within multiple contexts
 - e.g. a School aggregates multiple Courses, and each Course can be shared by multiple Students
- Composition has the stronger meaning of inclusion
 - the composed objects exists only in the context
 - E.g. the RegistryOfExams belongs to a Student
- In the implementation perspective, this is about
 - objects that derive identity from their embedding context, and calls into play concepts of sharing and defesive copy)
 - more firm semantics in Object Relational Mapping (embedded type, shared references, ...)

12/42

Generalization

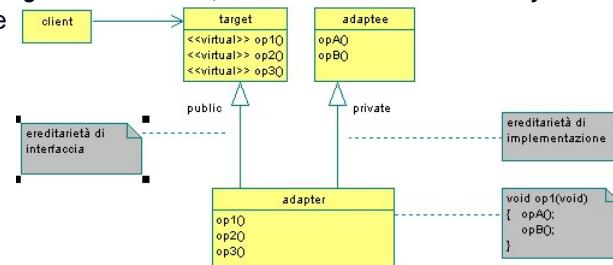
- The relation of generalization documents subtyping
 - a class extends some other class (inheritance) or implements some interface (substitutability)
 - *extends* drawn with continuous line, *implements* drawn with dotted line
- About substitutability:
 - a Rectangle can be passed whenever a Shape is expected, viceversa may not hold (Liskov substitution principle)
- (almost) equal notation for different concepts
 - specialization and generalization
 - inheritance and substitutability
 - (Java) extends and implements
 - (dotted vs continuous lines are relevant!)



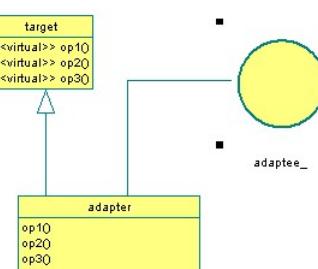
13/42

Generalization

- *TBD: apologies for notation, these slides were for c++ years ago*
- Multiple inheritance

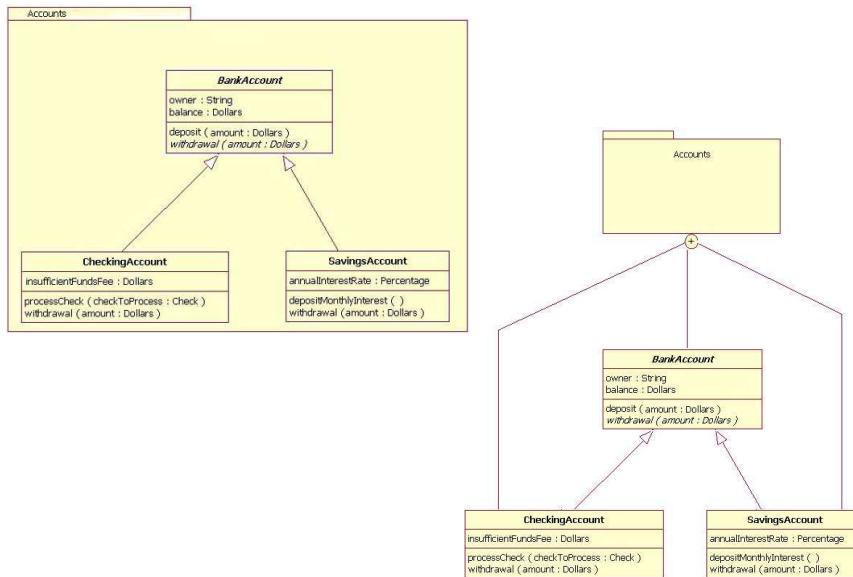


- Class extension



14/42

■ Classes in a package



15/42

■ Documents organization of class instances in some significant or exemplificative stage of execution

- represents objects, with their valued attributes, and relations between objects

■ Not a core diagram

- Useful to understand the consequences of a class diagram ... and to document complex relations in the conceptual perspective
- e.g. un medico è in relazione con i suoi assistiti, ma egli stesso è un assistito a cui è attribuito un altro medico (?!);
- e.g. defensive copies, bi-directional associations in mappedBy form, Decorator, Composite, ...

■ Is a static diagram

- but reflects an actual case that may occur in a transient stage of execution

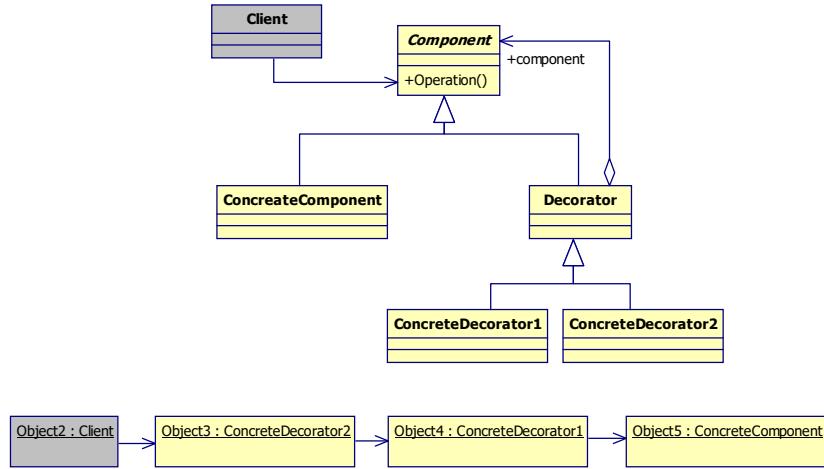
■ The topology of dynamic objects can be quite different than that of static classes

- E.g. the Composite or Decorator design patterns
- cycles in the class diag vs unbounded configurations in the object diag
- multiplicity in the class diag (Decorator vs Composite)

16/42

Object diagram

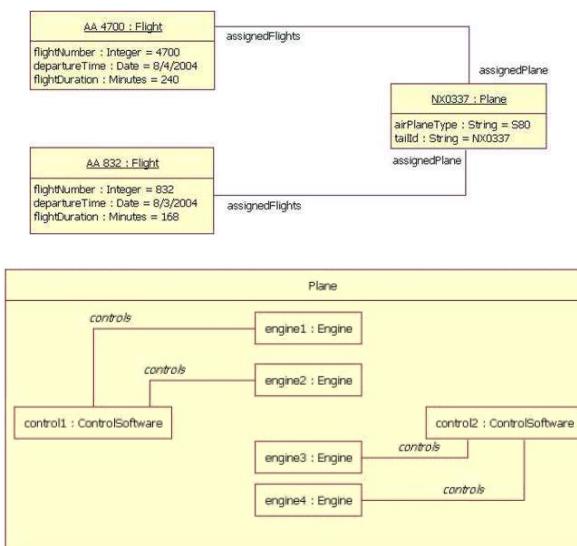
- On the different perspectives of types and instances
 - when writing code you see classes
 - ... but run-time execution is made by instances



17/42

Class instances – object diagram

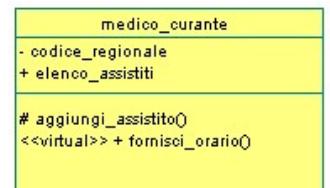
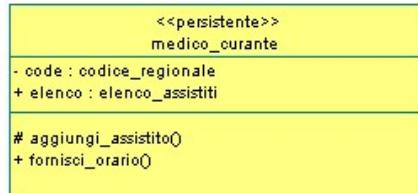
- class instances (objects) can be represented within a class diagram



18/42

■ **Stereotypes**

- Specializes the semantics of some construct of the UML diagram
- Denoted by <> brackets



■ Sometimes associated with a graphical notation

- E.g. the signs for composition, aggregation, use or creation, interface, ...

19/42

- Defines a reference conceptual model for some domain
- Specializes UML through stereotypes capturing domain specific concepts
- Shared and agreed within some context of development or standardized by the Object Management Group
 - SysML: System Modeling language (Relevant)
 - MARTE: Modeling and Analysis of Real Time and Embedded Systems
 - ...and many more
- see profiles at <http://www.omg.org/spec/>

20/42

- A UML profile is a specification that does one or more of the following:
 - Identifies a subset of the UML metamodel.
 - Specifies “well-formedness rules” beyond those specified by the identified subset of the UML metamodel.
“Well-formedness rule” is a term used in the normative UML metamodel specification to describe a set of constraints written in UML’s Object Constraint Language (OCL) that contributes to the definition of a metamodel element.
 - Specifies “standard elements” beyond those specified by the identified subset of the UML metamodel.
“Standard element” is a term used in the UML metamodel specification to describe a standard instance of a UML stereotype, tagged value or constraint.
 - Specifies semantics, expressed in natural language, beyond those specified by the identified subset of the UML metamodel.
 - Specifies common model elements, expressed in terms of the profile.
- Used not only in the implementation perspective

<ul style="list-style-type: none"> ■ SysML 	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Specification Name:</td><td>OMG Systems Modeling Language (SysML)</td></tr> <tr> <td>Description:</td><td>The OMG Systems Modeling Language (OMG SysML™) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities. In particular, the language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametrics, which is used to integrate with other engineering analysis models. SysML represents a subset of UML 2.0 with extensions needed to satisfy the requirements of the UML™ for Systems Engineering RFP. SysML uses the OMG XML Metadata Interchange (XMI®) to exchange modeling data between tools.</td></tr> <tr> <td>OMG Cross Reference:</td><td>Domain Specifications</td></tr> </table>	Specification Name:	OMG Systems Modeling Language (SysML)	Description:	The OMG Systems Modeling Language (OMG SysML™) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities. In particular, the language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametrics, which is used to integrate with other engineering analysis models. SysML represents a subset of UML 2.0 with extensions needed to satisfy the requirements of the UML™ for Systems Engineering RFP. SysML uses the OMG XML Metadata Interchange (XMI®) to exchange modeling data between tools.	OMG Cross Reference:	Domain Specifications																
Specification Name:	OMG Systems Modeling Language (SysML)																						
Description:	The OMG Systems Modeling Language (OMG SysML™) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities. In particular, the language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametrics, which is used to integrate with other engineering analysis models. SysML represents a subset of UML 2.0 with extensions needed to satisfy the requirements of the UML™ for Systems Engineering RFP. SysML uses the OMG XML Metadata Interchange (XMI®) to exchange modeling data between tools.																						
OMG Cross Reference:	Domain Specifications																						
<ul style="list-style-type: none"> ■ MARTE 	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Specification Name:</td><td>UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)</td></tr> <tr> <td>Description:</td><td>This specification of a UML™ profile adds capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES). This extension, called the UML profile for MARTE (in short MARTE), provides support for specification, design, and verification/validation stages. This new profile is intended to replace the existing UML Profile for Schedulability, Performance and Time.</td></tr> <tr> <td>Keywords:</td><td>Allocation Modeling, communication, computation, core elements, Detailed Resource Modeling (DRM), General Component Model (GCM), Generic Quantitative Analysis Modeling (GQAM), Generic Resource Modeling (GRM), Non-functional Properties Modeling (NFP), Performance Analysis Modeling (PAM), schedulability analysis modeling, Time modeling</td></tr> <tr> <td>Latest / past specifications:</td><td>Current version: 1.0 Past versions: n/a</td></tr> <tr> <td>Related OMG Specifications:</td><td>UML (2.0 Infrastructure, 2.0 and 2.1 Superstructure), UML Profile for SPT, XMI</td></tr> <tr> <td>OMG Cross Reference:</td><td>Specialized CORBA Specifications</td></tr> <tr> <td>Related Industry Standards:</td><td></td></tr> <tr> <td>Most recent IPR and Implementation questionnaire responses:</td><td> <table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">BCG</td> <td style="width: 25%;">Alcatel-Lucent</td> <td style="width: 25%;">ARTiSAN Software</td> </tr> <tr> <td>Lockheed Martin</td> <td>Soiteam</td> <td>TALES</td> </tr> </table> </td></tr> </table>	Specification Name:	UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)	Description:	This specification of a UML™ profile adds capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES). This extension, called the UML profile for MARTE (in short MARTE), provides support for specification, design, and verification/validation stages. This new profile is intended to replace the existing UML Profile for Schedulability, Performance and Time.	Keywords:	Allocation Modeling, communication, computation, core elements, Detailed Resource Modeling (DRM), General Component Model (GCM), Generic Quantitative Analysis Modeling (GQAM), Generic Resource Modeling (GRM), Non-functional Properties Modeling (NFP), Performance Analysis Modeling (PAM), schedulability analysis modeling, Time modeling	Latest / past specifications:	Current version: 1.0 Past versions: n/a	Related OMG Specifications:	UML (2.0 Infrastructure, 2.0 and 2.1 Superstructure), UML Profile for SPT, XMI	OMG Cross Reference:	Specialized CORBA Specifications	Related Industry Standards:		Most recent IPR and Implementation questionnaire responses:	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">BCG</td> <td style="width: 25%;">Alcatel-Lucent</td> <td style="width: 25%;">ARTiSAN Software</td> </tr> <tr> <td>Lockheed Martin</td> <td>Soiteam</td> <td>TALES</td> </tr> </table>	BCG	Alcatel-Lucent	ARTiSAN Software	Lockheed Martin	Soiteam	TALES
Specification Name:	UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)																						
Description:	This specification of a UML™ profile adds capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES). This extension, called the UML profile for MARTE (in short MARTE), provides support for specification, design, and verification/validation stages. This new profile is intended to replace the existing UML Profile for Schedulability, Performance and Time.																						
Keywords:	Allocation Modeling, communication, computation, core elements, Detailed Resource Modeling (DRM), General Component Model (GCM), Generic Quantitative Analysis Modeling (GQAM), Generic Resource Modeling (GRM), Non-functional Properties Modeling (NFP), Performance Analysis Modeling (PAM), schedulability analysis modeling, Time modeling																						
Latest / past specifications:	Current version: 1.0 Past versions: n/a																						
Related OMG Specifications:	UML (2.0 Infrastructure, 2.0 and 2.1 Superstructure), UML Profile for SPT, XMI																						
OMG Cross Reference:	Specialized CORBA Specifications																						
Related Industry Standards:																							
Most recent IPR and Implementation questionnaire responses:	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">BCG</td> <td style="width: 25%;">Alcatel-Lucent</td> <td style="width: 25%;">ARTiSAN Software</td> </tr> <tr> <td>Lockheed Martin</td> <td>Soiteam</td> <td>TALES</td> </tr> </table>	BCG	Alcatel-Lucent	ARTiSAN Software	Lockheed Martin	Soiteam	TALES																
BCG	Alcatel-Lucent	ARTiSAN Software																					
Lockheed Martin	Soiteam	TALES																					

Some standardized Profiles 2/3

Specification Name: UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms		
Description: This specification defines a set of UML extensions to represent Quality of Service and Fault-Tolerance concepts. These extensions reduce the problems of UML 2.0 for the description of Quality of Service and Fault-Tolerance properties, and integrate the extensions in two basic general frameworks (QoS Modeling Framework, and FT Modeling Framework). The general framework for the description of QoS requirements and properties gives the support to describe vocabulary that is used in high quality technologies (e.g., real-time, fault-tolerant).		
Keywords: coherence, demand, dependability, efficiency, integrity, latency, mitigation, security, risk assessment, scalability, throughput		
Latest / past specifications: Current version: 1.1 Past versions		
Contact Information: Realtime, Embedded and Specialized Systems PTF		
Related OMG Specifications: CORBA/IOP , UML , UML Profile for SPT		
OMG Cross Reference: Specialized CORBA Specifications		
Related Industry Standards:		
Most recent IPR and Implementation questionnaire responses: BCQ THALES Universidad Politecnica de Madrid		

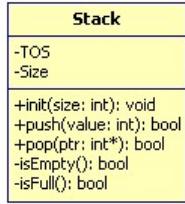
23/42

Some standardized Profiles 3/3

Specification Name: UML Profile for Schedulability, Performance and Time		
Description: Specifies a UML profile that defines standard paradigms of use for modeling of time-, schedulability- and performance-related aspects of real-time systems ¹¹ that (1.) enable the construction of models that can be used to make quantitative predictions regarding these characteristics; (2.) facilitate communication of design intent between developers in a standard way; and (3.) enable interoperability between various analysis and design tools.		
Keywords: analysis, concurrency, domain viewpoint, extensions, modeling, model processing, performance, real-time, resource, schedulability, tag value language, time, UML		
Latest / past specifications: Current version: 1.1 Past versions		
Contact Information: Realtime, Embedded and Specialized Systems PTF		
Related OMG Specifications: CORBA/IOP , Enhanced Time , UML Profile for MARTE , Real-time CORBA (Static Scheduling, Dynamic Scheduling) , Time , UML , UML Profile for QoS and FT		
OMG Cross Reference: Specialized CORBA Specifications		
Related Industry Standards:		

24/42

Can all this be used to write classes? 1/2



```

// Generated by StarUML(tm) C++ Add-In
// @ Project : Untitled
// @ File Name : Stack.cpp
// @ Date : 22/03/2010
// @ Author :

#include "Stack.h"

void Stack::init(int size) {
}

bool Stack::push(int value) {
}

bool Stack::pop(int* ptr) {
}

bool Stack::isEmpty() {
}

bool Stack::isFull() {
}

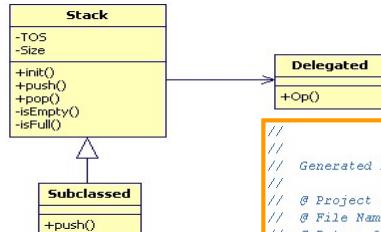
class Stack {
public:
    void init(int size);
    bool push(int value);
    bool pop(int* ptr);
private:
    int TOS;
    int Size;
    bool isEmpty();
    bool isFull();
};

#endif // _STACK_H

```

25/42

Can all this be used to write classes? 2/2



```

// Generated by StarUML(tm) C++ Add-In
// @ Project : Untitled
// @ File Name : Subclassed.h
// @ Date : 22/03/2010
// @ Author :

#ifndef _SUBCLASSED_H_
#define _SUBCLASSED_H_

#include "Stack.h"

class Subclassed : public Stack {
public:
    void push();
};

#endif // _SUBCLASSED_H_

```

Is this convenient ?

- possibly for round-trip engineering in Model Driven Development
- not for abstraction code is the best complete representation of itself
- ... not a practice for "real" SW developers
- (Model Driven Engineering is relevant, not for this aim)

26/42

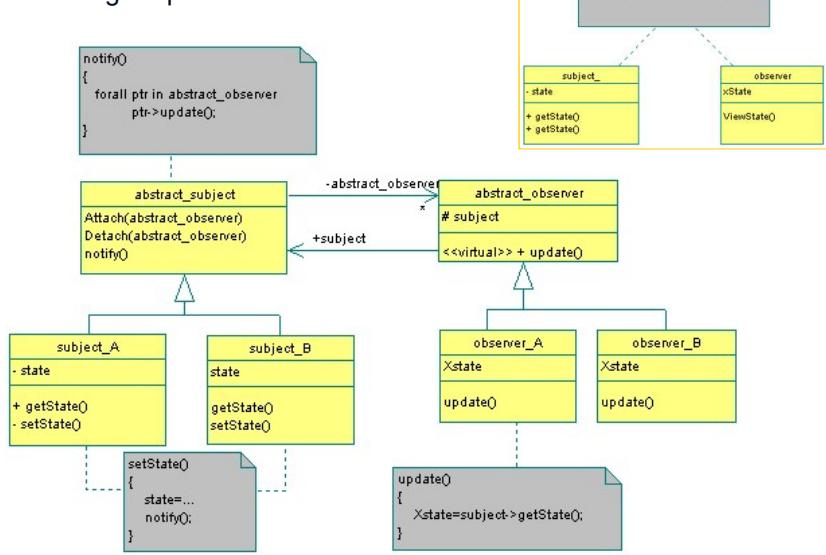
on details and abstraction perspective

- In principle an implementation model could be 1:1 wrt the code
 - Javadoc, Doxygen, ...
 - round trip engineering
- ... but in this case it would lose much of its "modeling" aim
 - hide details such as local classes, private and methods and attributes, ...
 - avoid definition of unessential or undetermined choices
- The right way: provide a “close enough” representation

27/42

on details and abstraction perspective: the Observer pattern

- Close enough representation

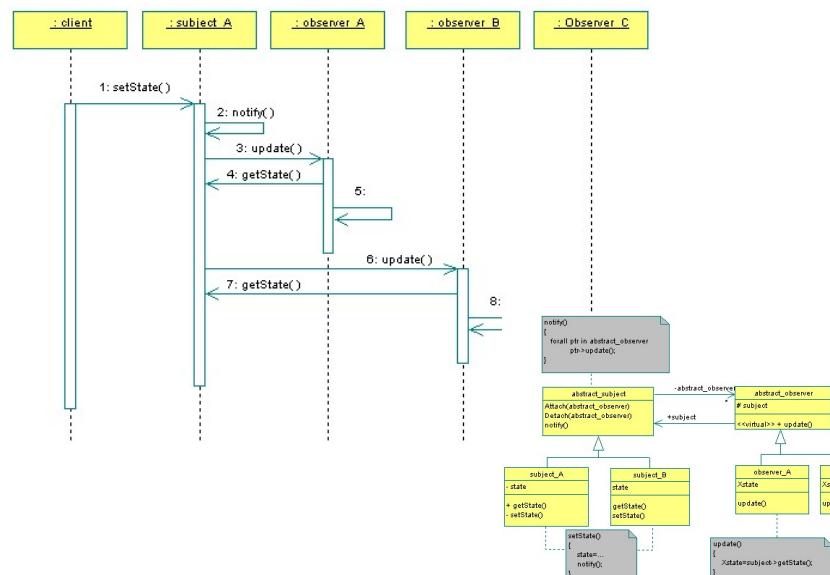


28/42

- Vista statica
 - Class diagram: tipi e relazioni tra gli oggetti nei tipi
 - Object diagram: una particolare istanza dei tipi
- Interfaccia tra il sistema e il suo ambiente
 - Use case diagrams
 - (very relevant, but discussed later for requirements analysis)
- Vista dinamica (Interaction diagrams)
 - Collaboration diagram e Sequence diagram
 - Documentano la sequenza dei messaggi scambiati tra un insieme di oggetti in uno scenario di esecuzione
 - Activity diagram: Aggiunge concetti di parallelismo e concorrenza

29/42

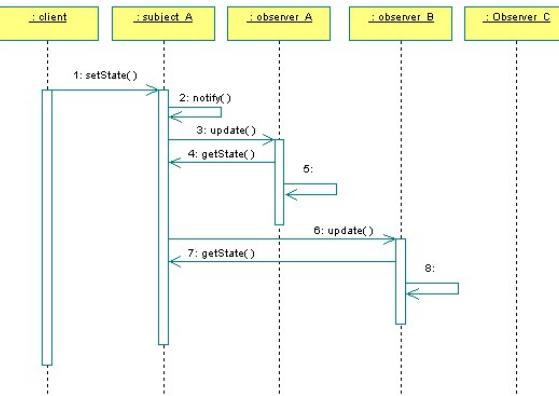
- Tipicamente per documentare la dinamica su un class diagram



30/42

Sequence diagram

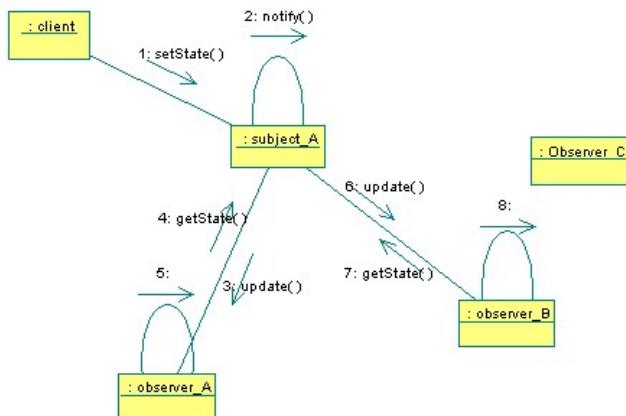
- Oggetti attivati in uno scenario di esecuzione
 - Tempo di vita e tempo di attivazione
- Messaggi scambiati
 - Creazione
 - invocazione di metodi
 - self-reference
- La direzione verticale è il tempo
- Quella orizzontale sono gli oggetti
- E' una vista dinamica
 - Utile per descrivere la dinamica associata a schemi statici
 - pericolosissimo nella progettazione, anche peggio nell'analisi
 - Applicabile a class diagrams ma anche a use cases



31/42

Collaboration diagram

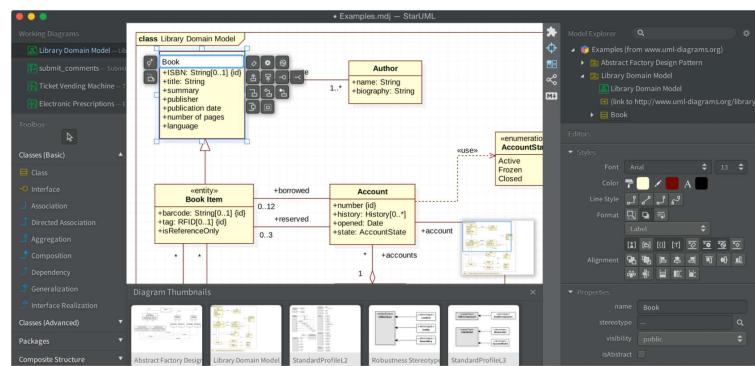
- Rappresenta l'ordine degli eventi numerandoli esplicitamente
 - numerazione strutturata: 1 ->2 ->3.1->3.2->3.3->4
- Equivalente ad un sequence diagram
 - I due tradotti in modo automatico
 - viste di uno stesso "interaction diagram"
 - Privilegia la rappresentazione dell'organizzazione degli oggetti



32/42

■ Star UML

- <https://staruml.io/>
- current (Oct.20) version: 3.2.2
- multiplatform (Windows, MacOS, Linux)
- free evaluation without time limits



33/42

3 abstraction levels

■

- class diagrams in the implementation, specification, and conceptual perspectives

■ *TBD: skip this now, and discuss it later, after design patterns and before requirements analysis*

34/42

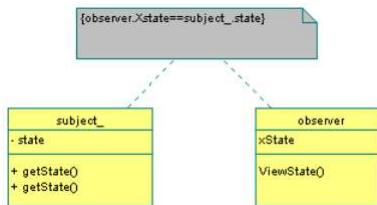
Class diagrams can serve for different abstractions and different intents

- 3 levels of abstraction (perhaps 2+1)
 - Implementation perspective
 - rappresenta il modo con cui è realizzato un sistema SW
 - Un oggetto nella rappresentazione corrisponde a un oggetto in un linguaggio OO (Java, c++, ...)
 - Specification perspective
 - Rappresenta le astrazioni che saranno realizzate in un sistema SW
 - Un oggetto della specifica è tipicamente realizzato attraverso più oggetti dell'implementazione
 - Non necessariamente viene realizzata con un linguaggio OO
 - Conceptual perspective
 - Descrive le entità di un dominio applicativo
 - Non necessariamente rappresentate in una realizzazione SW
- 2 representation intents
 - Specification: of something that shall be implemented
 - Description: of something that already exists
- Different combinations of 3x2 along the lifecycle of a SW system

35/42

Class diagrams in the specification perspective

- Classes capture abstractions implemented within an information system
 - E.g. internal and external interfaces implemented within a package, allocation of responsibilities
- May abstract from constructs (or idioms) of the specific language or framework used for the implementation
- Essential for "high-level" design
 - Enable discussion on SW architecture, at different levels of granularity
 - Document the organization of an existing component
 - Plan integration or maintenance



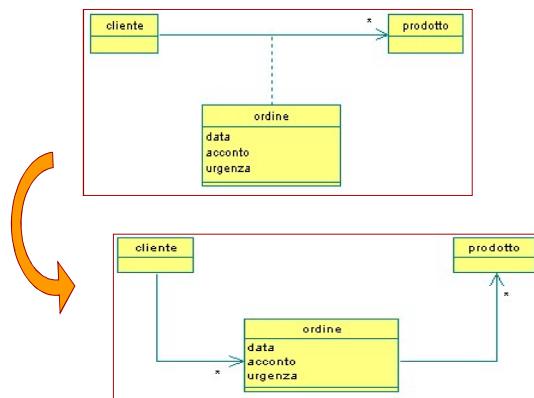
TBD: aggiungere la rappresentazione con Collaboration, e usarla per mostrare differenza tra cosa disegni nello specification e cosa nell'implementation (esempio ContoCorrente e MEF)

36/42

Some more constructs: association classes

- Association classes

- Provide attributes and also methods for associations
- Used in the specification (and conceptual) perspective to drive implementation



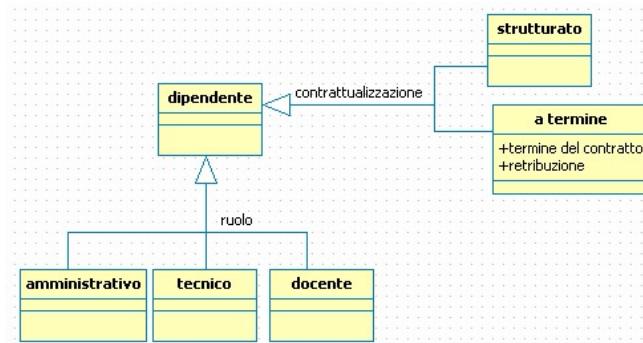
- useful also in the conceptual perspective

37/42

Some more constructs: independent generalizations

- Generalization may refer to multiple orthogonal directions, with a discriminator for each direction

- E.g. medici pediatri o di base / convenzionati o esterni
- Requires design in the step to implementation

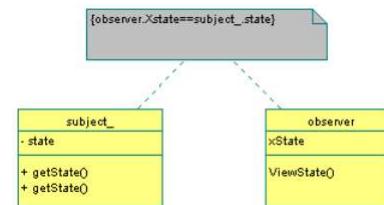


- Much useful also in the conceptual perspective

38/42

Some more constructs: constraints, comments, annotations

- Constraints
 - Within curly brackets {}
- Comments
 - Callouts boxes with free text
- Annotations
 - Supported by the editing tool

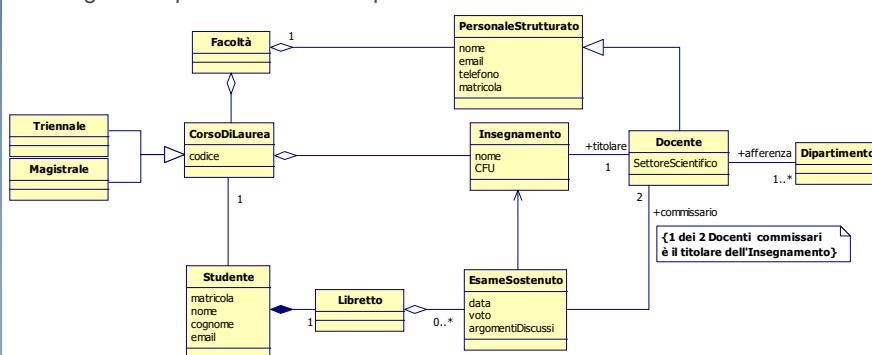


- Much useful to avoid over-specification and detail clutter
 - Also in the conceptual perspective

39/42

Class diagrams in the conceptual perspective

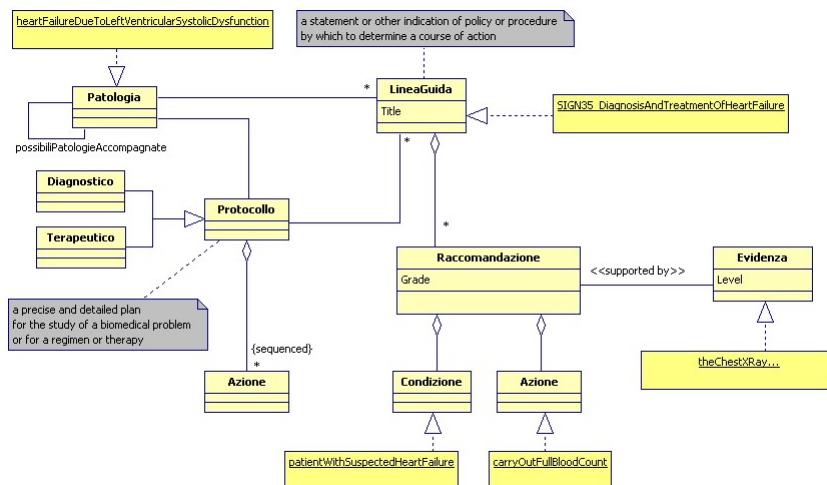
- Classes capture the concepts of some domain
 - includes concept in the *material process*
 - independently from their possible implementation within an information system
- Primarily intended for analysis and requirements definition
 - E.g. concepts involved in the process of examinations



40/42

Class diagrams in the conceptual perspective

- A class diagram in conceptual perspective: medical guidelines and protocols



41/42

Which are the right modeling artifacts to produce?

- Different artifacts
 - suited for different stages of the SW lifecycle,
 - with different responsibility for parties (developer, customer, ...)
 - Enable incremental refinement and natural transition, possibly supported by some methodology (e.g. Iconix)
- The choice depends on process characteristics
 - duration of development process, number of involved roles and people, contractual relation among parties, ...
 - Cost of documentation maintenance, reverse engineering and Model Driven Development
 - expected lifecycle of the product, duration, maintenance and evolution
- ... which in turn should depend on the characteristics of the product
 - complexity of functions, size of implementation structure, quality attributes
- Much about agility vs discipline
 - I.e. about the trade-off between lightness and robustness
 - Well represented in eXtreme Programming vs Unified Process
 - (More answers later, when dealing with SW lifecycle)

42/42

- Martin Fowler, "UML Distilled: Guida rapida al linguaggio di modellazione standard" - terza edizione, Pearson Education Italia, Febbraio 2004 .
- Martin Fowler, "UML Distilled: a brief guide to the standard object modeling language", third edition (Addison Wesley).

Software Engineering - A.A. 22/23

Design Patterns in Java
from the original [GOF] c++ version
with a certain regard for Java implementation



Enrico Vicario

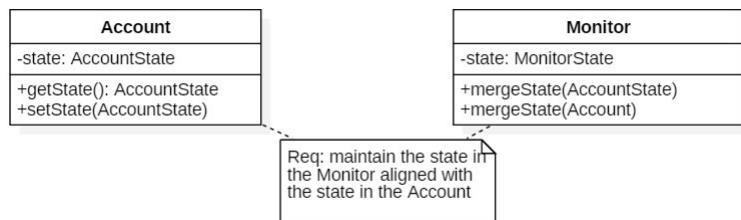
Dipartimento di Ingegneria dell'Informazione
Laboratorio Tecnologie del Software – stlab.dinfo.unifi.it
Università di Firenze

enrico.vicario@unifi.it, stlab.dinfo.unifi.it/vicario

- Design pattern: a reusable solution scheme for some recurrent problem
 - a name, a problem, alternative solutions, known consequences, ...
 - term first made explicit in Architecture by Christopher Alexander
 - like a handbook of electronics schemes (amplifiers push-pull, totem-pole, ...)
 - ProducerConsumer or Monitor are also patterns in operating systems
- Patterns in OO Design
 - Advanced use of OO programming schemes
 - Substitutability and polymorphism, inheritance, composition, ...
 - Override, method look-up or dynamic binding and virtual methods, ...
 - Basically agnostic, but differently mapped to different languages
 - ... also a way to focus features and idioms of a language in a top-down manner

■ The problem

- (an object of type) Account has a state variable whose value is relevant to (objects of type) Monitor
- this might occur in the transition from specification to implementation, or in refactoring for separation of main concerns
- much about maintaining consistency across decoupling
- (might generalize to multiple types of Account and multiple types of Monitor)



3/98

■ a naive solution

- (naive is a positive attribute, in the beginning)
- Account provides a public method `getState()` that will be invoked by Monitor to get the value of the Account state and merge it with the local Monitor state



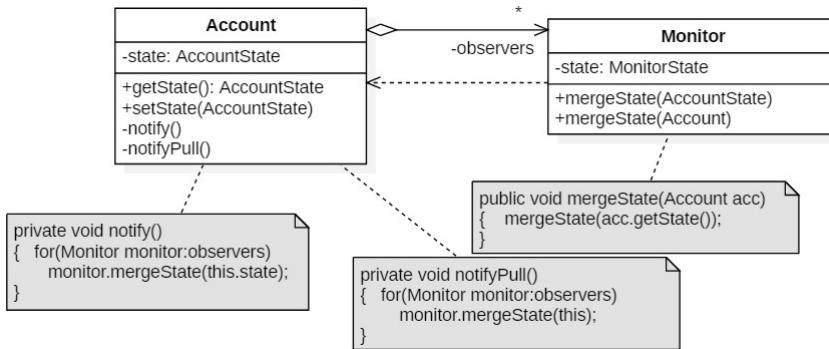
■ but, when should the Monitor update its copy of the state?

- ... polling is inefficient, and impractical
- ... might precede each use of the state
- ... but, intermediate states could be relevant as well (e.g. to compute the accumulated interest)

4/98

The Observer pattern

- assign to Account the responsibility of notifying Monitor (inversion of responsibility)
 - the method `setState()` of Account invokes `notify()` which in turn invokes `stateMerge()` on the Monitor
 - notify can be private, and can be implemented in push mode or pull mode, with `pullMode` implementing a callback on `getState()`
 - (usually, only one of the two modes is exploited)

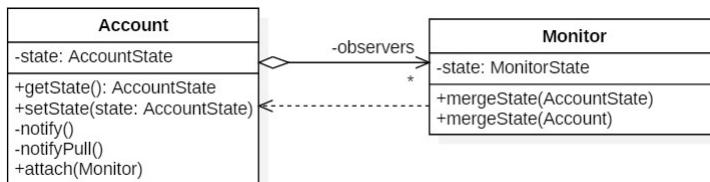


- ... yet, Account must be aware of all the types and instances of Monitor

5/98

The Observer pattern

- get Account rid of the responsibility to register the Monitor
 - (one more inversion of responsibility)
 - Account provides `attach(Monitor)`
 - ... which can be invoked by the Monitor itself (e.g. by a constructor `Monitor(Account)`), or by some other object that installs dependencies
 - `attach()` must thus be public

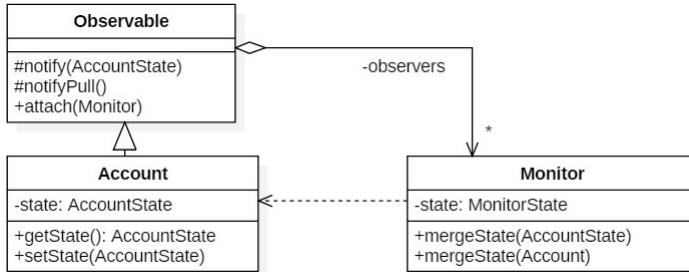


- yet, Account mixes its inherent business, with the agnostic function of notifying in the observation process

6/98

The Observer pattern

- move the responsibility of being observed away from Account
 - do this by subclassing to inherit and reuse the methods for observability
 - notify() is protected to permit invocation by setState()
 even if Account is in a different package;
 it requires a reference to the AccountState (in the push mode);
 it expresses the Account reference by «this» (in the pull mode)



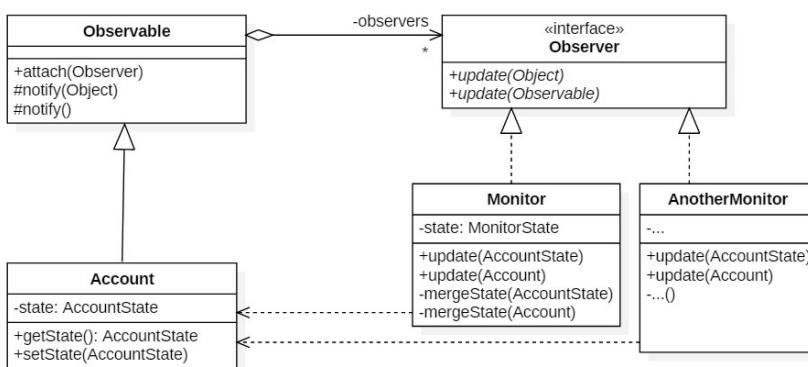
- yet, the agnostic Observable still depends on the specific type of Monitor

- TBD: what if responsibilities would be separate by composition?*

7/98

The Observer pattern

- generalize Monitor into an interface Observer
 - permit the observers collection to target different types of Observer
 - without committing to specific types of Monitor, Account, and AccountState

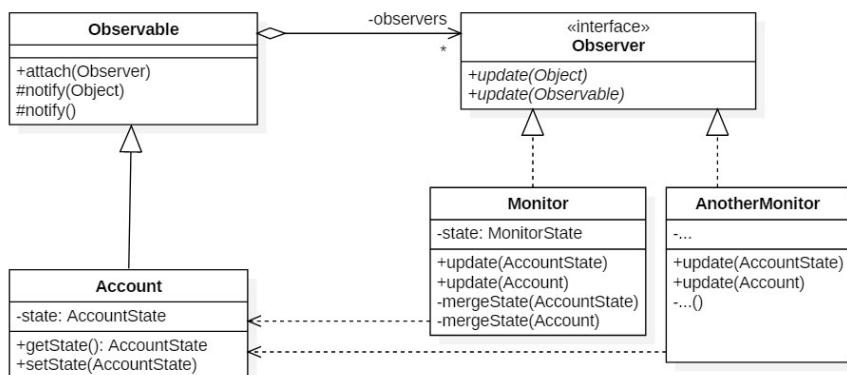


- TBD FixMe: c'e' una violazione del principio di Liskov, ... diventa necessario un downcast ... del resto, devi sapere come è fatto quello che osservi*

8/98

The Observer pattern

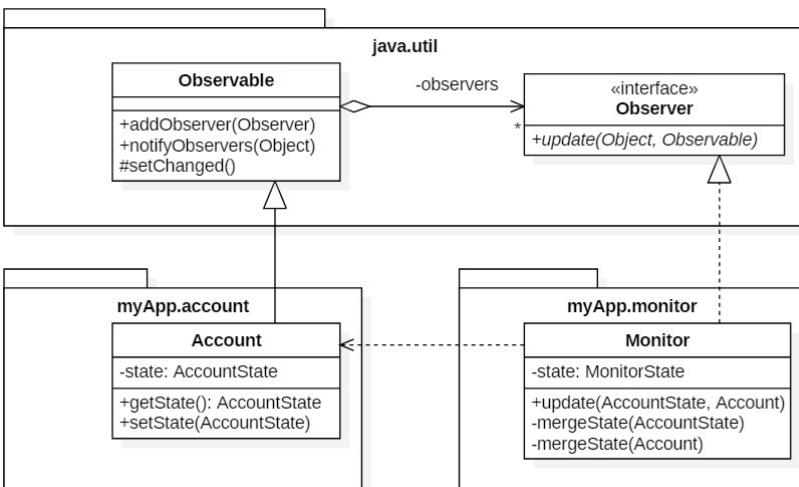
- Observable & Observer do not depend on Account or Monitor
 - implement the abstract coupling between an Observer and an Observable
 - without committing to specific types of Monitor, Account, and AccountState
- Account does not depend on Monitor
 - while the viceversa cannot be avoided



10/98

The Observer pattern

- implemented in the java.util package
 - single methods for push and pull modes, protect restriction moved to setChanged()

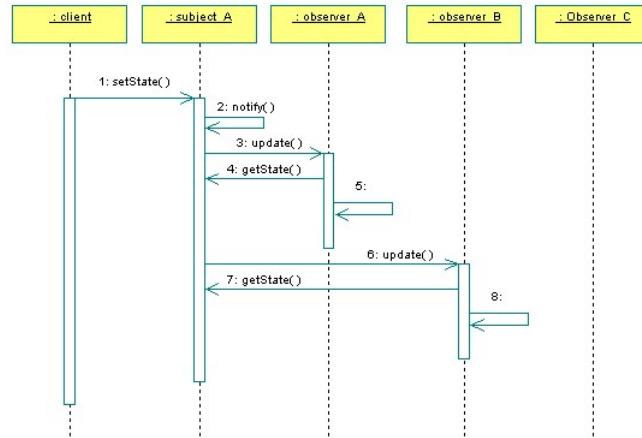


10/98

TO BE REMOVED - The Observer pattern

- Dynamic registration of Observers on the Subject

- attach() and detach() invoked by the Observer or someone else
- Subject notifies to AbstractSubject (same object) through notify()
- notify() invokes update() on each registered Observer
- Each Observer may call-back on getState() to get the state (pull vs push mode)



11/98

The Observer pattern

- Amplification of interaction :-

- In principle might even be unbounded (call-backs)
- Can be reduced by registering Observers on specific Topics, at the attach()
- usual in the enterprise scale publish&subscribe

- Pull vs push mode

- Pull: the Subject notifies that a change happened, and the Observer calls-back, if needed, according to its policy
- Push: Notification from the Subject carries the state change some parameter to distinguish among multiple subjects
- A matter of: efficiency, robustness, coupling

- State consistency

- When the update() is performed, the state must be "stable", but, some action might trigger multiple "atomic" state changes
- A change manager (Mediator pattern) may get responsibility of coordination

12/98

Observer: examples of code

TBD: show here the piece of code based on `java.util.Observable` class and `java.util.Observer` interface.
Stress the fact that the state is a primitive but it must be boxed into an Integer to pass the reference needed for the push mode.
Stress also the fact that the Observers shall be aware of the type passed in the Object reference received in the push mode notification.

TBD: A possible exercise: the state is a reference type (not a primitive int); in the push mode implementation, the subject erroneously passes the reference to the state without making a defensive copy; the observer modifies the state of the subject, though this was supposed to be private.

TBD: a thought experiment: what about separating responsibilities by composition rather than subclassing; notification trigger cannot be protected, multiple observers lists can be maintained and switched, ... more dynamic, more functional, more complex dependency installation, more error prone, more test demanding, ... as usual in the confrontation of composition versus subclassing

13/98

Patterns are often supported or applied by classes n standard APIs

■ class `java.util.Observable`

Methods	
Modifier and Type	Method and Description
void	<code>addObserver(Observer o)</code> Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	<code>clearChanged()</code> Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the <code>hasChanged</code> method will now return <code>false</code> .
int	<code>countObservers()</code> Returns the number of observers of this <code>Observable</code> object.
void	<code>deleteObserver(Observer o)</code> Deletes an observer from the set of observers of this object.
void	<code>deleteObservers()</code> Clears the observer list so that this object no longer has any observers.
boolean	<code>hasChanged()</code> Tests if this object has changed.
void	<code>notifyObservers()</code> If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
void	<code>notifyObservers(Object arg)</code> If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
protected void	<code>setChanged()</code> Marks this <code>Observable</code> object as having been changed; the <code>hasChanged</code> method will now return <code>true</code> .

■ interface `java.util.Observer`

Methods	
Modifier and Type	Method and Description
void	<code>update(Observable o, Object arg)</code> This method is called whenever the observed object is changed.

14/98

■ **notifyObservers() is public, setChanged() is protected**

- notification will not be effected unless setChanged has been invoked
- Observable and ConcreteObservables are not in the same package

■ **notifyObservers() is provided with 2 different signatures**

- No arguments for pull-mode, an Object for the State in push-mode

notifyObservers

```
public void notifyObservers()
```

If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.

Each observer has its `update` method called with two arguments: this observable object and `null`. In other words, this method is equivalent to:

```
notifyObservers(null)
```

See Also:

`clearChanged()`, `hasChanged()`, `Observer.update(java.util.Observable, java.lang.Object)`

notifyObservers

```
public void notifyObservers(Object arg)
```

If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.

Each observer has its `update` method called with two arguments: this observable object and the `arg` argument.

Parameters:

`arg` – any object.

See Also:

`clearChanged()`, `hasChanged()`, `Observer.update(java.util.Observable, java.lang.Object)`

■ **update() in java.util.Observer receives an Observable and an Object**

update

```
void update(Observable o,
```

```
Object arg)
```

This method is called whenever the observed object is changed. An application calls an Observable object's `notifyObservers` method to have all the object's observers notified of the change.

Parameters:

`o` – the observable object.

`arg` – an argument passed to the `notifyObservers` method.

15/98

■ **TBD: show more of the last documentation on JavaSE8**

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY NESTED FIELD CONSTR METHOD DETAIL FIELD CONSTR METHOD

compact1, compact2, compact3
java.util
Class Observable

java.lang.Object
java.util.Observable

public class Observable
extends Object

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface `Observer`. After an observable instance changes, an application calling the `Observable`'s `notifyObservers` method causes all of its observers to be notified of the change by a call to their `update` method.

The order in which notifications will be delivered is unspecified. The default implementation provided in the `Observable` class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose.

Note that this notification mechanism has nothing to do with threads and is completely separate from the `wait` and `notify` mechanism of class `Object`.

When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the `equals` method returns true for them.

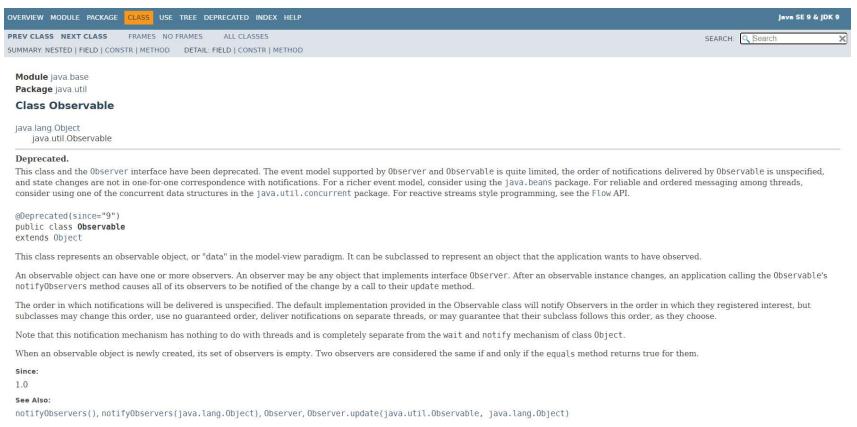
Since:
JDK1.0

See Also:
`notifyObservers()`, `notifyObservers(java.lang.Object)`, `Observer`, `Observer.update(java.util.Observable, java.lang.Object)`

16/133

Enrico Vicario - AA 22/23
SW Engineering

- TBD: discuss deprecation since JavaSE9
<https://docs.oracle.com/javase/9/docs/api/java/util/Observer.html>



17/133

Enrico Vicario - AA 22/23
SW Engineering

Design Patterns [GOF]

- A system of 23 Design Patterns
 - [\[GOF95\]](#) Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns, elements of reusable Object-Oriented software"

***■ In the next few slides:
emphasis on how a pattern is described,
not on the pattern itself***

18/98

description of a pattern (Observer) : WHEN

■ Intent

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

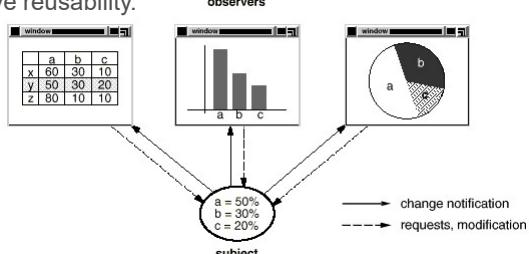
■ Also known as

- Publish & subscribe, Dependents, (*Listener*)

■ Motivation:

- <a situation where the problem may occur>
- A common side-effect of partitioning is the need to maintain consistency between related objects, without making the classes tightly coupled, so as because that preserve reusability.

- A User Interface based on the Model View Controller



■ Applicability

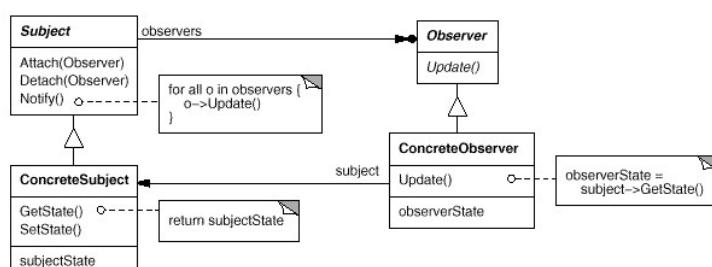
- When a change to one object requires changing others, and you don't know how many objects need to be changed.

19/98

description of a pattern (Observer) : HOW 1/3

■ Structure

- Described through a class diagram (in a partially superseded notation)
- Close enough, C++ biased, but basically agnostic



20/98

■ Participants

- Subject: knows its observers; any number of Observer objects may observe a subject; provides an interface for attaching and detaching Observer objects.
- Observer: defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject: stores state of interest to ConcreteObserver objects; sends a notification to its observers when its state changes.
- ConcreteObserver: maintains a reference to a ConcreteSubject object; stores state that should stay consistent with the subject's; implements the Observer updating interface to keep its state consistent with the subject's.

■ Collaborations

- ConcreteSubject notifies its observers whenever a change occurs
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information.
- ConcreteObserver uses this information to reconcile its state with that of the subject.

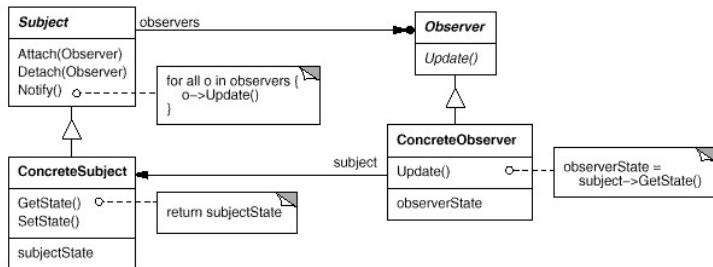
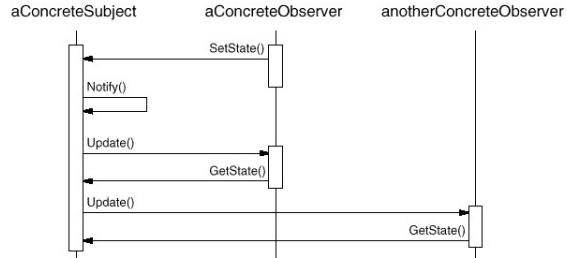
Class	Collaborators
Model	<ul style="list-style-type: none">• View• Controller
Responsibility <ul style="list-style-type: none">• Provides functional core of the application.• Registers dependent views and controllers.• Notifies dependent components about data changes.	

Class	Collaborators
View	<ul style="list-style-type: none">• Controller• Model
Responsibility <ul style="list-style-type: none">• Creates and initializes its associated controller.• Displays information to the user.• Implements the update procedure.• Retrieves data from the model.	

Class	Collaborators
Controller	<ul style="list-style-type: none">• View• Model
Responsibility <ul style="list-style-type: none">• Accepts user input as events.• Translates events to service requests for the model or display requests for the view.• Implements the update procedure, if required.	

description of a pattern (Observer) : HOW 3/3

- collaborations often documented through a sequence diagram



123/98

description of a pattern (Observer) : CONSEQUENCES

- Consequences

- <Good and bad news>
- As main consequences you can:
vary subjects and observers independently;
reuse subjects without reusing their observers, and vice versa;
add observers without modifying the subject or other observers.
- Abstract coupling between Subject and Observer :
Subject doesn't know the concrete class of any Observer
(but not between their concrete implementations)
- Support for broadcast communication.
- Unexpected updates:
since Observers do not know of each other,
unexpected updates may occur :-)

24/98

description of a pattern (Observer) : IMPLEMENTATION

- **Implementation**
 - *Observing more than one subject:*
an Observer may depend on many Subjects;
in this case, extend the update() interface to let the Observer know
which Subject is sending the notification (a parameter, or a self reference).
 - *Who triggers the update?*
Let state-setting methods of the Subject call notify():
simple, but possibly expensive with multiple consecutive changes;
Or else, make clients responsible for calling notify() at the right time:
is more efficient and flexible, but puts responsibility over (unknown) clients
 - *Dangling references to deleted Subjects*
 - *Make sure Subject state is self-consistent before notification:*
unexpected notification may occur
(e.g. a Subject subclass call inherited operations);
document which Subject methods invoke notify()
 - *Update protocols:*
push/pull notification;
 - *Specifying modifications of interest explicitly:*
specify topics/aspects of interest to limit interaction and update
- **Sample Code**
 - <pieces of code, in c++>

25/98

description of a pattern (Observer): MORE

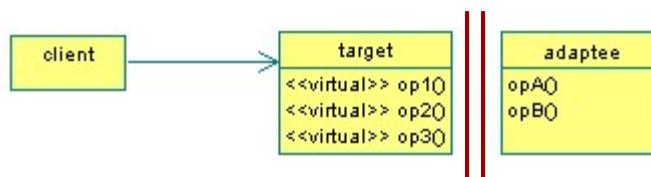
- **Known uses**
 - Model view controller
 - <often coming from some kind of superior yet past world>
- **Related Patterns**
 - Mediator: between Subjects and Observers, to encapsulate complex update
 - Singleton: to ensure single instance of the Mediator

26/98

- [GOF] Design Patterns refer to C++
 - but, patterns are by construction somehow language agnostic:
no impact on WHEN, minor impact on STRUCTURE,
some impact on IMPLEMENTATION
- What changes from C++ to Java ?
 - <intentionally incomplete>
 - Method lookup vs dynamic binding and virtual methods
 - single inheritance, interfaces, and abstract classes,
 - Protected and package-private visibility
 - Garbage collection
 - Nested classes
 - Concrete support from standard API
 - <Often an opportunity more than a problem to fix>
- Some references to work with
 - J.W.Cooper, "The design patterns java companion"
 - F.Guidi Polanco "GoF's Design Patterns in Java"
 - J.Bloch, "Effective Java"
 - ... GOF, "Design Patterns, elements of reusable OO software"

27/98

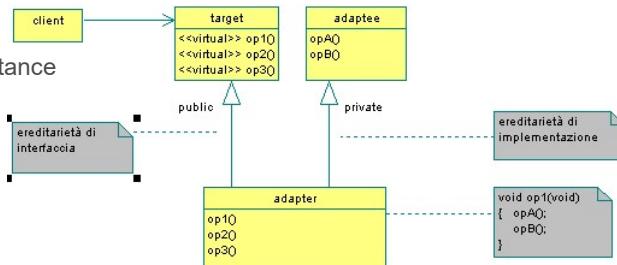
- Intent
 - Convert the interface of a class into another interface that clients expect
- Also Known as
 - Wrapper
- Motivation
 - an already implemented Adaptee provides operations needed by some new client
 - ... but, the operations of Adaptee do not match the expected interface (e.g. for some mismatch on names, signature, aggregation of operations, ...)



28/98

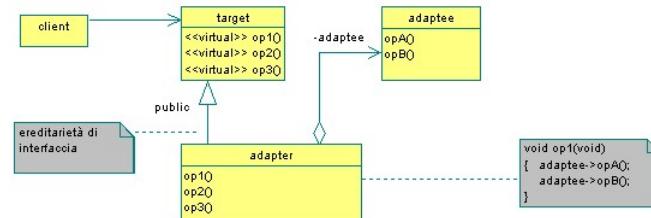
■ Class adapter

- Based on inheritance



■ Object adapter

- Based on composition

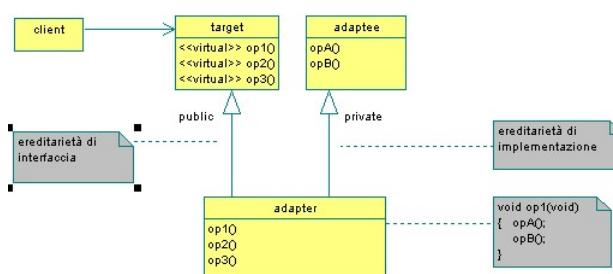


▪ TBD: revise UML diagrams: naming style, interface implementation

29/98

■

- Implements the interface of target by extending the adaptee

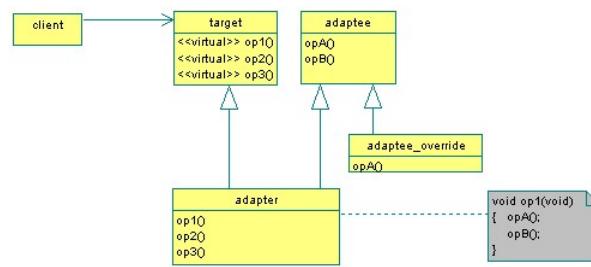


30/98

Class adapter

- A single object instance :)

- simple to create, and to debug



- Adapter does not inherit an override of the adaptee (method lookup)

- ... which might be not bad news, to avoid fragility due to unknown override
- The override can be done anyway in the adapter

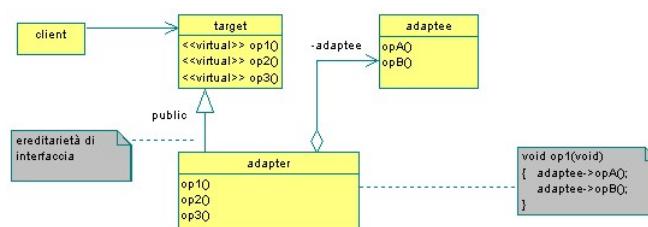
▪ *TBD: single inheritance!*

▪ *TBD: discutere di class vs object adapter e circa subclassing vs composition*

31/98

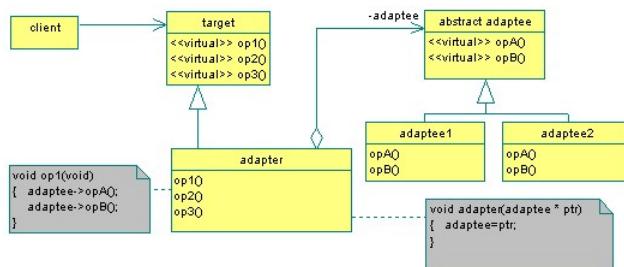
Object adapter

- The adapter Implements the target interface by composition of the adaptee (forwarding)



32/98

- Different implementations of the adaptee can be composed :)
- the adaptee can be an interface or an abstract class
- The constructor of the adapter shall install the specific concrete adaptee



- Two objects will be instantiated and composed :-)
- In principle, the adaptee could be replaced dynamically :-)
 - viceversa, this might be a feature to avoid, through an immutable field
- The adaptee is not aware to be part of a composition (self problem)
- In the end, this is the basic trade-off between inheritance and composition

33/98

- *TBD: Exercise: write an adapter in the object and class scheme, showing how the object adapter scheme is sensitive to subclassing of the adaptee*

34/98

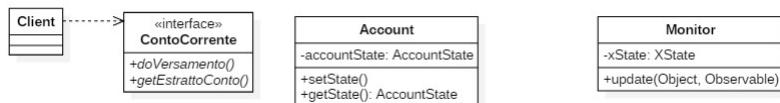
An exercise: adapt to observe

- (Esame 11 Lug.2016 – 20/30 punti – 40 minuti)
- Si consideri una combinazione dei patterns *Object Adapter* e *Observer*:
 - un Client usa un'interfaccia Target (in riferimento al pattern Adapter) che è implementata da una classe Adapter la quale compone (i.e. wraps, delegates to) un Adaptee.
 - l'oggetto che svolge il ruolo di Adaptee nel pattern Adapter svolge anche il ruolo di Observer (in riferimento al pattern Observer) potendosi registrare su un altro oggetto di tipo Subject (in riferimento al pattern Adapter).
- Si produca un modello delle classi UML
 - in prospettiva di implementazione,
 - evidenziando il ruolo delle classi nei due patterns realizzati (7 punti).
- Si producano frammenti di codice significativi
 - per illustrare l'interazione tra gli oggetti (7 punti), tra cui
 - il codice con cui viene creato l'oggetto Adapter, viene in esso composto un Adaptee, e l'Adaptee si registra sul Subject;
 - il codice con cui l'Adaptee riceve notifica a seguito di un cambiamento di stato del Subject.

35/98

TBD: An exercise: adapt to observe

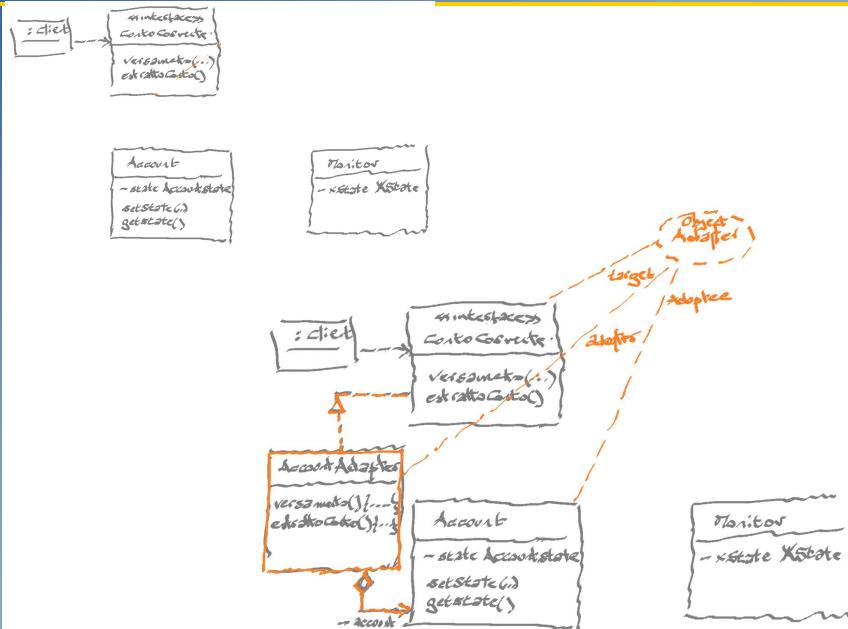
- TBD (Esame 11 Lug.2016 – 20/30 punti – 40 minuti)
portato in un contesto funzionale e un po' complicato (22/10/2021)
- Si consideri una combinazione dei patterns *Object Adapter* e *Observer*:
 - una classe Account implementa la logica di un conto corrente
 - un Client deve usare la capacità di Account attraverso una diversa interfaccia
 - un Monitor deve potere osservare lo stato di Account
- Si produca un modello di implementazione delle classi in UML
 - evidenziando il ruolo delle classi nei due patterns realizzati
- Si producano frammenti di codice significativi
 - per illustrare l'interazione tra gli oggetti, tra cui
 - il codice con cui viene creato l'oggetto Adapter, viene in esso composto un Adaptee, e l'Adaptee si registra sul Subject;
 - il codice con cui l'Adaptee riceve notifica a seguito di un cambiamento di stato del Subject
- Si valuti come evitare qualsiasi modifica in Account e in Monitor



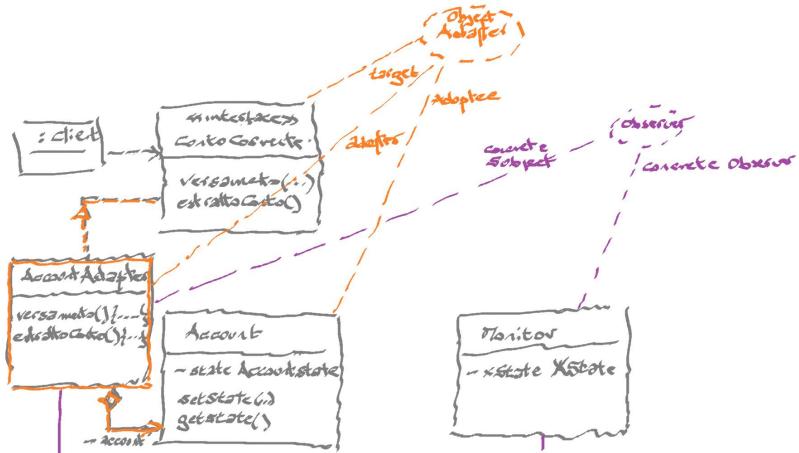
36/98

- In relazione allo schema, si illustri il cosiddetto "self problem" nell'uso del pattern Object Adapter
 - ("Because a wrapped object doesn't know of its wrapper, it passes a reference to itself (this) and callbacks elude the wrapper [Lieberman86]"")
 - definendo uno scenario di test (6), realizzato per semplicità nella forma di un main(), che produce la seguente sequenza di eventi:
 - l'Adaptee si registra sul Subject con il proprio riferimento
 - quando il Subject modifica il proprio stato, la notifica è consegnata all'Adaptee e non all'Adapter

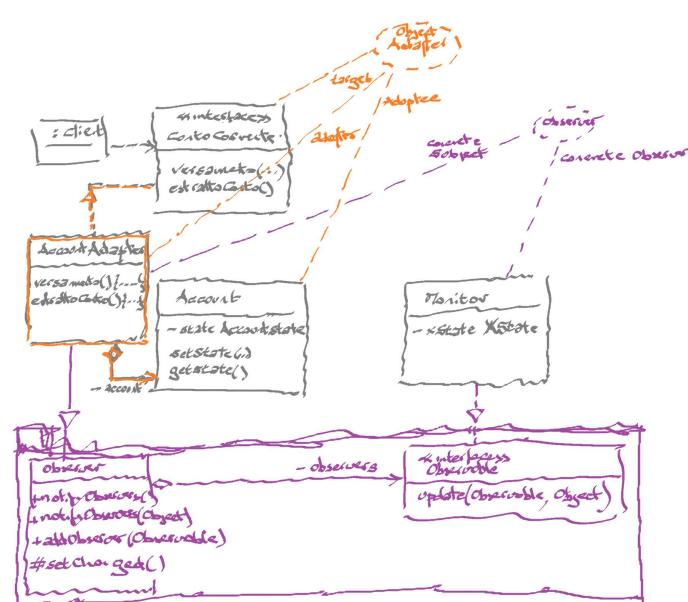
37/98



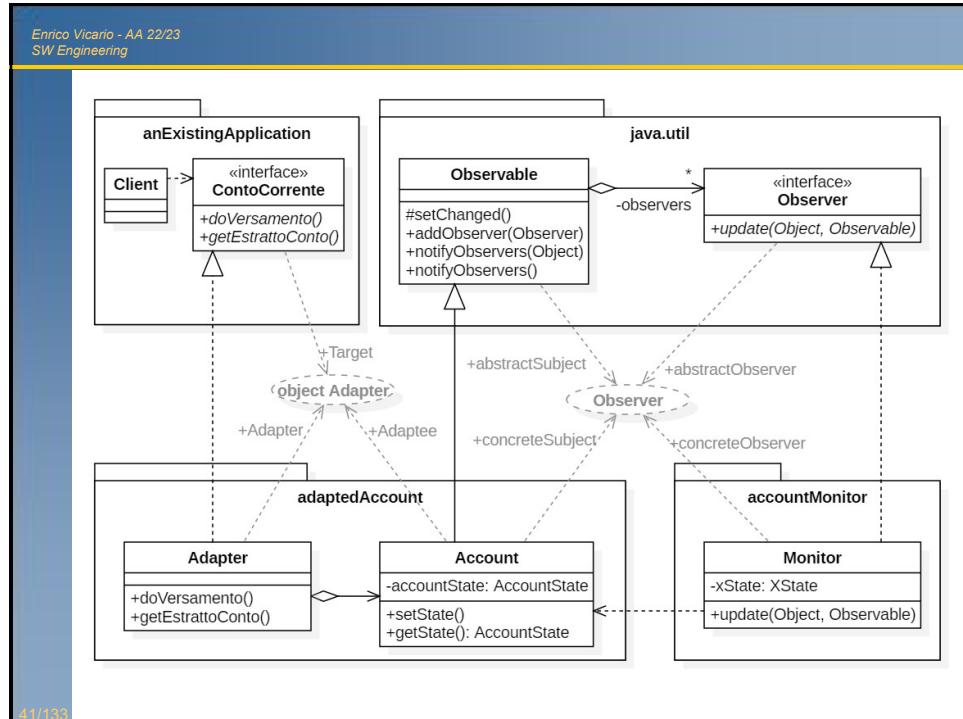
38/133



39/133



40/133



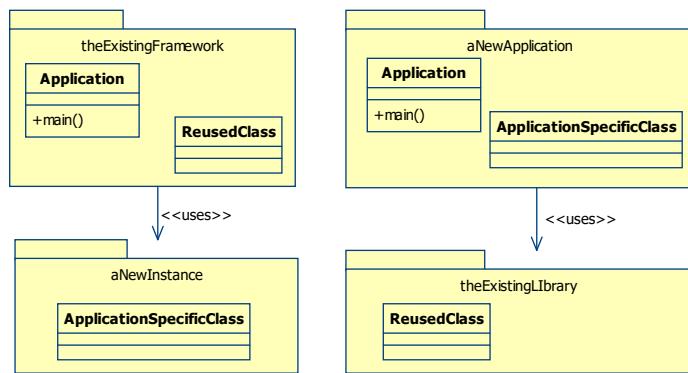
Enrico Vicario - AA 22/23
SW Engineering

A short aside on "frameworks" and "libraries" - 1/2

- A library
 - a set of functions that you can call, these days usually organized into classes.
Each call does some work and returns control to the client.
- A framework
 - embodies some abstract design, with more behavior built in.
To use it you insert your behavior into various places in the framework either by subclassing or by plugging in your own classes.
The framework's code then calls your code at these points.
 - "One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code.
The framework often plays the role of the main program in coordinating and sequencing application activity.
This inversion of control gives frameworks the power to serve as extensible skeletons.
The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application." (Ralph Johnson and Brian Foote)
 - <http://martinfowler.com/bliki/InversionOfControl.html>

42/98

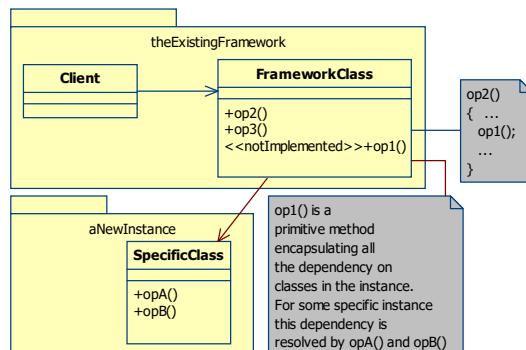
- The shift from libraries to frameworks is much about inversion of control in who-calls-whom vs who-reuses-whom
 - new code invokes reused objects of a library
 - new code is invoked by the objects in a reused framework
 - (inversion of control, Hollywood principle, ...)



43/98

- Consider a framework providing an abstract implementation for responsibilities applicable to a variety of types
 - E.g. wrap a payload within an envelope, send the envelope, ... envelope of what?
- The framework may prepare installation of an adapter to facilitate the concrete instantiation
 - Note that here the adapter is designed in advance, not as a retrofit
- Two schemes
 - Abstract operations (*here skipped*)
 - Delegate objects

44/98



SKIP - Pluggable adapters: delegate objects - 1/2

- **Delegate**

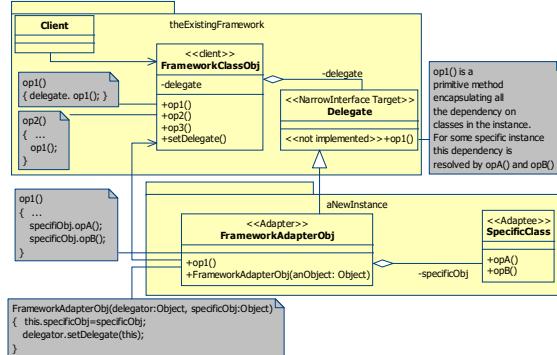
- a narrow interface with few "primitive" methods covering context dependence

- **FrameworkClassObj**

- Delegates context dependent operations to Delegate
- Exposes a method setDelegate() enabling installation of the Delegate

- **FrameworkAdapterObj**

- Implements Delegate using SpecificClass
- Installs itself in FrameworkClassObj as delegate (e.g. in the constructor)



45/98

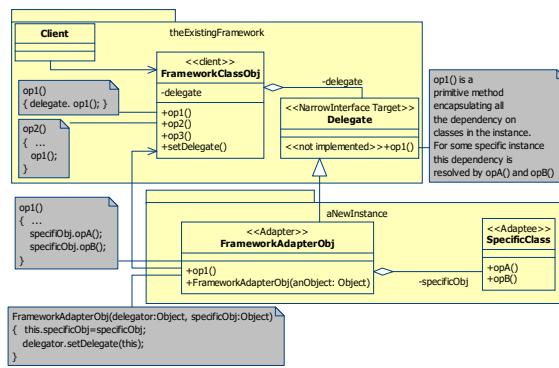
SKIP - Pluggable adapters: delegate objects - 2/2

- *TBD: how does the FrameworkAdapterObj get the reference to the FrameworkClassObj?*

- (static items, Singleton, a container implementing Dependency Injection ...)

- *TBD: who shall implement the FrameworkAdapterObj?*

- Is this a skeleton provided with the Framework as a reference implementation?



46/98

- TBD: exercise: write here the code of a pluggable adapter in the delegate object scheme.

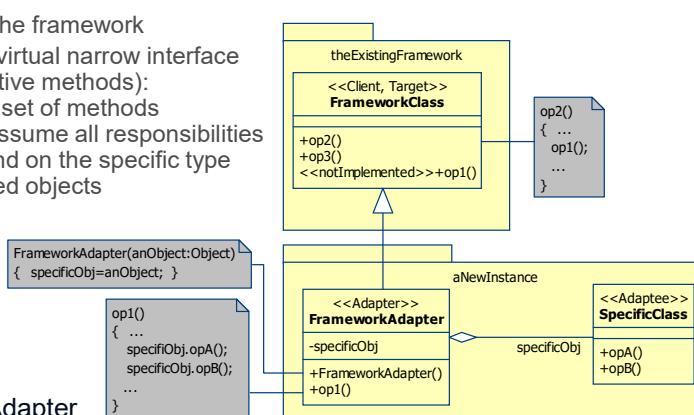
▪ TBD: As a subcase, distinguish different ways how the object in the instance may recover the reference to the object where it shall install itself (e.g. a static method, but in this case also the reference shall be static; or, a singleton exposed by the framework that provides needed references where dependencies shall be installed, ...)

- TBD: add some content on the context dependency injection pattern CDI, and some reference to containers.

- TBD: is this really a different scheme? Is there enough conceptual space for distinguishing two kinds of pluggable adapter?

FrameworkClass

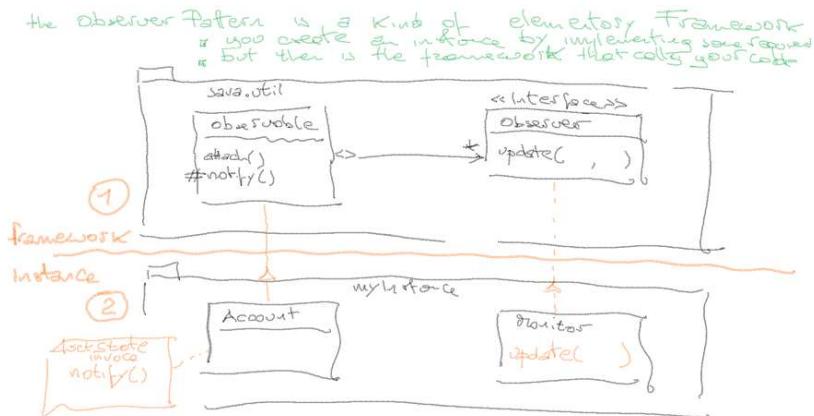
- Is part of the framework
- defines a virtual narrow interface (aka primitive methods): a minimal set of methods that can assume all responsibilities that depend on the specific type of managed objects



FrameworkAdapter

- invokes concrete methods in the specific domain
- ... to implement operations in the narrow interface

TBD: the Observer pattern is a kind of Framework



49/133

The Factory Pattern

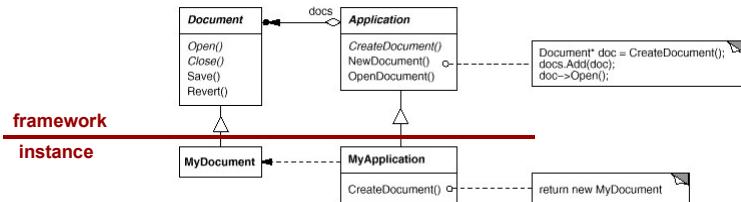
Intent

- define an interface for creating an object,
 but let subclasses decide which class to instantiate.

Motivation

- Frameworks use abstract classes to define and maintain relationships between objects.
- E.g. implement the lifecycle of a document, applicable to any type of document:
 the framework implements the concept that an Application manages a collection of Documents that can be opened, processed, saved, closed;
 a specific Application (instance) will determine the specific structure of Document, and the way how operations are concretely performed

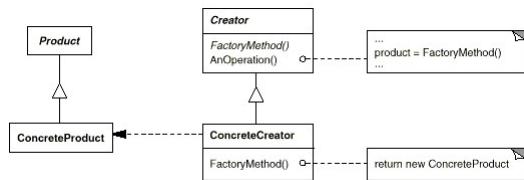
TBD: an electronic health record manages observations,... of what type?



50/98

■ Structure (and Participants and Collaborations)

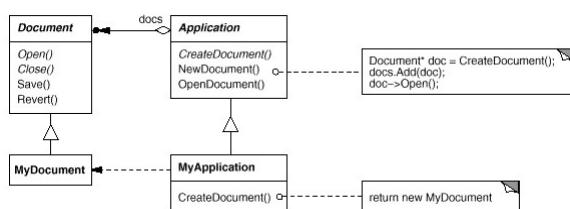
- Creator is an abstract class:
anOperation() is implemented by delegating to a not-implemented factoryMethod() the creation of a Product
- Product is an interface
- ConcreteCreator implements the factoryMethod(), determining the specific Product



51/98

■ Consequences

- *Main*: eliminate the need to bind application-specific classes into your code, and thus make your code (framework) reusable
- *Provides hooks for subclasses*:
Document could *define* `createFileDialog()` that creates a default file dialog object for opening an existing document;
a subclass of Document defines an application-specific override;
(in this case the factory method is not abstract and provides a reasonable default)



52/98

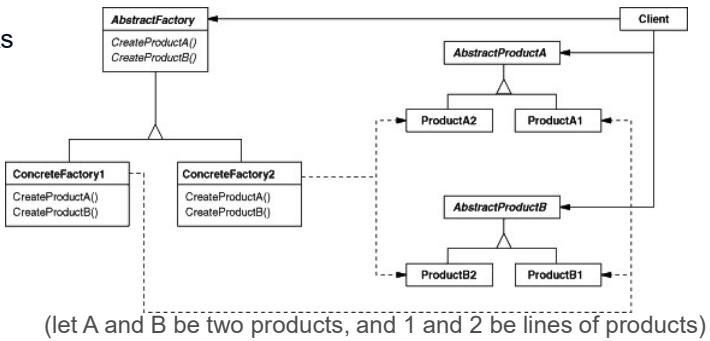
The Abstract Factory pattern

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also Known As

- Kit



(let A and B be two products, and 1 and 2 be lines of products)

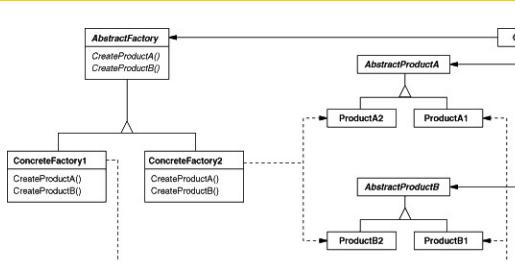
- Client delegates to the **AbstractFactory** the creation of Products
- The choice of the concrete implementation for **AbstractFactory** determines the line in which Products will be instantiated
- (in Java, **AbstractFactory** is an interface better than an abstract class)

53/98

Abstract factory

Consequences

- Localizes the choice of the line of Products :-)
- rende agile la modifica della famiglia di prodotti :-)
- Guarantees consistence of the line of Products :-)
- Hurdles the addition of a new type of Product :-)



Who chooses the concrete implementation for the Abstract Factory?

- The client itself
- A third object that sets up the all scheme

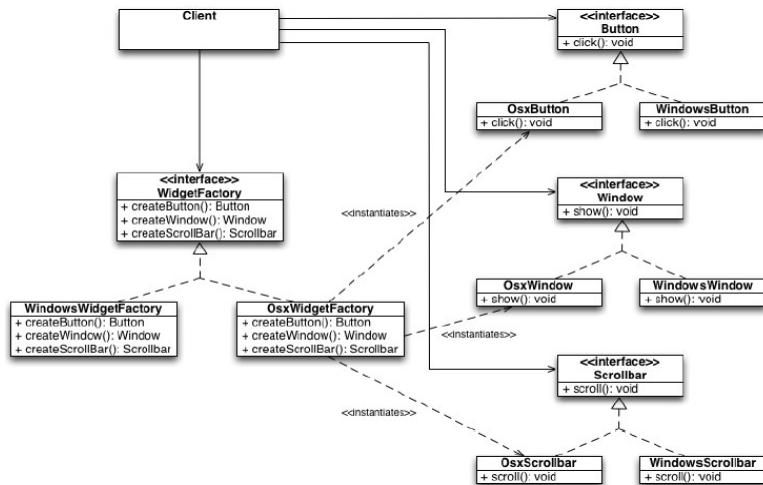
The Factory holds no state and can be implemented as a Singleton

54/98

Abstract factory: an example

■ Construction of a GUI

- Made of buttons, Windows, Scrollbars, ... running under Windows or Osx



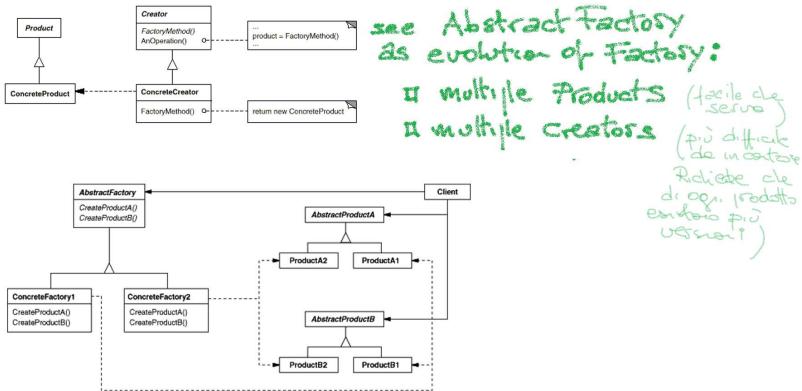
55/98

TBD

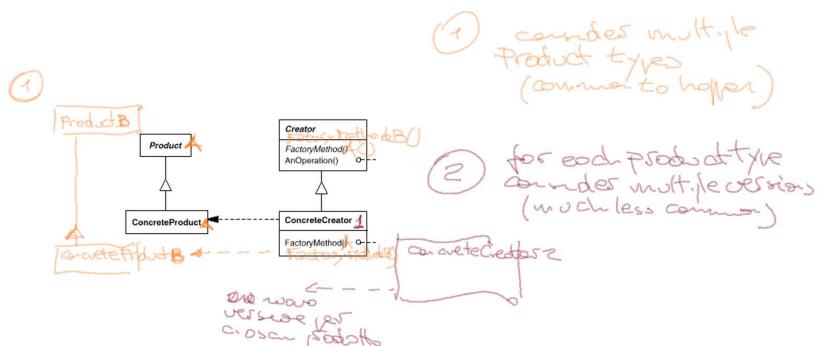
- TBD exercise: build an abstract factory that is implemented as a singleton.*

56/98

TBD: Abstract Factory as an evolution of Factory



57/133



58/133

Classification of patterns: 3 types of purpose

- **creational patterns:**
 - E.g. the **Abstract Factory** Provides an interface for creating families of related or dependent objects without specifying their concrete classes
 - Abstract about the process of object creation:
encapsulate knowledge about which concrete classes the system uses, and hide how instances of these classes are created and put together; to provide flexibility in *what* gets created, by *whom*, *how* and *when*.
 - <*no counterpart for managing destruction? A good reason for Java*>
- **structural patterns:**
 - E.g. the **Adapter**: converts the interface of a class into another interface that some client expects.
 - about how classes and objects are composed to form larger structures:
class patterns use inheritance to compose interfaces or implementations;
object patterns compose objects to realize new functionality.
- **behavioral patterns:**
 - E.g. the **Observer**: defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified
 - about abstracting on the control-flow of operations in different objects, to shift the focus from control-flow to the way objects are interconnected

59/98

Language idioms and design patterns

- **Idiom**
 - An elementary scheme, with direct mapping on some language construct
 - An effective way to use some language construct
 - In C, passing the address value; in Java, using a static factory method, ...
 - "Effective Java" by J. Bloch is much about idioms for Java
- **Design pattern**
 - Still a scheme, of larger size, usually involving multiple classes
 - More language agnostic, using idioms in the concrete implementation (Opposite view: a way to reproduce something that the language misses)
 - "Design Patterns" by GOF: C++ oriented, but applicable on Java as well
- **Both are about reusing solutions**
 - Characterizing recurrent problems, alternative choices, consequences, ...
- **Incrementally involved in the incremental transition**
 - from a specification model to an implementation (design patterns)
 - ... and then to the concrete implementation (idioms)
 - (Tutorial view: Java language -> Java Idioms -> Design Patterns (in Java))

60/98

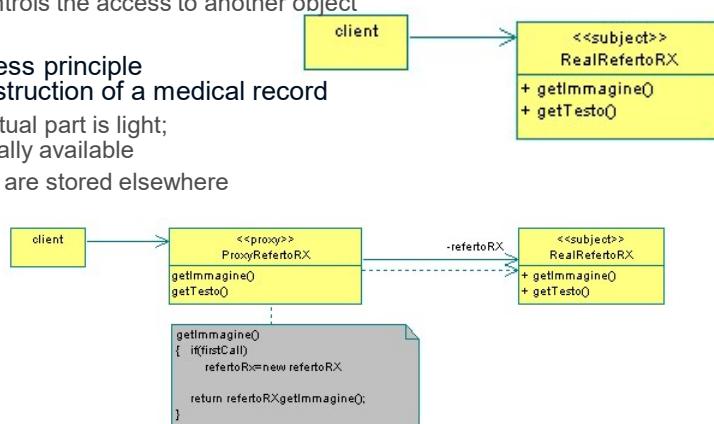
Proxy - (structural)

Intent

- Creates a surrogate or placeholder that controls the access to another object

E.g. laziness principle in the construction of a medical record

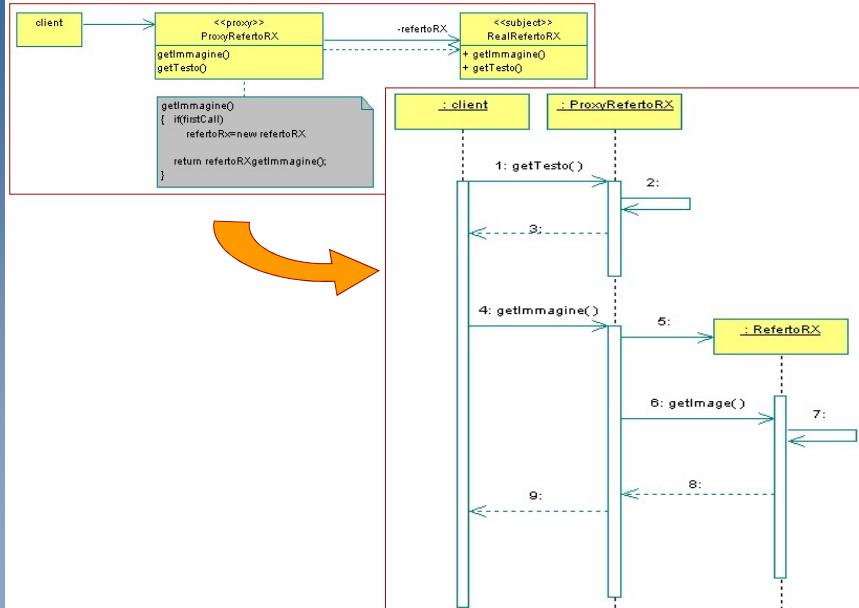
- The textual part is light; and locally available
- Images are stored elsewhere



... also in displaying a resultset or a complex page

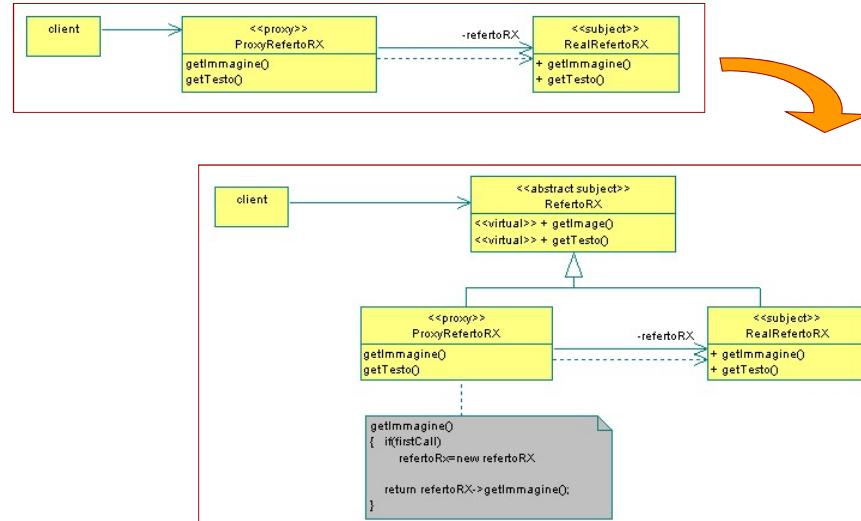
61/98

Proxy



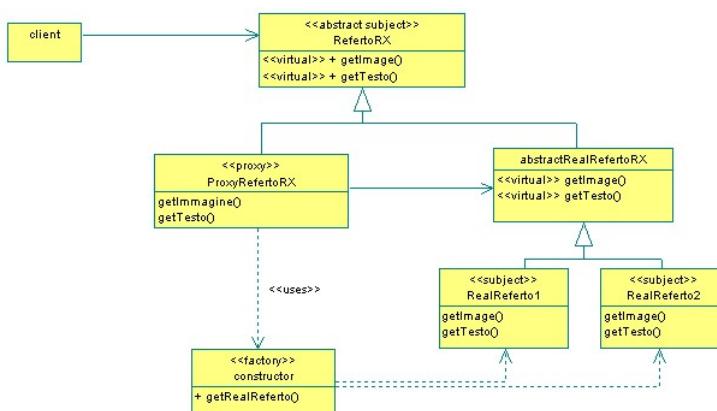
62/98

- Make the client unaware of proxy intermediation



63/98

- Make the proxy unaware of the specific type of the real object
 - Localize the choice at the time of installation



64/98

An exercise: Proxy on an Observer

- (Esame 5 aprile 2018 – 21/30 punti – 45 minuti)
- Si consideri uno scenario che combina i patterns *Observer* e *Proxy*:
 - un oggetto di tipo Monitor deve tenere traccia delle variazioni di stato di un oggetto di tipo Component secondo un meccanismo implementato attraverso il design pattern Observer;
 - la creazione del Monitor è onerosa, e viene quindi rimandata fino al momento della prima notifica, secondo un meccanismo di lazy load implementato attraverso il design pattern Proxy;
 - un oggetto di tipo SmartMonitor si registra come Observer del Component, e al momento in cui riceve la prima notifica provvede a mettere in vita il Monitor e a passargli le notifiche in forwarding.
- Si descriva la struttura con cui
 - vengono composti i due patterns, il Monitor e un Client che opera nel ruolo di test driver (il main) attraverso l'uso di un class diagram
- si dettaglino frammenti di codice che caratterizzano le implementazioni
- si definisca il codice di un metodo main
 - che testa l'implementazione in uno scenario caratterizzante.
- si discutano le conseguenze
 - della implementazione dell'Observer in modo push o pull, evidenziando nei due casi la diversa struttura della signature dei metodi di notifica e update

165/98

An exercise: Proxy on an Observer

-
- ```

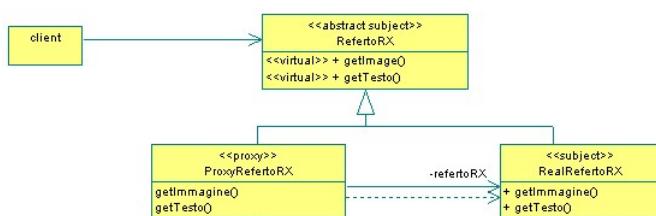
classDiagram
 Observable <|-- Component
 Observable <|-- SmartMonitor
 Observable <|-- Monitor
 Observable "1..*-->" Observer
 Component --> Subject
 Component --> Observer
 SmartMonitor --> Subject
 SmartMonitor --> Observer
 Monitor --> AbstractMonitor
 SmartMonitor --> creates --> Monitor

```
- Remark: might the «real» objects require adaptation?
    - Component must be implemented to be an Observable
    - Monitor is less aware, but must declare implements and expose setState()
  - What's AbstractMonitor for? no client is using it,
    - ... and Monitor would not receive notifications for update

166/98

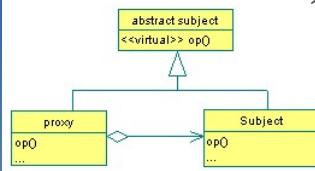
## Chaining Proxy - chaining

- Proxies can serve for various purposes
  - Virtual proxy: virtualizes operations on some object (e.g. lazy load)
  - Remote proxy: a local placeholder for some remote object
  - Protection proxy: e.g. for authentication, authorization, access control
  - Smart reference: Replaces a reference and adds behavior on the invocation (e.g. count or limit the number of accesses)
- ... but it is not possible chaining of multiple proxies layering specialized responsibilities
  - a Proxy can refer to a Subject, but not to another Proxy

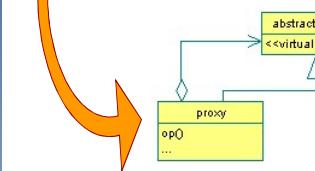


## Evolving the Proxy structure to enable chaining

1. Initial structure: The proxy is associated to a Subject



2. The proxy can also be associated to another proxy



3. proxies can have different types



Chaining is now allowed

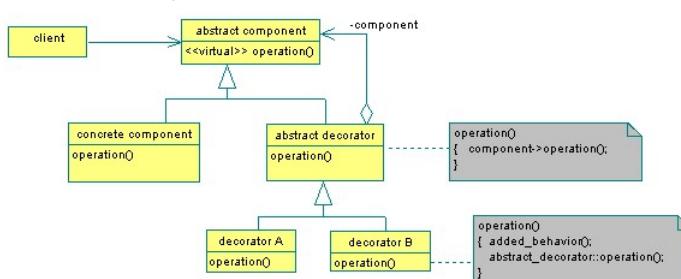


69/98

## Decorator - (behavioral)

### Call this a Decorator

- Evolution of a Proxy towards repetitive composition



### What can this serve for ?

- Dynamically add behavior and responsibilities to an operation (not to add a new operation)

### patterns are discovered rather than invented, they emerge through evolution

- ... much more to say on overdesign and patterns emerging during refactoring

70/98

### Decorator from a case example

- TBD: replace this example with the case of the description of a course: the course carries a name and an acronym; it can be decorated with information on the curriculum (curriculum name (or reference), year, term, CFU), on the contents (reference to a Syllabus), on the timetable .... A function `showDescription` is common to all these abstractions

- Il tipo cittadino è caratterizzato da attributi diversi in diversi contesti
  - Anagrafe, sanità, lavoro, ...
  - ... sono aspetti

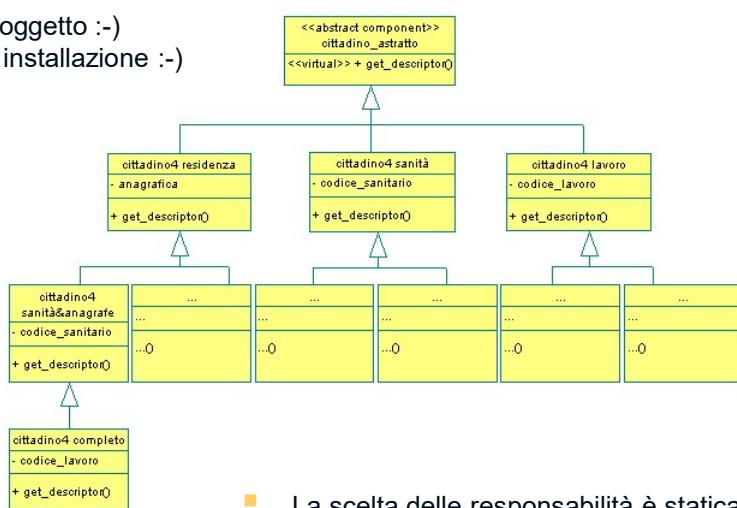


- In ciascun contesto è implementata, in modo diverso, l'operazione di costruzione del descrittore

71/98

### Decorator from a case example - solution by inheritance

- Un solo oggetto :-)
- Di facile installazione :-)

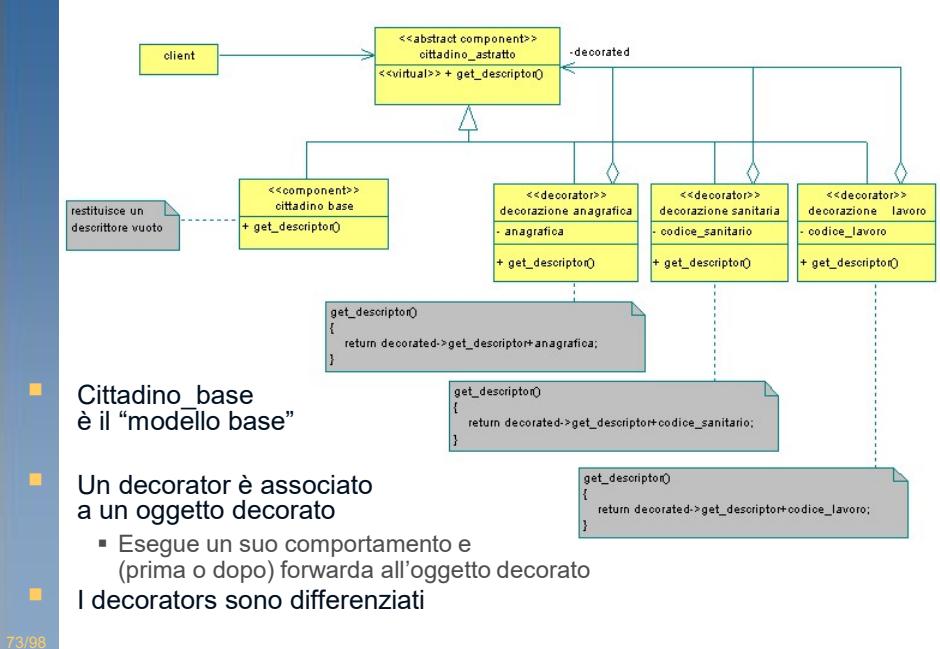


- La scelta delle responsabilità è statica ;-)
- Le combinazioni esplodono :-)

- Typical pros and cons of solutions based on sub-classing

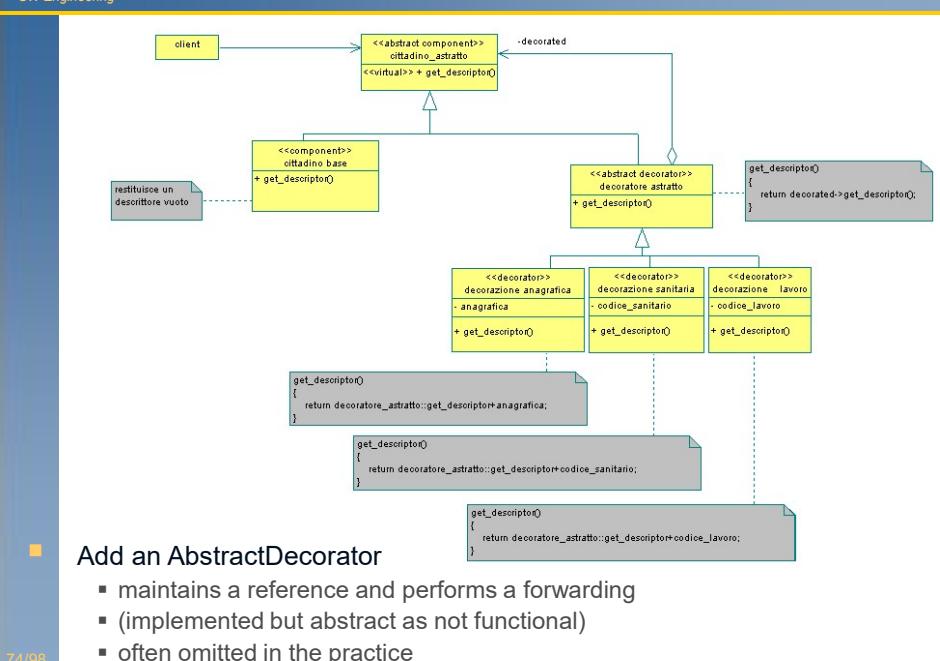
72/98

## Decorator from a case example: solution by composition



73/98

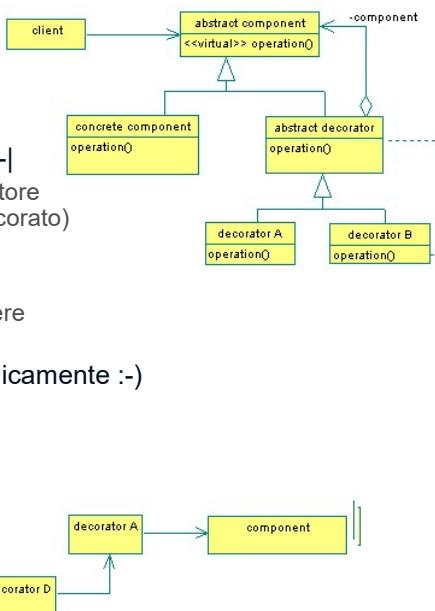
## Decorator from a case example: solution by composition



74/98

## Decorator - consequences

- Il numero delle classi scala in modo lineare :-)
- N+1 oggetti :-(
  - Difficoltà nel debug e testing
- Necessaria l'installazione iniziale :|
  - A carico del costruttore del decoratore (riceve il riferimento all'oggetto decorato)
- Efficienza :-(
  - Forwarding
  - Le classi alte devono essere leggere
- Configurazione modificabile dinamicamente :-(
  - Addition and withdraw



75/98

## Decorator - consequences

- le decorazioni modificano il comportamento ma non l'interfaccia
  - i.e. aggiunge behavior a operation(), ma non aggiunge una operation2()
  - spesso la decorazione implica operazioni specifiche
  - e.g. il cittadino nel profilo dell'assistenza sanitaria ha delle operazioni (scelta del medico, set del numero di codice, ...)  
che non sono rilevanti nel profilo anagrafico, e viceversa
- ciascuna funzione rilevante in qualche decorazione deve essere dichiarata nella classe base
  - la classe base raccoglie responsabilità non coesive
  - ciascun decoratore deve implementare il forwarding su operazioni che non sono per lui rilevanti
  - gli oggetti si appesantiscono in termini di occupazione di memoria
- Questo limita molto il campo di applicazione dello schema
- Dire che due oggetti hanno la stessa interfaccia significa che hanno le stesse operazioni, a livello concettuale
- non basta che ereditino da una classe base comune

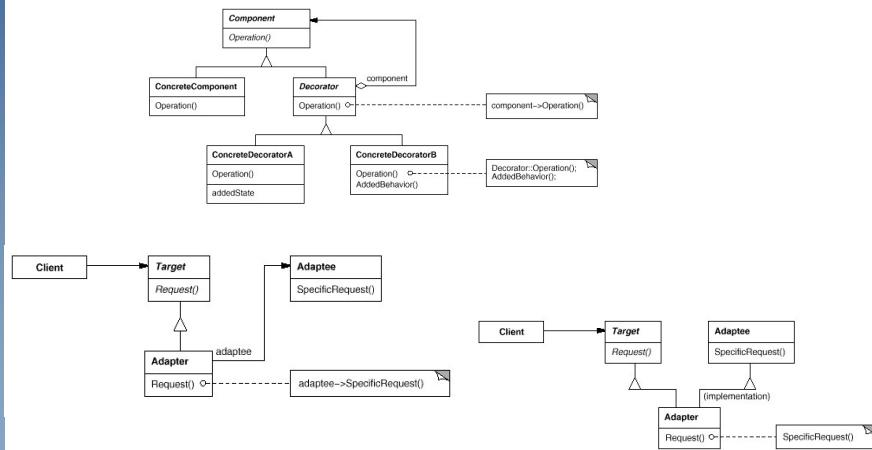
76/98

- **Decorator**
  - aggiunge comportamento senza modificare l'interfaccia
  - E per questo può essere applicato ripetutivamente
  - Adapter aggiusta l'interfaccia realizzando lo stesso comportamento
- **Adapter**
  - fa un po' il contrario:  
adatta una implementazione disponibile a una interfaccia attesa
- **Proxy**
  - Non cambia né l'interfaccia né il comportamento
  - Ha l'intento di placeholder
- **Bridge**
  - ha l' intento di separare a priori (up-front) l'implementazione dall'astrazione  
per favorire l'evoluzione disaccoppiata delle due
  - Adapter risponde piuttosto ad un approccio a posteriori
- **Decorator and Aspect Oriented Programming**

- (esame del 15 feb.2018 - 21 punti – 45 minuti)
- si consideri il caso in cui:
  - sono disponibili le implementazioni di una classe Base che espone un metodo op(), una classe D1 che espone un metodo op1(), una classe D2 che espone un metodo op2();
  - si vuole potere decorare l'operazione op() concatenando ad essa l'esecuzione di op1() e op2().
- Si progettano:
  - la struttura di classi e interfaccia Java che realizzano l'intento
  - senza modifiche alle classi Base, D1 e D2,
  - utilizzando il design pattern Decorator per la concatenazione delle operazioni
  - e il design pattern Adapter per adattare le implementazioni D1 e D2 all'interfaccia attesa per essere integrati nello schema del Decorator.
- Si dettaglino frammenti di codice che caratterizzano l'implementazione.
- Si discutano vantaggi e svantaggi
  - conseguenti all'adattamento in forma di Object-Adapter- o Class-Adapter in riferimento al caso specifico.

### An exercise: adapt classes to decorate

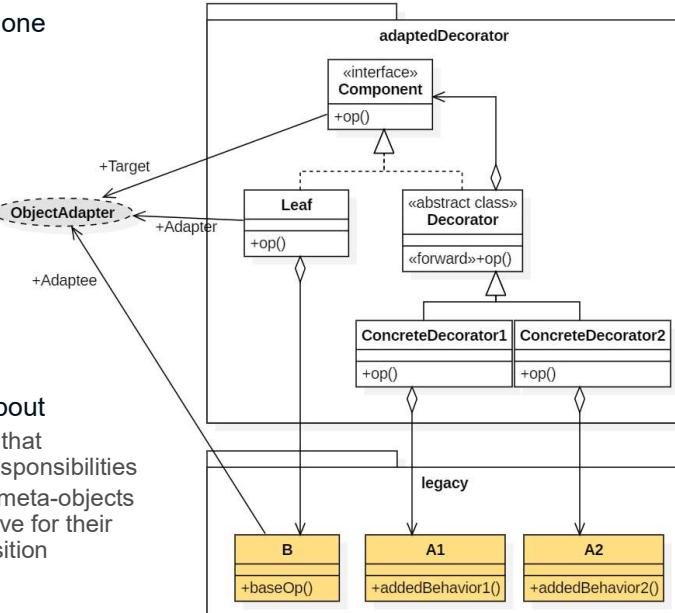
- (Per convenienza, sono riportati gli schemi dei due patterns, nella forma c++ fornita in [GOF])



79/98

### An exercise: adapt classes to decorate

- Una soluzione

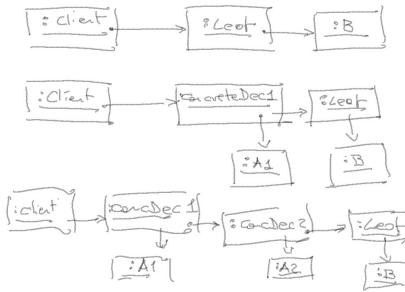
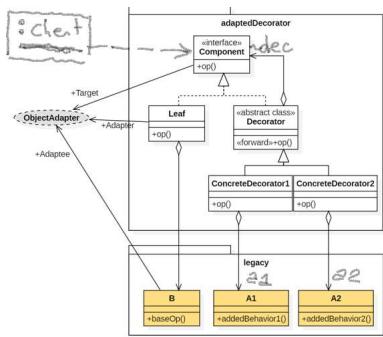


80/98

#### Remark about

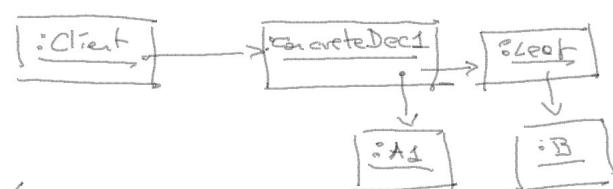
- objects that carry responsibilities
- ... and meta-objects that serve for their composition

## TBD: polymorphism – some possible object diag



81/133

## TBD: object creation and dependencies installation



```

// nel client
A1 myA1 = new A1(); // gli oggetti di tipo legacy
B myB = new B();
Component myChainedObject =
 new ConcreteDec1(myA1, new Leaf(myB));

```

```

// in Concrete Dec 1
// devo avere un costruttore con 2 argomenti
// da mettete le dipendenze
public ConcreteDec1(A1 a, Leaf l)
{
 dec = l; // installo sul field dec dell'abstract Decorator
 // l riferimento all'oggetto Leaf creato
 a1 = a; // compre l'oggetto Legacy ricavato
}

```

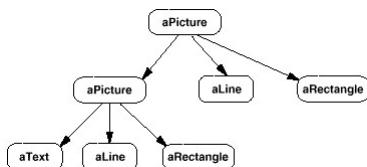
82/133

■ Intent

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

■ Motivation

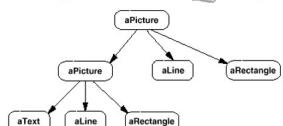
- a configuration is often made by compositions of components, that, in turn, can be either elementary or made by composition
- E.g. the elements of a graphical interface, the structure of a Computer SW Configuration Item (CSCI), a system made of subsystems in mechanics or electronics, ...
- Another perspective: a structure of classes that produces a tree of objects



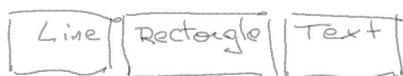
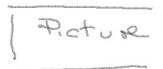
- The key to the Composite pattern is an abstract class that represents *both* primitives and their containers, so that a client is not requested to distinguish leaves from intermediate nodes

183/98

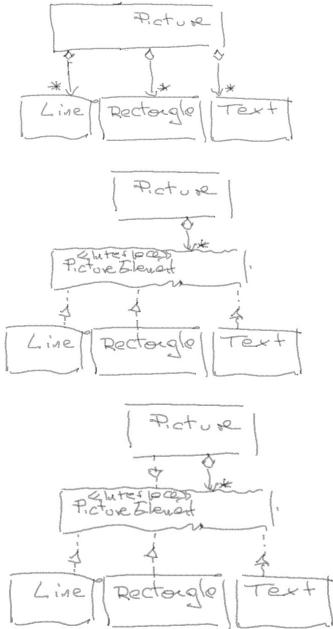
Assume we want to represent an object *Dog* shaped as a tree:



c' vorranno allora 4 classi:



84/133



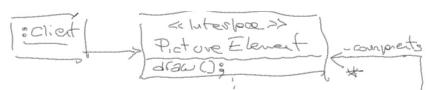
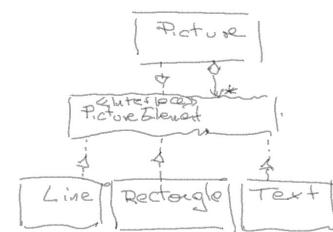
:( Picture deve gestire  
3 collezioni di riferimenti:  
a diversi tipi di oggetti  
generalizziamo i 3 tipi  
sotto uno come interface

:( non ho ancora lo spazio  
di strutturare lo Picture  
in sub pictures (layers)

:( anche Picture deve poter  
essere in PictureElement

call this Composite.

85/133

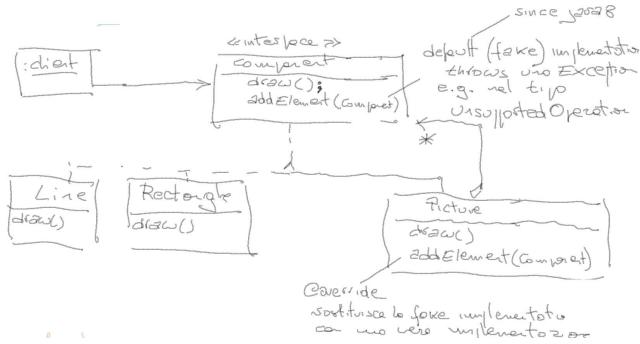


implementazione tipo specifico  
di draw

implementata con uno switch  
foreach (PictureElement c: components)  
{ c.draw(); }

86/133

## TBD: Maximizing the interface



### Consequences

- Per il client ho traspresso, rendendolo tipo
- è un tipo concreto  
ho metodi solo per questo  
gli si affido.
- I.e. Line e Rectangle restano  
completamente visibili  
di essere composti in un Composite

87/133

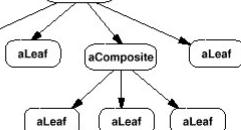
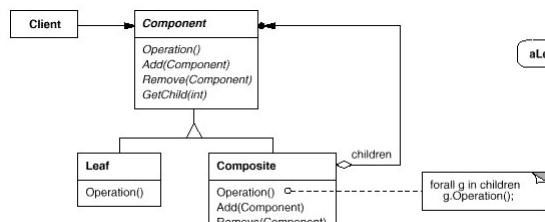
questo è un meccanismo  
di volo re generale

- se l'interfaccia espone metodi  
che non si affidano a tutti i tipi concreti
- allora l'interfaccia fornisce una finta implementazione  
(since Java8 a default method)  
che genera un'eccezione
- sulle classi a cui si affida il metodo  
esegue un @override della finta implementazione

88/133

## Composite - Structure

### Structure (of classes, and objects)



### Participants

- **Component**: declares the interface for objects in the composition; implements default behavior for child-related operations.
- **Leaf**: defines behavior for primitive objects in the composition.
- **Composite**: defines behavior for child-related operations; stores reference to child components.
- **Client**: manipulates objects in the composition through the Component interface, without distinguishing Leafs from Composites
- Default behavior in the Component is an example of the evil that occurs when a base class is charged of operations not shared by all derived classes

189/98

## Composite

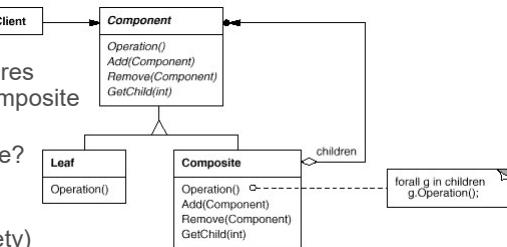
### Consequences

- defines class hierarchies consisting of primitive objects and composite objects.
- makes the client simple by allowing uniform treatment of composite structures and individual objects
- makes easy to add new kinds of components: newly defined Composite or Leaf subclasses work automatically, and Clients don't have to be changed.

190/98

■ Maximizing the Component interface

- the first aim is transparency:  
make client unaware  
of Leafs and Composites
- to this end, Component declares  
all operations of Leaf and Composite
- syntactic question: how does  
the interface become concrete?
- behavioral question:  
what if add() is invoked  
on a Leaf object? (aim of safety)



■ Component is an abstract class

- provides a "fake" implementation for child-related operations  
overridden by Composite, and used by Leaf
- ... possibly rising an exception, which should not happen:  
the pattern is not for the case of a Leaf be promoted to Composite;  
neither for providing robustness in the creation;  
much more for making traversal easy when the structure is up.

■ The solution is standard but violates a principle of class hierarchy design

- a class should define only operations that are meaningful to all subclasses

■ explicit parent references

- a child can maintain a reference to its (unique) parent to facilitate traversal
- usually charged to Component
- In this case, special behavior occurs on the root

■ Sharing components

- when parent reference is not explicit,  
the same (equal) Component can appear in multiple Composites
- to reduce storage, but also to guarantee consistency

- *The pattern Composite is the right place to discuss the concept of subtyping  
(and interface maximization): is the subclass really a subtype?*
- *It is also the right place to exemplify the usage of default implementations*

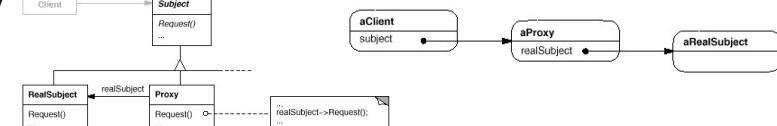
## On topologies of classes and objects - 1/2

- When you program you see classes, but behavior, execution, debugging, and testing are much about objects
  - the relation is often not direct
  - the line of evolution proxy->decorator->composite is a good example
  - by the way, note that they are all classified as structural (convert the interface of a class into another interface), but they also much support the purpose of behavioral patterns: (shift the focus from control flow among objects to composition of classes)

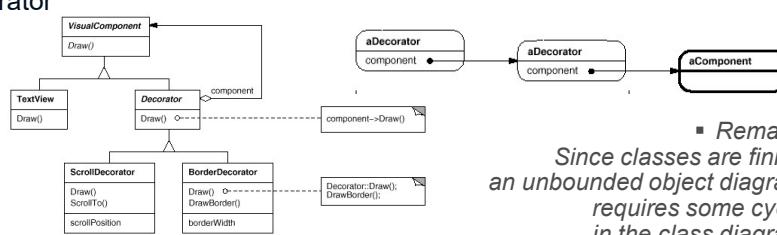
193/98

## On topologies of classes and objects - 2/2

### Proxy

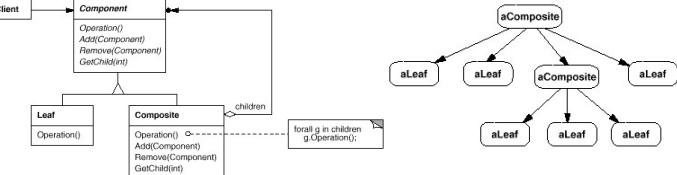


### Decorator



▪ Remark:  
Since classes are finite, an unbounded object diagram requires some cycle in the class diagram

### Composite



194/98

Enrico Vicario - AA 22/23  
SW Engineering

### Composite + Observer

- exercise: Composite + Observer : esame 22 settembre 2016  
(le parti sono osservate, e.g. da un agente di analytics)
- Exercise: puo' esserci anche l'inverso: le parti osservano  
(e.g. un listino prezzi)
  
- TBD (exercise) represent an arithmetic expression implementing evaluate();  
is limited by the fact that the expression is usually binary or unary, unless also  
a ternary operator is added.
- TBD: (exercise) represent a fault tree with repeated events.  
Addresses the case of shared components,  
and operators with multiple components  
(And-gate over N inputs, or also K-out-of-N-gate).  
implement the method evaluate().  
Can motivate the usage of a Builder in the construction,  
based on the pasring of an input string

95/98

Enrico Vicario - AA 22/23  
SW Engineering

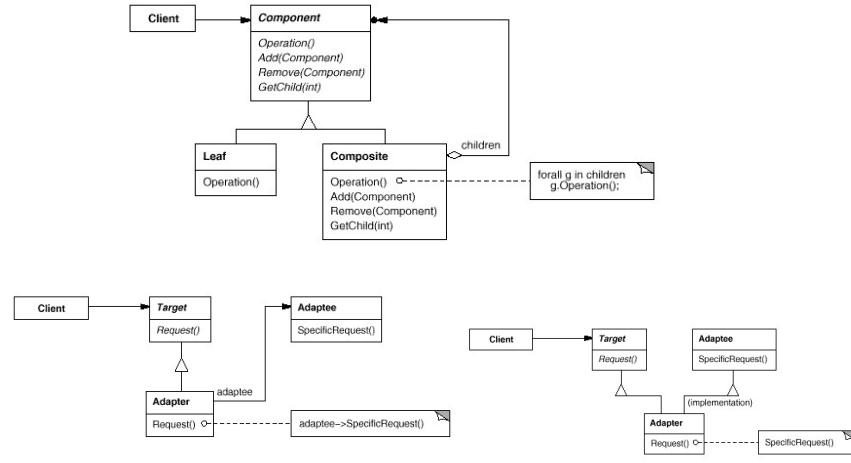
### An exercise: adapt classes to fit in a composite

- (esame 18 gen.2018 - 21 punti – 45 minuti)
- Si consideri il caso di
  - un insieme di componenti SW aggregati in una gerarchia per realizzare una funzione di monitoraggio applicativo:
  - la radice del sistema espone un'operazione boolean test() che viene distribuita verso i componenti per identificare in quali livelli si riscontrano dei malfunzionamenti.
  - Si consideri in particolare il caso in cui i componenti, che qui denominiamo come A, B, C sono già disponibili ed offrono operazioni rispettivamente denominate opA(), opB(), opC() con cui è possibile testare la corretta funzionalità.
- Si definisca
  - la struttura di classi e interfacce Java che realizzano lo schema
  - usando il pattern Composite per rappresentare la gerarchia
  - e il pattern Adapter per adattare i componenti pre-esistenti all'interfaccia attesa per la composizione.
- Si dettaglino frammenti di codice che caratterizzano l'implementazione.
- Si discutano vantaggi e svantaggi consequenti
  - all'adattamento in forma di Object-Adapter- o Class-Adapter in riferimento al caso specifico.
  - alla realizzazione del Composite come interfaccia o classe astratta.

96/98

### An exercise: adapt classes to fit in a composite

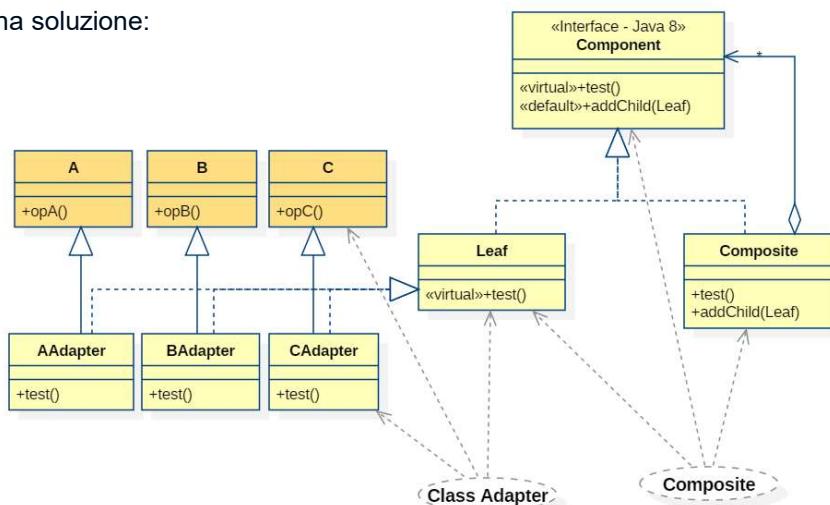
- (Per convenienza nella denominazione viene riportato uno schema dei due patterns, riferito al c++, che però omette il dettaglio su come sono implementate le children-related operations in Leaf)



97/98

### An exercise: adapt classes to fit in a composite

- Una soluzione:



#### Remark:

- A,B, and C are completely unaware of being incorporated in a Composite
- What would change if an Object Adapter was used?

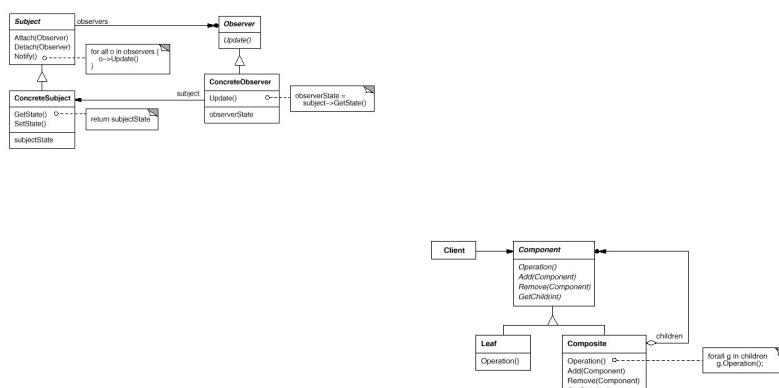
98/98

### An exercise: a Monitor observes the Leaves of a Composite

- (esame 22 Sett.2016 – 22/30 punti – 45 minuti)
- Si consideri uno scenario che combina i patterns *Composite* e *Observer* per realizzare un meccanismo di monitoraggio dello stato delle parti di un prodotto composto:
  - Il pattern Composite è usato per rappresentare un sistema composto di parti;
  - Il pattern Observer è usato per realizzare un monitor globale che riceve notifica delle variazioni dello stato di ciascuna parte;
  - per ottenere questo comportamento, ciascuna parte implementa il ruolo di concrete subject e il monitor implementa il ruolo di observer.
- Si descriva la struttura con cui vengono composti i due patterns
  - attraverso l'uso di un class diagram;
- Si descriva il funzionamento in uno scenario caratterizzante;
- Si discuta il modo con cui il monitor può gestire
  - notifiche che provengono da più parti distinte e di tipo diverso;
- Si dettaglino frammenti di codice della realizzazione
  - che illustrano aspetti salienti dello schema.

199/98

### An exercise: a Monitor observes the Leaves of a Composite

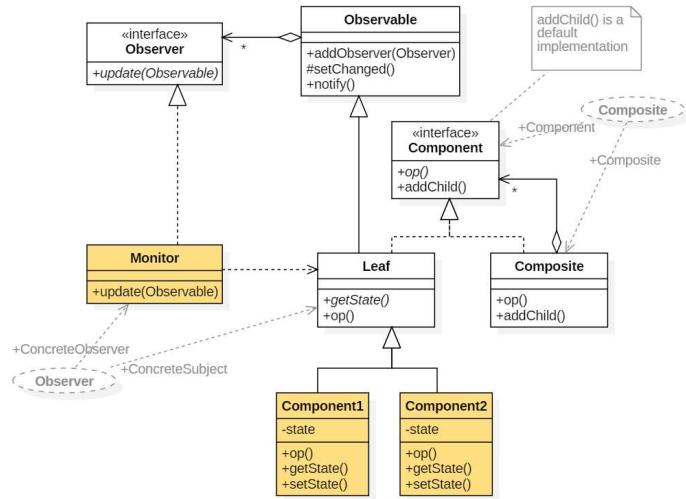


100/98

## An exercise: a Monitor observes the Leaves of a Composite

### Intent

- the leaves of a Composite structure are observed by a Monitor



### Variants

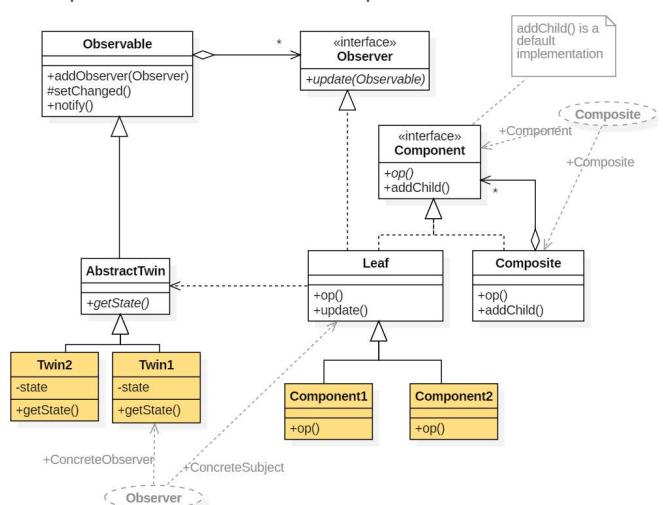
- the Monitor might observe also Composites
- legacy Components might need adaptation

101/98

## An exercise: the leaves of a structure observe respective twins

### Intent

- the leaves of a composite structure observe respective twin entities



### Variant

- also the Composites might be observing a twin

102/98

- exercise: Composite + Builder : esame 16 febbraio 2016

■ Problem

- Assistito is responsible for the operation `getStorico()`, which retrieves the history of health services provided to some Cittadino
- `getStorico()` must have different implementations depending on whether the history is retrieved for clinical purposes by the Cittadino or by his/her present/past Physician, or if it is retrieved for administrative purposes

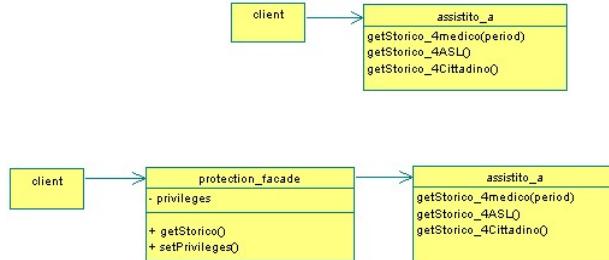


• TBD: this example is somehow misleading:  
in general, the Strategy does not depend on Client type (or identity).  
It may commonly depend on the state of some Context

## A first solution

- Basic solution: Assistito exposes different operations

- The responsibility of choosing the right implementation is left to the Client
- An intermediate proxy could add protection
- Yet, the choice is repeated by some object at each call



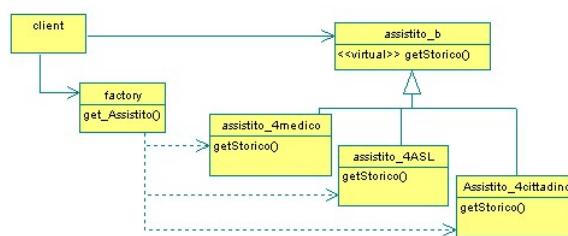
- TBD: Out of the expected rhetoric: what about overloading the operation name and passing the client self-reference to drive the binding?*

105/98

## Strategy: a solution based on subclassing

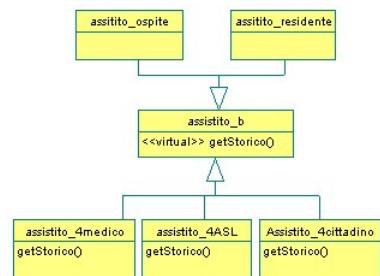
- Localize selection of the specific implementation at creation

- Possibly abstracted through delegation to some Factory



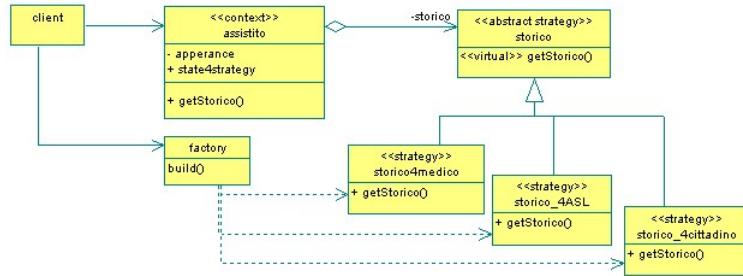
- Leads to a stiff pattern,

- TBD: questo ha a che fare col bridge, c'entra anche con lo strategy?*
- E.g. not open to encompass multiple directions of specialization or to evolve for a refinement of types
- Usual pros and cons of extension by subclassing



106/98

## Strategy: a solution based on composition

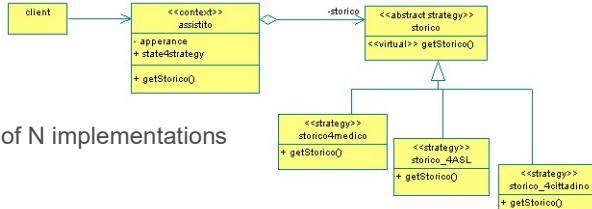


- In principle the ConcreteStrategy can be changed dynamically
  - E.g. based on some run-time performance metrics
  - If not useful, might be a limit, adding space for unexpected mutability
    - TBD: changeability can be inhibited making storico be final
- Strategy can receive a reference to some Context
  - or provide a setDelegate() operation allowing an external object to select

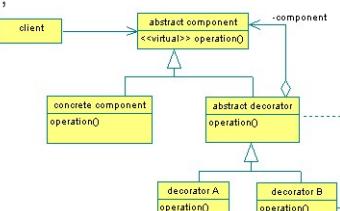
107/98

## Strategy - Related patterns

- Strategy
  - Changes behavior, by selecting one out of N implementations



- Decorator
  - Changes behavior through layered additions, with different possible combinations and orders



108/98

■ Java idioms for the Strategy pattern

- using anonymous nested classes  
specialized behavior can be determined even object by object

- *TBD: later on, comparing strategy vs bridge is the right place to mention that the structure is not sufficient to identify a pattern, intent is also needed*

109/98

■ (Esame 24 Gen.2018 – 22 punti – 45 minuti)

■ Si consideri uno scenario che combina i patterns *Observer* e *Strategy* per realizzare un meccanismo di adattamento:

- il Context è registrato come Observer su un qualche oggetto che opera come Monitor dell'ambiente di operazione;
- quando il Monitor rileva una variazione nello stato di operazione, il Context dello Strategy riceve notifica e adatta di conseguenza la Strategy con cui viene eseguita una qualche operazione.

■ Si descriva la struttura

- con cui vengono composti i due patterns, il Monitor e un Client che opera nel ruolo di test driver (il main) attraverso l'uso di un class diagram (8 punti)

■ Si illustri il funzionamento dello schema

- con un sequence diagram riferito a uno scenario caratterizzante (6 punti);

■ Si dettaglino frammenti di codice della realizzazione

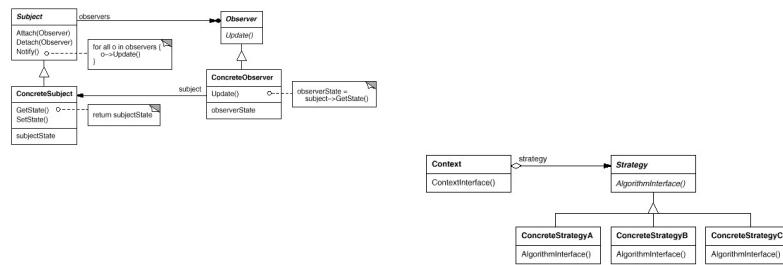
- che illustrano gli aspetti salienti dello schema

■ si definisca uno scenario di test

- realizzato per semplicità nella forma di un main(),
- che esercita lo schema in uno scenario che ne caratterizza l'intento (8 punti).

110/98

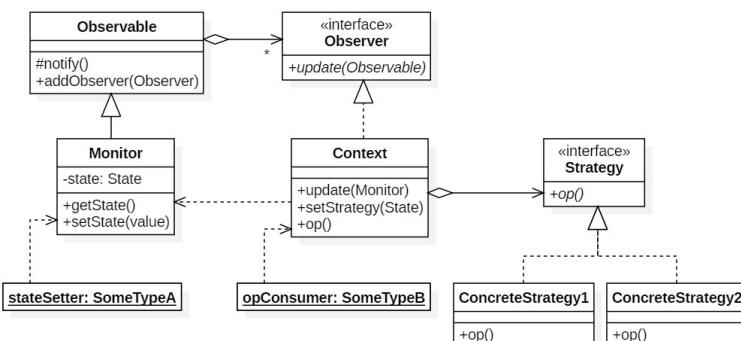
### An exercise: adapt the strategy upon notification



111/98

### An exercise: adapt the strategy upon notification

- Intent: a Context adapts the Strategy upon notifications from a Monitor



- A test that verifies correctness and documents the intent

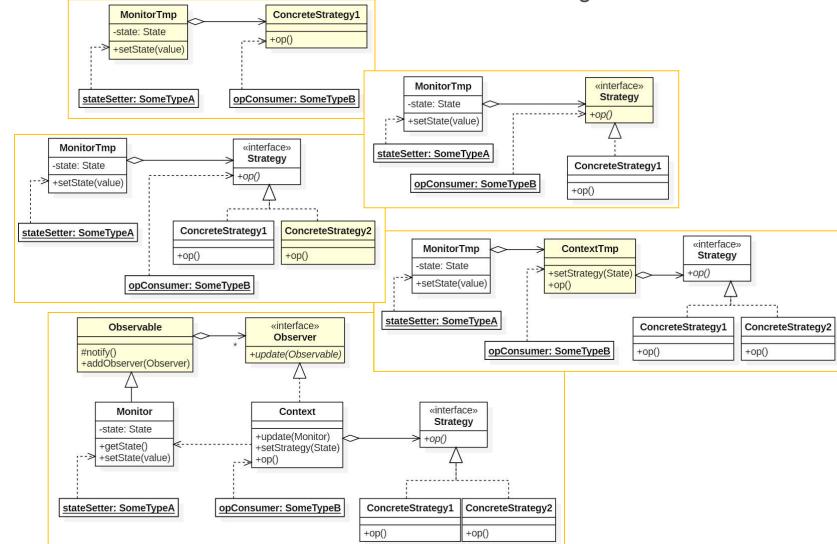
- create a **Monitor**; create a **Context** with a default **Strategy**; register the **Context** on the **Monitor**.
- Test #1: invoke **op()**.
- Test #2: invoke **setState()** with a value that does not change **Strategy**; invoke **op()**.
- Test #3: invoke **setState()** with a value changing the **Strategy**; invoke **op()**.

112/98

### An exercise: adapt the strategy upon notification

- A roadmap to the scheme, a case of refactoring

- subsequent refactoring enabling intermediate testing checkpoints in XP style
- to be applied in the implementation more than in design

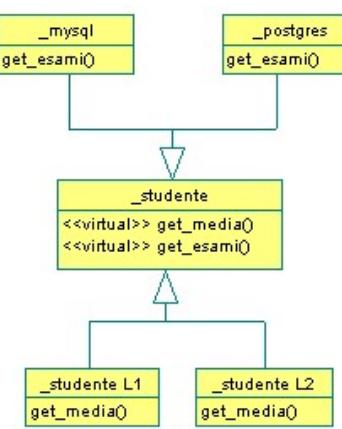


113/98

### Bridge (structural)

- Un tipo nel modello di specifica ha diverse specializzazioni e diverse implementazioni

- Gli studenti della laurea triennale e della laurea di specializzazione hanno una diversa rappresentazione del curriculum e.g. un diverso calcolo della media
- Facoltà diverse usano diverse tecnologie di database e.g. diversa rappresentazione degli esami sostenuti

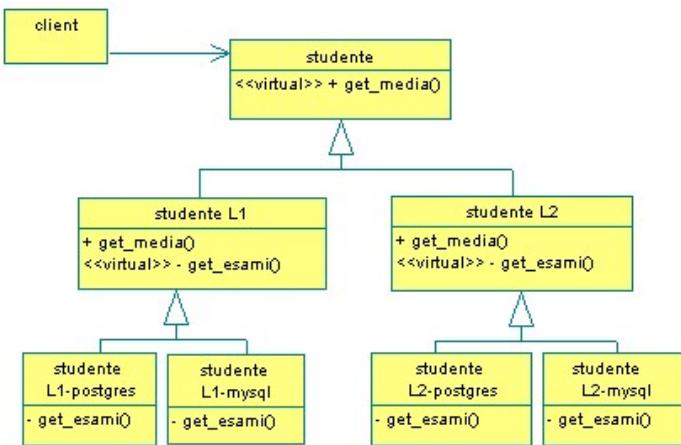


- E' un caso di nested generalization [Rum91]

- Una direzione attiene al tipo, l'altra all'implementazione

114/98

## Soluzione basata sull'ereditarietà



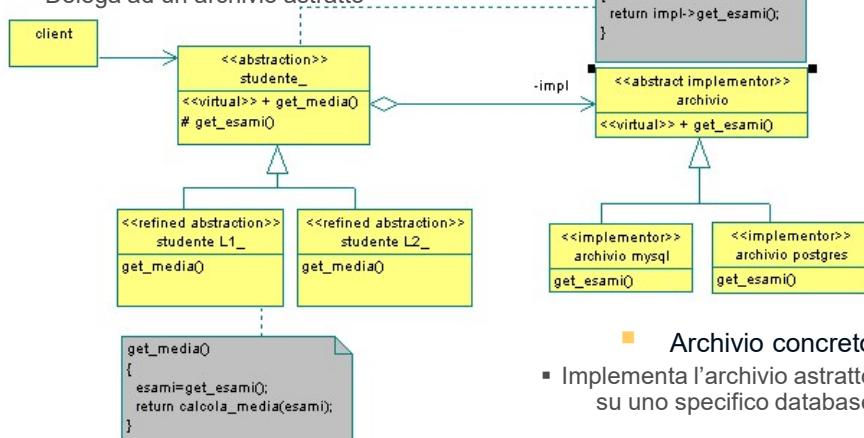
- Ha le solite caratteristiche di uno schema di subclassing
  - Un solo oggetto, facile da installare :-)
  - Ma il numero delle classi esplode :-(
  - E la scelta della classe è statica :|  
(non un problema nello specifico esempio)

115/98

## Bridge - schema per composizione

### Studente astratto

- Implementa una narrow interface (protected) che incapsula la dipendenza dalla tecnologia del database (`get_esami()`)
- Delega ad un archivio astratto



### Archivio concreto

- Implements the abstract storage on a specific database

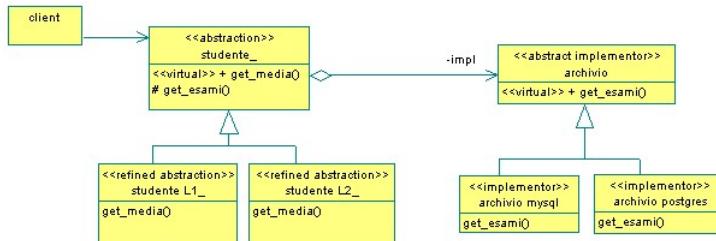
### Studente concreto

- Implements the operations that depend on the type of student

116/98

## Bridge - conseguenze

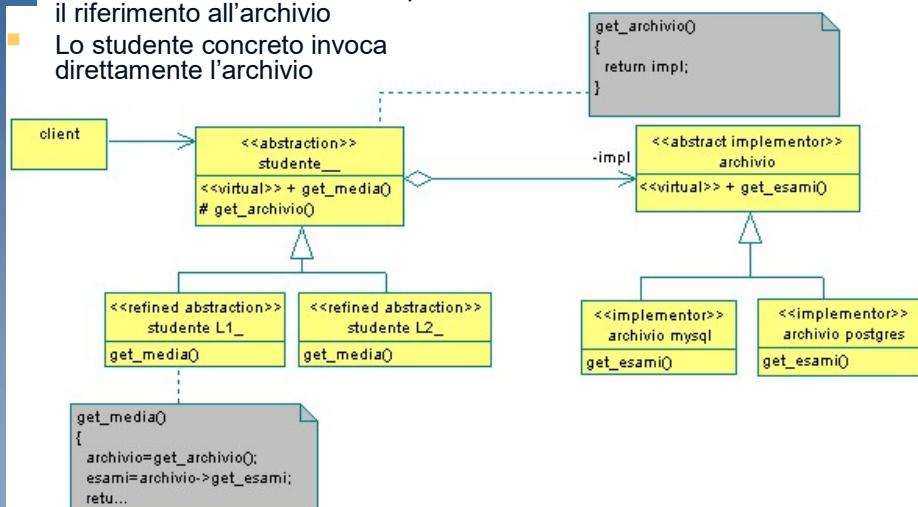
- Ci sono 2 oggetti :-(
- L'associazione deve esser installata :-(
  - responsabilità a carico del costruttore dello studente concreto
- In principio l'associazione può essere modificata dinamicamente :|
  - Ma non è l'intento, semmai è il caso di strategy
- Disaccoppia una astrazione dalla sua implementazione per lasciare che le due ammettano gerarchie di specializzazione indipendenti :-)



117/98

## Bridge - raffinamento

- Lo studente astratto rende disponibile ai suoi eredi il riferimento all'archivio
- Lo studente concreto invoca direttamente l'archivio

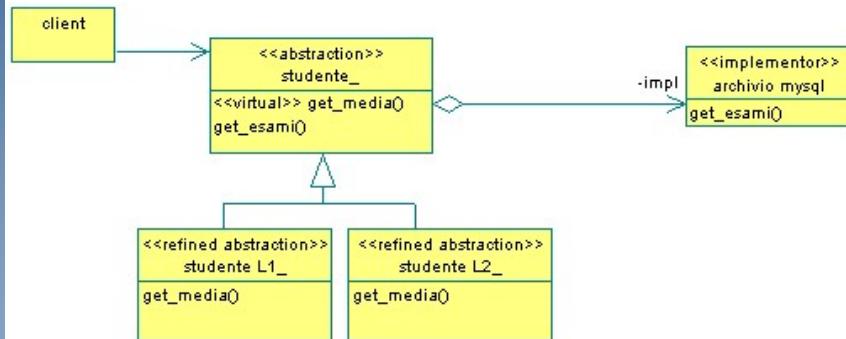


118/98

- Risparmia un forwarding (efficienza) :-)
- Però accoppia lo studente concreto con l'archivio :-(

## Bridge - semplificazione

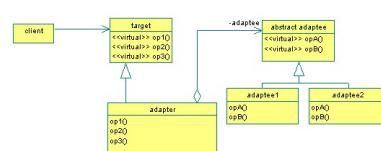
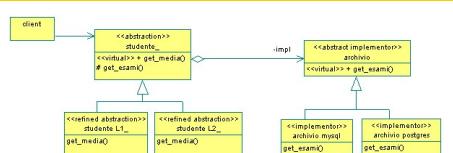
- Lo schema della delega può servire comunque a disaccoppiare astrazione e implementazione
- oppure a disaccoppiare parti dei requisiti con diversa stabilità
- Ovviamente fino a che c'è una unica implementazione è inutile la implementazione astratta (!)



119/98

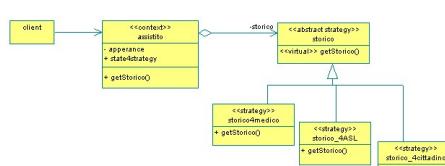
## Bridge - related patterns

- Bridge
- vs. Adapter
  - Il bridge divide un componente in due parti ortogonali, l'adapter adatta due componenti concettualmente equivalenti
  - Il bridge è a priori (up-front): separa l'implementazione dall'astrazione per favorirne una probabile evoluzione disaccoppiata
  - l'adapter a posteriori



### vs. Strategy

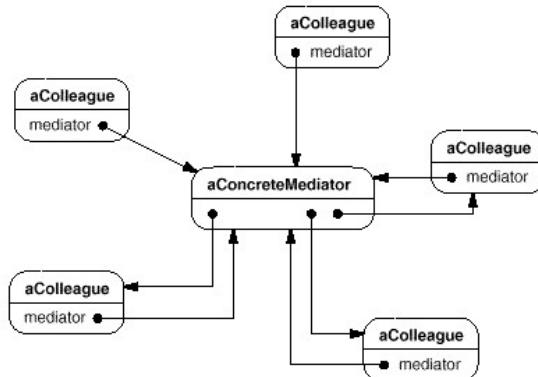
- La struttura di strategy è un subset di bridge
- Ma l'intento è diverso



120/98

## Mediator (behavioral)

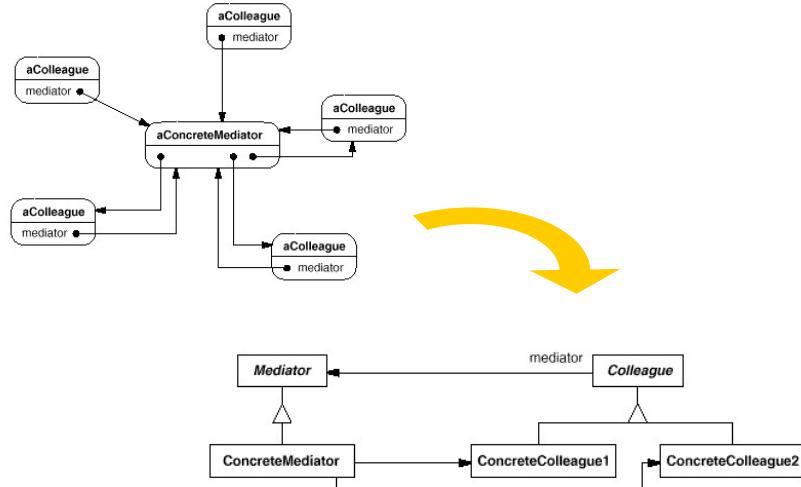
- Definisce un oggetto che incapsula il modo con cui interagiscono un insieme di oggetti
  - disaccoppia gli oggetti evitando che essi si referenzino direttamente



- TBD: è behavioral o structural? (se questo conta qualcosa)

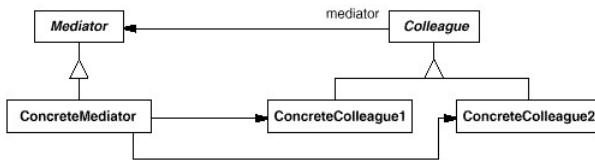
121/98

## Mediator - structure



122/98

- i colleghi sono di tipo concreto diverso
- tutti i colleghi vedono solo il mediator
  - sono alleviati dal conoscersi a vicenda
  - stanno sotto una classe comune da cui ereditano il riferimento al mediator
- Il mediator vede la diversità dei colleghi
  - assume la responsabilità di formare la configurazione
  - i colleghi stanno sotto la stessa classe ma hanno operazioni diverse
  - il mediator astratto può essere omesso se i colleghi partecipano ad un unico gruppo
- Il mediator opera come front-end (interfaccia) verso i clienti

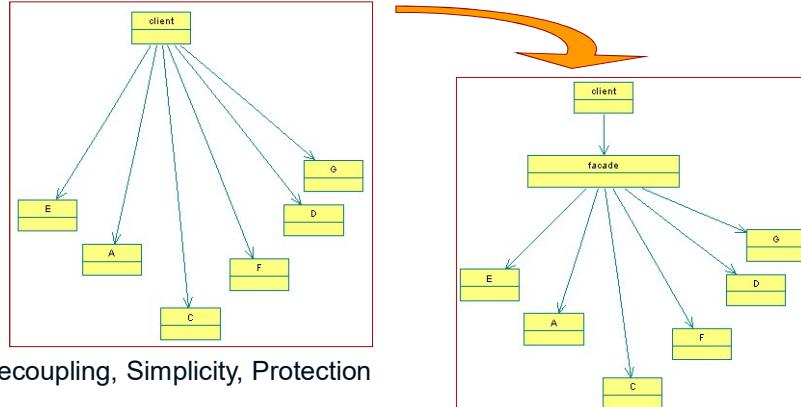


*TBD: ci sono cmq dei down-casts?  
i.e. il mediator concreto deve conoscere il tipo di ciascun colleague concreto?  
si capisce su un'implementazione*

- risponde alla complessità che nasce con l'uso diffuso della composizione e del suo possibile adattamento dinamico
  - crea astrazione sul modo con cui oggetti sono composti e interagiscono
  - per cambiare la composizione è sufficiente subclassare il mediator riusando gli stessi colleghi
- Il mediator centralizza il controllo dell'interazione
  - scomponete relazioni n->n in relazioni n-a-1 e 1-a-n :-)
  - questo può rendere il mediator molto complesso e monolitico :-(
- Possibile organizzare il controllo secondo lo schema observer
  - il mediator opera come observer e i colleghi come subjects
  - quando un collega cambia stato lo notifica al mediator che poi propaga l'interazione
  - il notify del subject/collegue non ha iterazione perché esiste un unico observer/mediator
  - l'update sull'observer/mediator chiama indietro più subjects concreti secondo lo schema di controllo del gruppo
    - schema observer in modo push o pull ?

## Facade (structural)

- Provides a unified interface for interfaces of a set of classes



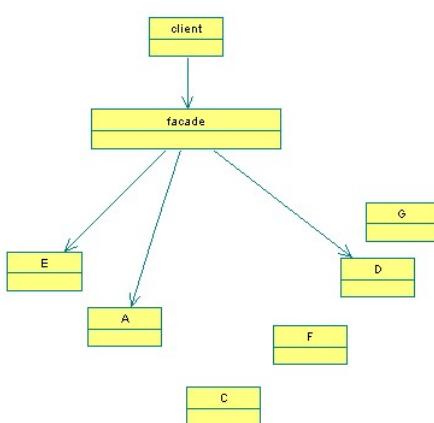
- Decoupling, Simplicity, Protection

- But, exposes something that shall then be preserved,
  - the facade itself may absorb the evolution.
  - Better through a combined use of interfaces and abstract classes (reduces the impact on the side of classes used by the facade, not on the side of clients of the facade)
    - ... possibly through interfaces with default implementations

125/98

## Facade - configurazione

- La facciata può essere configurata all'installazione

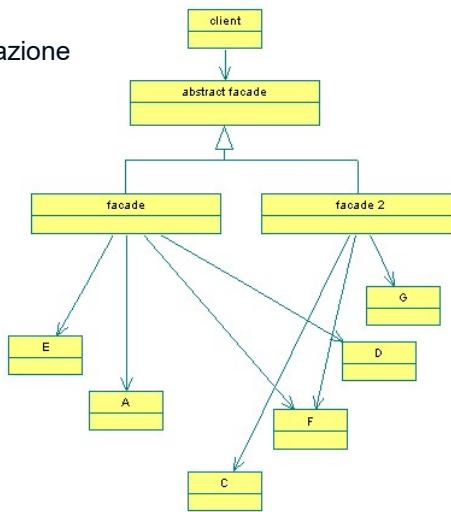


- In principio la configurazione può essere variata dinamicamente

126/98

## Facade - configurazione

- La specializzazione della facciata può essere realizzata per ereditarietà
- La scelta è statica
- Non c'è complessità di installazione

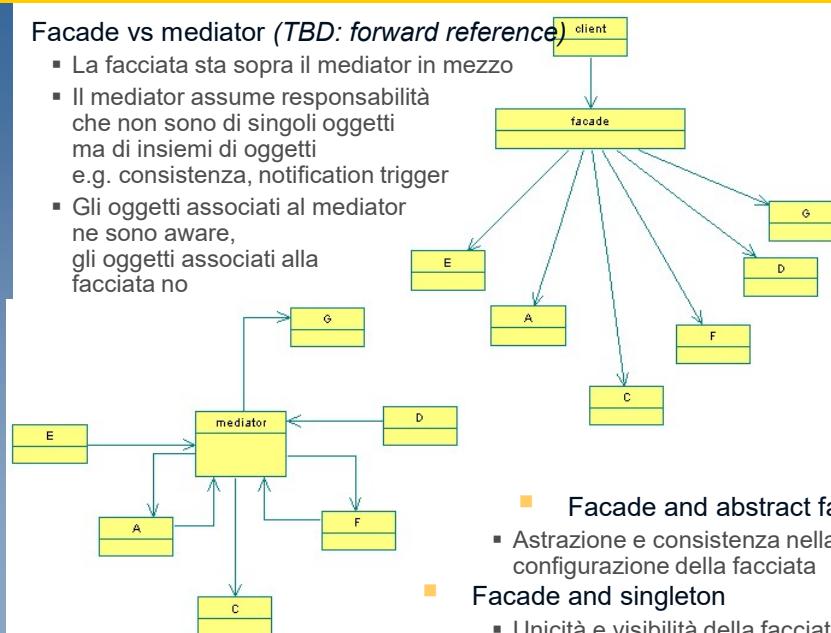


127/98

## Facade - related patterns

### Facade vs mediator (TBD: forward reference)

- La facciata sta sopra il mediator in mezzo
- Il mediator assume responsabilità che non sono di singoli oggetti ma di insiemi di oggetti e.g. consistenza, notification trigger
- Gli oggetti associati al mediator ne sono aware, gli oggetti associati alla facciata no



### Facade and abstract factory

- Astrazione e consistenza nella configurazione della facciata

### Facade and singleton

- Unicità e visibilità della facciata

128/98

- *TBD: mention that facade can be used as public class of a package*
- *TBD: later on in the development, after use cases, domain model and business logic, it will be possible to say that: facade can also serve to bridge use cases (in the functional perspective) and classes in a domain model (in the structural perspective)*
- *TBD: mention the role of facade in relation to Rest APIs and microservices*

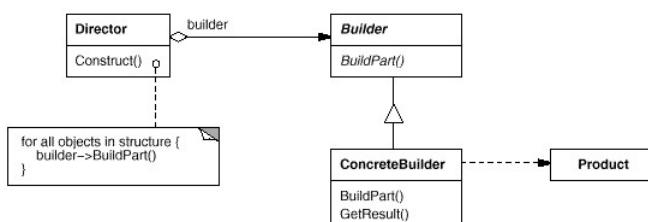
## Builder (creational)

### Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - *TBD: could the relevance of the latter line be questioned?*

### Motivation

- Separate the responsibility of knowing the sequence of construction from the ability to perform single steps *also* with different implementations
- E.g. assemble a Composite with different types of Leaf
- E.g. assemble a memory structure based on the parsing of some XML file



### Participants

- Builder: specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder: implements the Builder interface; maintains the state of the Product under construction, provides a method to retrieve the Product
- Director: determines the sequence of construction of parts

## Builder - more on participants CRC

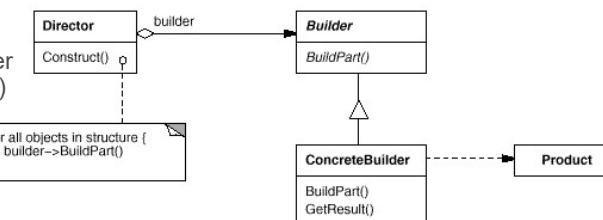
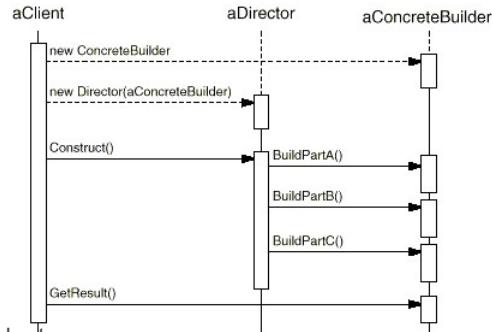
- Client
  - creates a Director and gives it a ConcreteBuilder (stored as a Builder)

### Director

- on invocation of construct(), implements the construction through properly sequenced commands to the Builder
- does not keep any information on the state of the Product, unless parts are mutually dependent (e.g. install a connection between two nodes)

### Client

- finally retrieves the Product from the ConcreteBuilder through getResult()



131/98

## SLIDE ORRENDA! Builder - an example: assembling a Pizza - 1/2

### Client view of the process:

- creates a PizzaBuilder, choosing the concrete version for a specific type of Pizza
- passes it to the constructor of a Waiter (Director);
- invokes construct on the Waiter;
- get the Pizza (Product) from the PizzaBuilder

- *TBD: questo esempio è abbastanza orrendo.*

*Però illustra bene che cambiando Director si cambia la struttura del prodotto, che fa la differenza rispetto a un abstract factory (che cambia solo la linea di implementazione degli ingredienti)*

- *E.g. un migliore esempio potrebbe essere*

*il driver del test di un decorator (o un altro schema strutturale) dove ciascun diverso director testa un insieme di classi in una diversa configurazione dell'object diagram.*

- *e.g. anche uno schema di composizione di microservizi che si adatta a uno scenario*

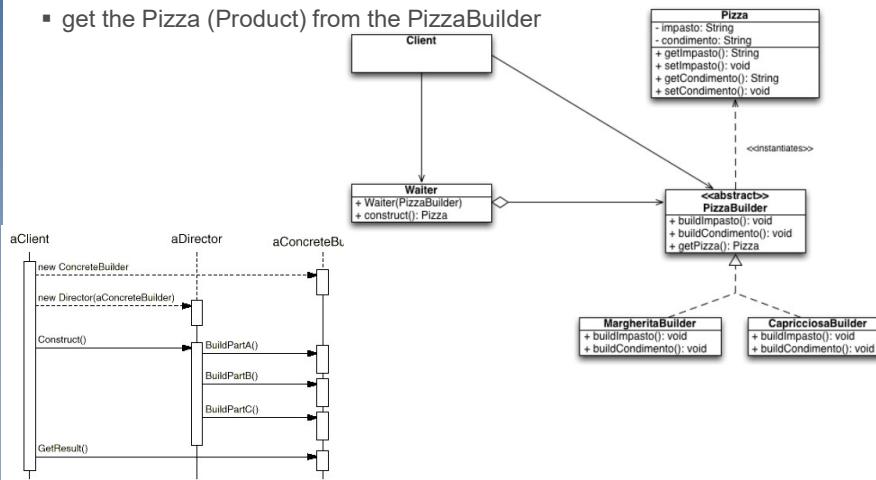
- *e.g. anche il concetto di software product line*

132/98

## Builder - an example: assembling a Pizza - 1/2

- Client view of the process:

- creates a PizzaBuilder, choosing the concrete version for a specific type of Pizza
- passes it to the constructor of a Waiter (Director);
- invokes construct on the Waiter;
- get the Pizza (Product) from the PizzaBuilder



133/98

## Builder - an example: assembling a Pizza - 2/2

- Client

- depends on the particular type of Pizza (Product) only at the creation of the Builder

- Waiter (Director)

- Sequences the steps of construction (first the pasta and then the dressing), but is not aware of how the steps are performed (different Waiters could produce different dishes)

- Builder

- Performs the steps (add pasta, add condimento), maintains a representation of the Product and returns it through the method getPizza()
- is not aware of the sequence of steps

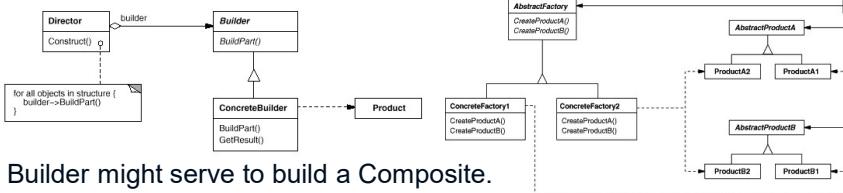
- Positive consequences follow

- Change the sequence without affecting creation of parts, and viceversa
- Support multiple versions of parts (*really relevant ?*)

134/98

■ **Builder vs Abstract Factory**

- Builder focuses on constructing a complex object step by step.  
Abstract Factory emphasizes a family of product objects (simple or complex).
- Builder returns the product as a final step,  
with an Abstract Factory, the Product gets returned immediately.



■ **Builder might serve to build a Composite.**

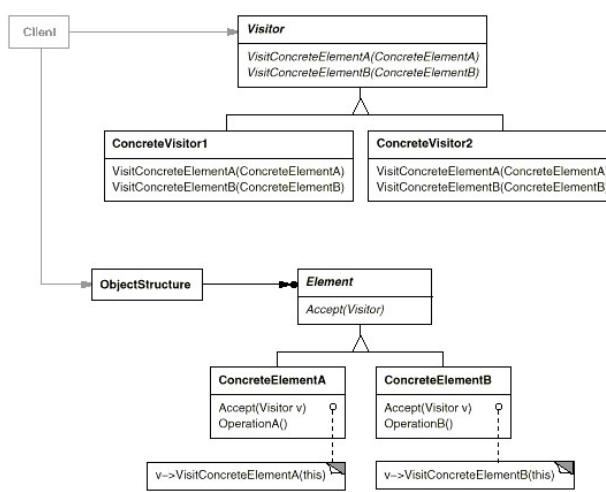
- overcoming the safety weakness of Composite  
due to the possible invocation of child-related operations on Leaf nodes

■ **Builder is often used in testing**

- to build a scaffold around the tested set of classes
- And in the construction of a memory structure from the parsing of a file
  - The parser act as Director
  - Full exploitation with different implementations for the same component
- **Builder is to creation as Strategy is to algorithm (TBD: forward reference)**
  - In the sense that it allows different implementations for steps

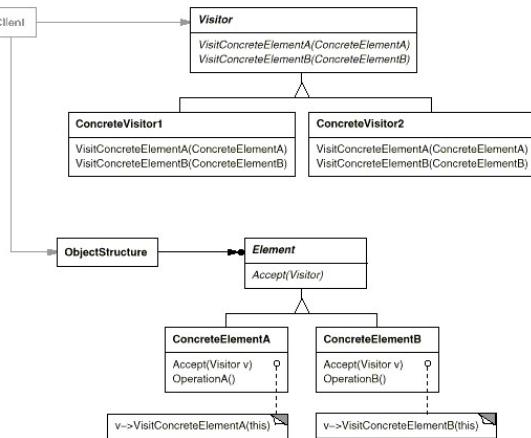
135/98

- rappresenta una operazione  
eseguita sugli elementi di una struttura di oggetti  
■ separa una gerarchia di dati e una gerarchia di operazioni



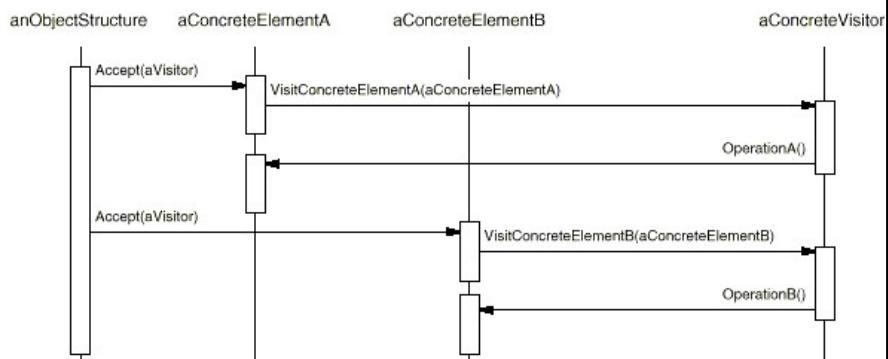
136/98

- object structure
  - punta una collezione di Element con implementazione concreta diversa
- concrete element
  - ciascun elemento accetta chiamate caratterizzate dal riferimento a un Visitor
  - sull'interfaccia esiste una unica funzione di accept
  - implementa l'interfaccia astratta element
- concrete visitor
  - implementa una operazione nelle diverse versioni che trattano elementi di tipo concreto diverso
  - implementa l'interfaccia astratta visitor



137/98

- call back - pull mode
  - sull'accept l'oggetto risponde chiamando il visitor concreto passato nella chiamata e passandogli il proprio indirizzo
  - il visitor opera sull'oggetto

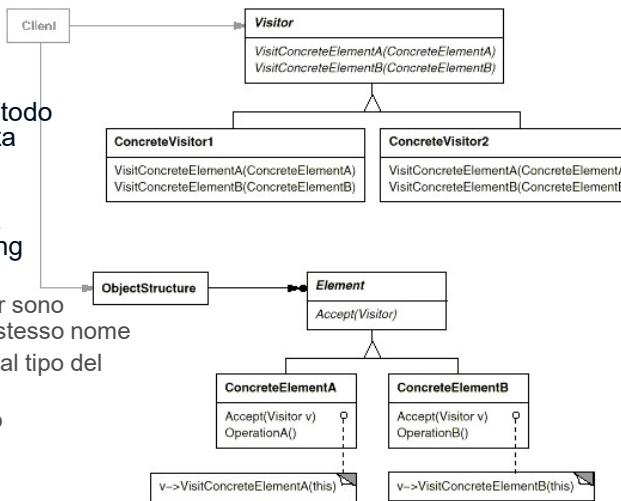


- TBD: il ConcreteElement invoca il tipo di visita che si applica al suo tipo su un Visitor di cui non conosce il tipo concreto

138/98

■ Double dispatch

- il client del metodo accept seleziona il doppio contesto formato dall'elemento trattato e dal visitor che esegue il trattamento



139/98

■ abilita l'evoluzione delle operazioni applicate agli elementi :-)

- una struttura dati stabile su cui non si prevedono a priori tutte le possibili elaborazioni (e.g. elaborazioni su un set di dati complessi)
- è una chiusura strategica (i.e. una cerniera, hinge)

■ mantiene localizzate operazioni logicamente coesive che operano su tipi diversi :-)

- offre anche un contesto locale per l'accumulazione nel corso della visita

■ la classe Visitor assomiglia un po' a un functoide :-)

- c'e' una operazione ma non uno stato
- il contesto su cui lavorare e' ricevuto nella chiamata
- il nome della classe (Visitor) e' quello di una funzione
- separa dati e funzioni

■ e' costoso aggiungere un nuovo tipo di elemento concreto :-)

- l'operazione è moltiplicata per il numero di visitors concreti

*▪ TBD: puo' aiutare una fake implementation come default method?*

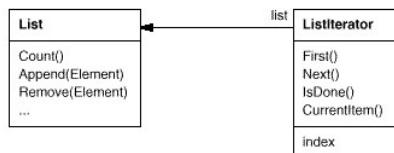
Element deve esporre un'interfaccia pubblica per tutto quello che è usato dal Visitor :-)

- problema comune a schemi in modo pull, e.g. getState nel pattern Observer

140/98

## Iterator (behavioral)

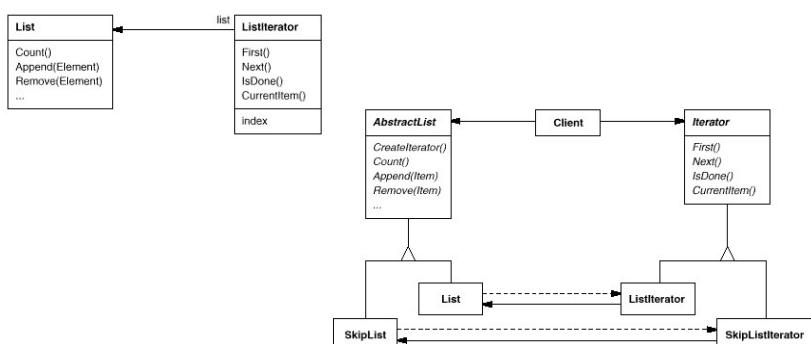
- offre un accesso sequenziale agli elementi di un oggetto aggregato senza esporne la rappresentazione
- oggetto aggregato = (multi)insieme di elementi omogenei
  - sull'oggetto sono possibili attraversamenti diversi
  - e.g. lista, albero, grafo, ...
  - e.g. visita in pre- o post-ordine, visita breadth- o depth-first, filtraggio, ...
- lo schema iterator sottrae all'aggregato la responsabilità della politica di attraversamento
  - mantiene e manipola un cursore che punta un elemento corrente
  - operazioni all'interfaccia: first, next, currentItem, isDone



141/98

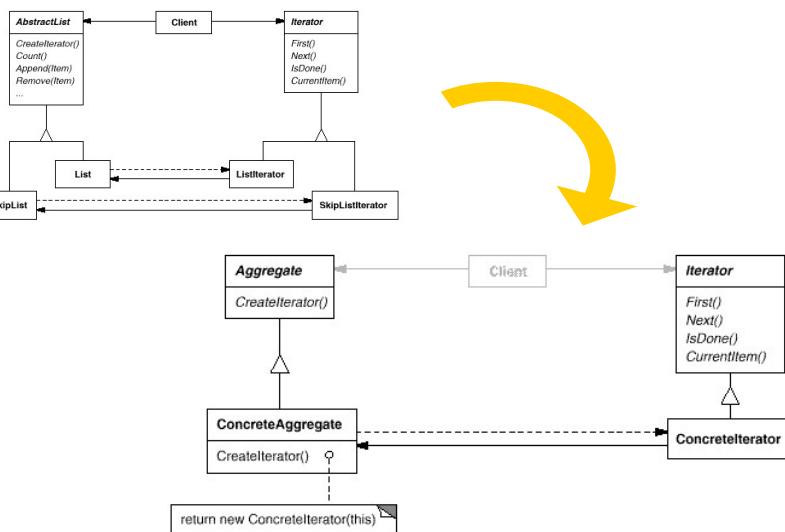
## Iterator - polymorphic iteration

- il client vede l'aggregato attraverso una interfaccia astratta che nasconde la rappresentazione concreta
  - e.g.: una lista può essere implementata con (quasi) la stessa interfaccia in forma sequenziale o collegata, con puntatori o indici, in ordine o meno, ...
- l'implementazione concreta dell'iteratore dipende dalla rappresentazione concreta dell'aggregato
  - la creazione dell'iteratore concreto è a carico della rappresentazione concreta dell'aggregato (operazione *CreateIterator()*)



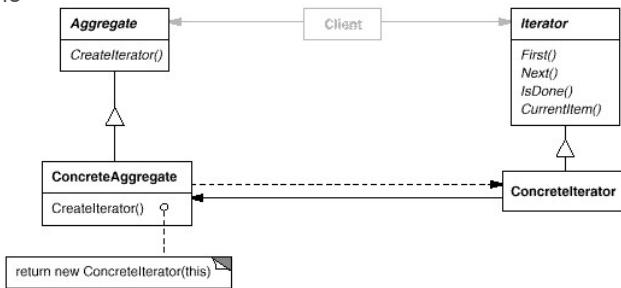
142/98

■ ... in generale:



143/98

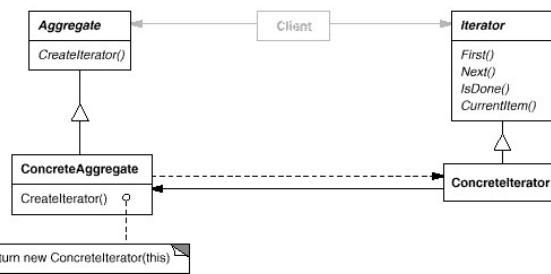
- **Iterator**
  - definisce una interfaccia astratta capace di tenere traccia di un elemento corrente e determinare un elemento successivo
  - diverse forme concrete implementano diverse politiche di attraversamento
- **Aggregate**
  - interfaccia astratta sull'aggregato (`addElement()`, `CountElements()`, ...)
  - include anche un metodo `CreateIterator()`
- **ConcreteAggregate**
  - implementa la rappresentazione concreta di `Aggregate`
  - crea l'iterator concreto e ci installa il proprio indirizzo



144/98

■ Client

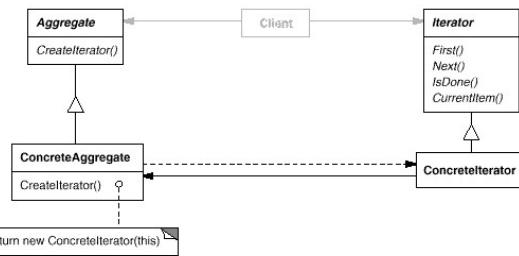
- delega ad aggregate la creazione di un iterator che poi usa per accedere gli elementi di aggregate stesso
- può controllare direttamente la visita operando su iterator (external iterator) oppure può delegare a iterator l'intera visita (internal iterator)
  - trade-off tra flessibilità e semplicità di uso



145/98

■

- permette di variare la politica di attraversamento di un aggregato
  - e.g. visita in pre-ordine o post-ordine, filtraggio sugli elementi visitati, ...
  - per farlo è sufficiente cambiare la forma concreta istanziata dell'iteratore
- presenta una interfaccia astratta sull'aggregato a cui possono corrispondere più rappresentazioni
- semplifica l'interfaccia nell'aggregato
  - sottrae l'interfaccia delegata all'iterator
- permette di avere più visite contemporaneamente pendenti
  - ciascun iteratore tiene memoria dello stato della sua visita
- ha un forte accoppiamento tra iteratore concreto e aggregato concreto
  - l'aggregato concreto deve rendere pubblica l'interfaccia usata dall'iterator



146/98

- **java.util.Iterator interface**
  - public boolean hasNext()
  - public <E> next()
  - public void remove()

Methods

| Modifier and Type | Method and Description                                                                                                            |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| boolean           | <code>hasNext ()</code><br>Returns true if the iteration has more elements.                                                       |
| E                 | <code>next ()</code><br>Returns the next element in the iteration.                                                                |
| void              | <code>remove ()</code><br>Removes from the underlying collection the last element returned by this iterator (optional operation). |

- *TBD: un esercizio su collections che mostri l'uso dell'Iterator (SM)*

- Per agevolare l'intervento sulle singole parti, i patterns tendono a rendere complesso l'impianto di insieme
  - aumentano il rischio di over-design
- + riducono l'accoppiamento tra classi e oggetti
  - + congelano nel modello statico e nelle interfacce di un sistema meccanismi di creazione, strutturazione e comportamento degli oggetti che realizzano requisiti del sistema stesso indipendentemente dai dettagli della programmazione
  - + creano un linguaggio di discussione che eleva il livello di astrazione e sintesi nella descrizione di una struttura del codice
  - + aumenta il peso della progettazione rispetto alla implementazione

- [GOF95] E.Gamma, R.Helm, R.Johnson, J.Vlissides, "Design Patterns: elements of reusable object oriented software," Addison Wesley, 1995.
- Franco Guidi Polanco, "GOF's design patterns in Java," 2002.  
<http://eii.ucv.cl/pers/guidi/designpatterns.htm>.
- Joshua Bloch, "Effective Java," Addison Wesley, 2008.
- James W.Cooper, "The Design Patterns Java companion," Addison Wesley, 1998.
- [http://sourcemaking.com/design\\_patterns](http://sourcemaking.com/design_patterns)
- <http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns>

# Java Collections & Iterator Pattern

## Software Engineering Course

Leonardo Scommegna

[leonardo.scommegna@unifi.it](mailto:leonardo.scommegna@unifi.it)

Università degli Studi di Firenze  
Software Technologies Lab



# Outline

## 1 Iterator Pattern

- Intent
- An Incremental Example
- The Solution

## 2 Java Collection Framework

- Introduction
- Parametric Polymorphism and Java Generics
- JCF Interfaces and Implementations
- Iterating Over Collections
- Comparable<T> and Comparator<T> Interfaces

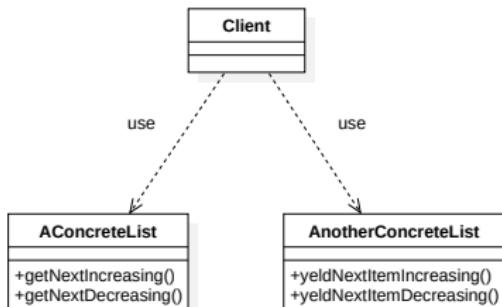


# Iterator Pattern



# The Problem

- Aggregated objects (i.e., collections) usually provide a way to **sequentially access** their elements
- Issues:
  - Traversal implementation is **strictly coupled** to the aggregated object
  - What if multiple traversal criteria are required? e.g., pre-order and post-order
  - In case of multiple types of aggregated objects, you have to know in advance how each type implements the traversal to use it



```
1 // Client Code for incremental traversal
2 if(aggregatedObj instanceof AConcreteList)
3 nextElement = aggregatedObj.
4 getNextIncreasing();
5 else
6 nextElement = aggregatedObj.
7 yieldNextItemIncreasing();
```



# Iterator Pattern, the Solution

- **Iterator pattern** provides a solution:

- getting out from the aggregated object the responsibility to provide a traversing mechanism (decoupling)
- developing multiple criteria keeping the aggregated class clean and modular
- preventing the underlying representation exposure
- providing a uniform interface for traversing different aggregate classes (polymorphic iteration)



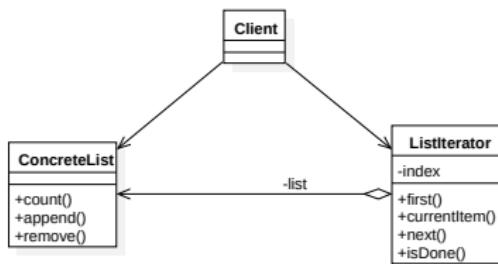
# A First Step Towards the Pattern



- Shifting the **responsibility** of the sequential access to another class
- `first()` initialize the index (reset index)
- `currentItem()` keeps track of the current element of `ConcreteList` (index)
- `next()` advances the current element to the next element (modify index)
- `isDone()` test whether the last element was reached
- This allows the implementation of **multiple traversal strategies**



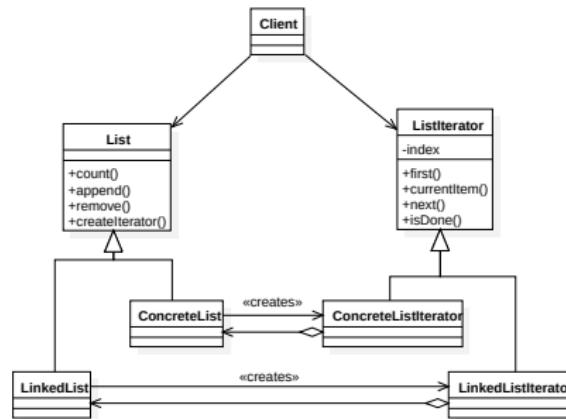
# Issues of the Current Implementation



- **ConcreteList** and **ListIterator** are **strongly coupled**:
  - The client has to know how the Iterator is built and which type of class it is designed for
  - What if the aggregated class type can change dynamically?
  - What if another Iterator implementing a different traversal criteria is written with different methods names?
  - The instantiation of an iterator depends on the type of aggregated classes

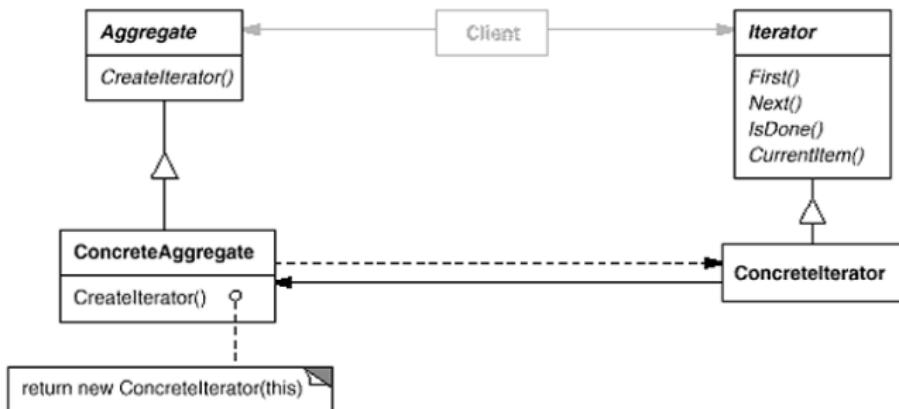


# Refining the Solution



- List and Iterator define a **common interface** for both the aggregated class and the iterator
- ConcreteList is now responsible for the instantiation of the corresponding concrete iterator through the `createIterator()` method
- This defines a neat decoupling between the aggregated object and the iterator

# The General Structure



# Java Collection Framework



# Java Collection Framework (JCF)

- A **collection** is an object that represents a group of objects
- Java Collection Framework (JCF) is a **unified architecture** for representing and manipulating collections
  - Enables collections to be manipulated independently of implementation details.
  - Mainly packaged within `java.util`



# Primary advantages of JCF

- **Reduces programming effort:**

- By providing built-in data structures (e.g., list, set, map) and algorithms (e.g., sorting, searching)
- By not requiring to produce *ad-hoc* collections APIs

- **Increases performance**

- By providing high-performance implementations of data structures and algorithms

- **Provides interoperability** between unrelated APIs:

- By establishing a common language to pass collections

- **Fosters software reuse:**

- by providing a standard interface to manipulate collections and algorithms



# Elements of JCF

- **Collection Interfaces:**

- Represent different types of collections e.g., sets, lists, and maps
- Allow to manipulate concrete implementations independently

- **Implementations**

- Various concrete interfaces implementations
- e.g., `List<E>` interface is implemented by the `ArrayList<E>`, `LinkedList<E>` and `Vector<E>` concrete classes

- **Algorithms**

- Static methods that perform useful functions on collections, such as sorting a list
- Same methods appear in different interface implementations: *polymorphic algorithms*

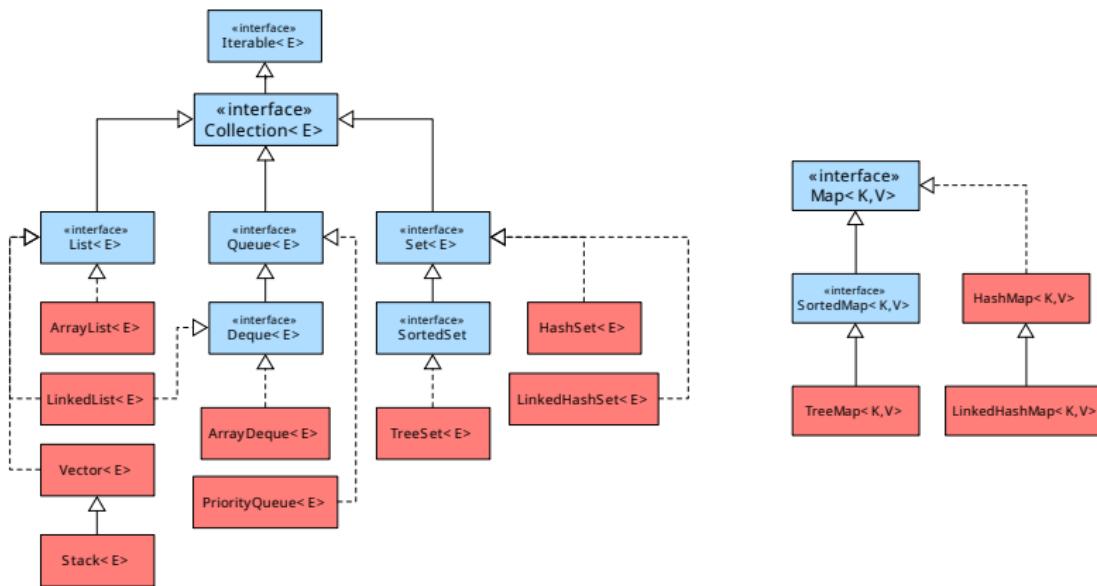


# JCF two Main Interfaces Groups

- Interfaces Derived from `java.util.Collection<E>` interface:
  - `java.util.Set<E>`
  - `java.util.SortedSet<E>`
  - `java.util.NavigableSet<E>`
  - `java.util.List<E>`
  - `java.util.Queue<E>`
  - `java.util.Deque<E>`
  - `java.util.concurrent.BlockingQueue<E>`
  - `java.util.concurrent.TransferQueue<E>`
- Interfaces Derived from `java.util.Map<K, V>` interface:
  - `java.util.SortedMap<K, V>`
  - `java.util.NavigableMap<K, V>`
  - `java.util.concurrent.ConcurrentMap<K, V>`
  - `java.util.concurrent.ConcurrentNavigableMap<K, V>`



# JCF two Main Interfaces Groups



# Parametric Polymorphism in JCF

- **Parametric Polymorphism**

- Another kind of Polymorphism in Java
- It allows methods and data types to be written in a **generic** way
- Still preserving full **static type-safety**
- Doing so, they can handle values **without depending** on their types:
  - `List<E>` represents a generic list of elements of type E
  - `add(E element)` declares a method able to handle a generic parameter of type E

- JCF can represent collections of **any kind** of reference data type:
  - Any kind of `List<E>` e.g., `List<String>`, `List<Integer>`, `List<Student>`
  - Any kind of `Set<E>` e.g., `Set<Double>`, `Set<String>`, `Set<Book>`
  - Any kind of `Map<K,V>` e.g., `Map<String, Integer>` `Map<String, Exam>`



# Java Generics Types

- Java realizes the parametric polymorphism through **Java Generic Types** (a.k.a. Generics)
  - Introduced in JDK 5.0 with the aim of reducing unsafe code and adding an extra layer of abstraction over types
- Generic classes or interfaces parameterized over reference types
- Many benefits over non-generics code:
  - Stronger type checks at compile time
  - Elimination of casts
  - Allows to implement **generic algorithms** and avoid code duplication

```
1 // a collection (i.e., List) without generics
2 List list = new ArrayList(); // raw type:
 // elements of type Object
3 list.add("hello"); // no type check
4 String s = (String) list.get(0); // casting required
5
6
7 // a collection (i.e., List) with generics
8 List<String> list = new ArrayList<String>();
9 list.add("hello"); // strong type check
10 String s = list.get(0); // no cast
```



# Avoiding Code Duplication with Java Generics Types

```
1 // non-generic class: only String can be wrapped
2 // by this class
3
4 public class StringWrapper {
5 private String value;
6 public StringWrapper(String value) {
7 this.value = value; }
8 public void setValue(String value) {
9 this.value = value; }
```

```
1 // another non-generic class: only Integer can
2 // be wrapped by this class
3
4 public class IntegerWrapper {
5 private Integer value;
6 public IntegerWrapper(Integer value) {
7 this.value = value; }
8 public void setValue(Integer value) {
9 this.value = value; }
```



# Avoiding Code Duplication with Java Generics Types

```
1 // non-generic class: only String can be wrapped
2 // by this class
2 public class StringWrapper {
3 private String value;
4 public StringWrapper(String value) {
5 this.value = value; }
6 public void setValue(String value) {
7 this.value = value; }
```

```
1 // another non-generic class: only Integer can
2 // be wrapped by this class
2 public class IntegerWrapper {
3 private Integer value;
4 public IntegerWrapper(Integer value) {
5 this.value = value; }
6 public void setValue(Integer value) {
7 this.value = value; }
```



```
1 // a generic class able to wrap any type of object
2 public class Wrapper<T> {
3 private T value ;
4 public Wrapper(T value) {
5 this.value = value; }
6 public void setValue(T value) {
7 this.value = value; }
```



# Using Java Generics Types

- **Defining a Generic Type:**

- `access_modifier class/interface name<T1,T2,...,Tn> {...}`
- $T_1, T_2, \dots, T_n$  are class parameters and can be used anywhere inside the class/interface as placeholder for the actual type

- **Declaring a Generic Type:**

- Means replacing the type parameter with a concrete type
- `Wrapper<String> stringWrapper;`

- **Instantiating a Generic Type:**

- Means calling the constructor method using the ‘‘new’’ keyword and a concrete value for the type parameter
- `Wrapper<String> stringWrapper = new Wrapper<String>("foo");`



# More on Generics

- **Definition of a Generic Type:**

```
1 // a generic class able to wrap any type of object
2 public class Wrapper<T> {
3 private T value ;
4 public Wrapper(T value) {
5 this.value = value; }
6 public void setValue(T value) {
7 this.value = value; }
```

- **Declaration and Instantiation:**

```
1 // Declarations
2 Wrapper<String> stringWrapper;
3 Wrapper<Integer> integerWrapper;
4
5 // Instantiations
6 stringWrapper = new Wrapper<String>("foo");
7 integerWrapper = new Wrapper<>(1); // diamond syntax
8
9 String s = stringWrapper.getValue(); // returns a String
10 int i = integerWrapper.getValue(); // returns an int
```



# Type Erasure

- Generics provide type check at compile time while supporting generic programming
- To implement generics, compiler applies the "**Type Erasure**" to avoid runtime overhead
  - Replace all generic types with their bound types or Object type if the type parameters are unbounded. Thus, produced bytecode will contain only ordinary classes, interfaces, and methods
  - Insert **type casts** if necessary preserving type safety
  - Generate **bridge methods** to preserve polymorphism in extended generic types
- the type erasure process erases all the type parameters and replaces them with their **bound**.



# Unbounded Type Parameters

- Unbounded Type parameter T:

```
1 public class Record<T> {
2 private T data;
3 private Record<T> next;
4
5 public Record(T data, Record<T> next) {
6 this.data = data;
7 this.next = next;
8 }
9 public T getData() { return data; }
10 // ...
11 }
```

- Type Parameter T Replaced with Object by the compiler:

```
1 public class Record {
2 private Object data;
3 private Record next;
4
5 public Record(Object data, Record next) {
6 this.data = data;
7 this.next = next;
8 }
9 public Object getData() { return data; }
10 // ...
11 }
```



# Bounded Type Parameters

- Type parameter T bounded with Comparable Type through ‘‘extends’’:

```
1 public class Record<T extends Comparable<T>> {
2 private T data;
3 private Record<T> next;
4
5 public Record(T data, Record<T> next) {
6 this.data = data;
7 this.next = next;
8 }
9 public T getData() { return data; }
10 // ...
11 }
```

- Type Parameter T Replaced with Comparable Type by the compiler:

```
1 public class Record {
2 private Comparable data;
3 private Record next;
4
5 public Record(Comparable data, Record next) {
6 this.data = data;
7 this.next = next;
8 }
9 public Comparable getData() { return data; }
10 // ...
11 }
```



# Type Erasure in Methods

- Compiler also erases parameter types in generic methods:

```
1 public static int count(T[] array, T itemToCount) {
2 int counter = 0;
3 for (T item : array)
4 if (item.equals(itemToCount))
5 ++counter;
6 return counter;
7 }
```

- T is unbounded then it is replaced with Object:

```
1 public static int count(Object[] array,
2 Object itemToCount) {
3 int counter = 0;
4 for (Object item : array)
5 if (item.equals(itemToCount))
6 ++counter;
7 return counter;
8 }
```



# A Subtyping Issue with Type Erasure

- Erasure process could result in **polymorphism violation**

```
1 public class Record<T> {
2 public T data;
3 public Record(T data) { this.data = data; }
4 public void setData(T data) {
5 System.out.println("RECORD setData");
6 // other optional operations
7 this.data = data;
8 }
9 }
10
11 public class MyIntegerRecord extends Record<Integer> {
12 public MyIntegerRecord(Integer data) { super(data); }
13 public void setData(Integer data) {
14 System.out.println("MyRECORD setData");
15 // other optional operations
16 super.setData(data);
17 }
18 }
```

- After type erasure method signatures do **not match**
- `Record.setData(Object)`  $\neq$  `MyIntegerRecord.setData(Integer)`

# The Bridge Methods as a Solution

- Compiler creates synthetic methods called **bridge methods** to preserve polymorphism

```
1 class MyIntegerRecord extends Record {
2
3 // Bridge method generated by the compiler
4 public void setData(Object data) {
5 setData((Integer) data);
6 }
7
8 public void setData(Integer data) {
9 System.out.println("MyRECORD setData");
10 // other optional operations
11 super.setData(data);
12 }
13
14 // ...
15 }
```



# Generics Tricky Mistake

- Consider the code:

```
1 MyIntegerRecord myRec = new MyIntegerRecord(5);
2 Record rec = myRec;
3 rec.setData("test"); // This is syntactically correct
4 Integer x = myRec.getData();
```

- After type erasure:

```
1 MyIntegerRecord myRec = new MyIntegerRecord(5);
2 Record rec = (MyIntegerRecord) myRec;
3 rec.setData("test"); // Causes a ClassCastException
4 Integer x = (String)myRec.getData();
```

- exception caused by bridge method in MyIntegerRecord:

```
1 // Bridge method
2 public void setData(Object data) {
3 setData((Integer) data); // trying to cast String
4 to Integer
5 }
```

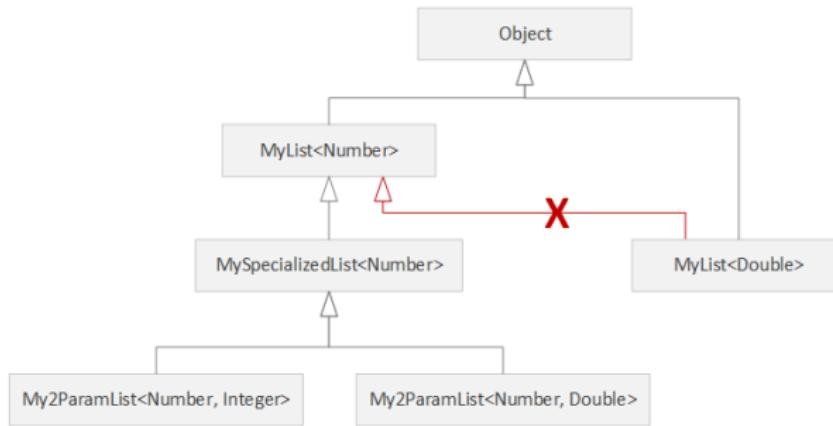


# Generic Parameters Subtypes

- Polymorphic generic arguments do not imply polymorphic generic classes

```
1 public class MyList<T> {}
2
3 public class MySpecializedList<T> extends MyList<T> {}
4
5 public class My2ParamList<T, S> extends MySpecializedList<T> {}
```

- Resulting inheritance structure:

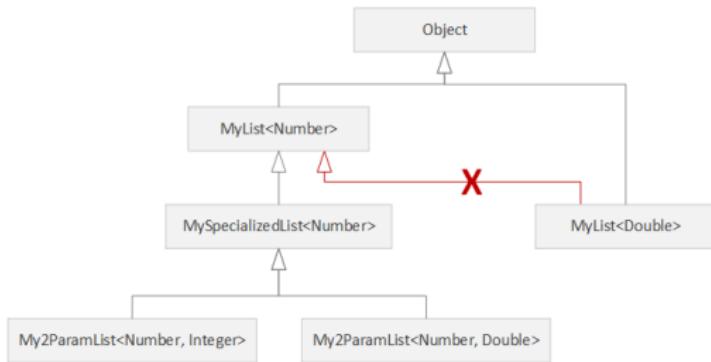


# Generic Parameters Subtypes: an Example

```
1 public class MyList<T> {
2
3 public void insert(T value) {
4 // ... insert the value ...
5 }
6 }
7
8 MyList<Number> numberList = new MyList<>();
9 numberList.insert(new Integer(7));
10 numberList.insert(new Double(5.72));
```

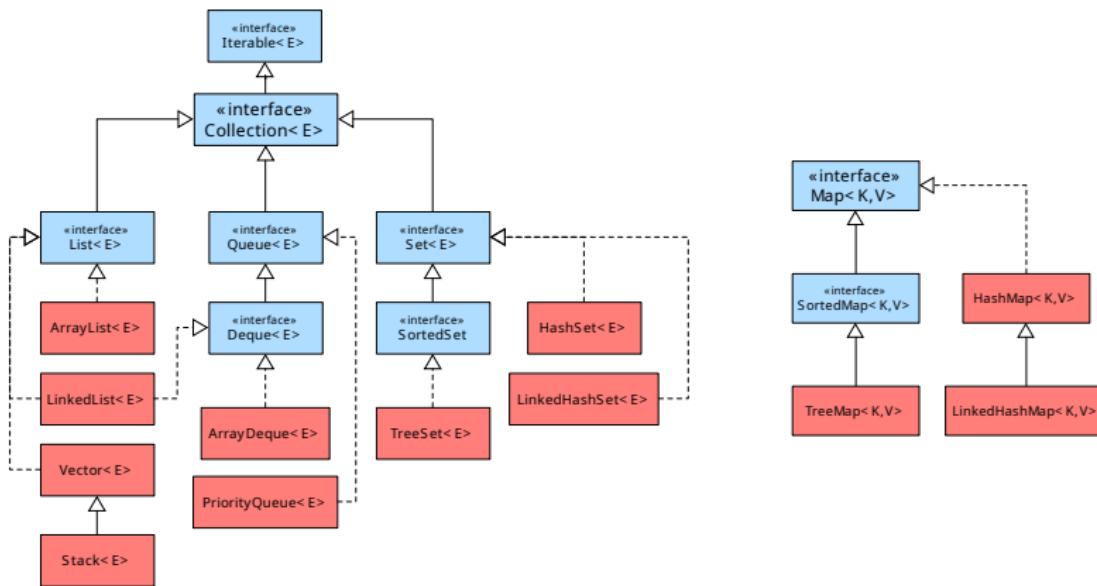
- `MyList<Double>` is a subtype of `MyList<Number>` if and only if is substitutable for any instance of `MyList<Number>` (Liskov Substitution Principle)
- **Liskov principle:** Let  $\phi(x)$  be a property provable about objects  $x$  of type  $T$ . Then  $\phi(y)$  should be true for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ .
- But this is not the case since: `MyList<Number>.insert(Integer)` is allowed while `MyList<Double>.insert(Integer)` is not

# What about the bottom of the inheritance structure?

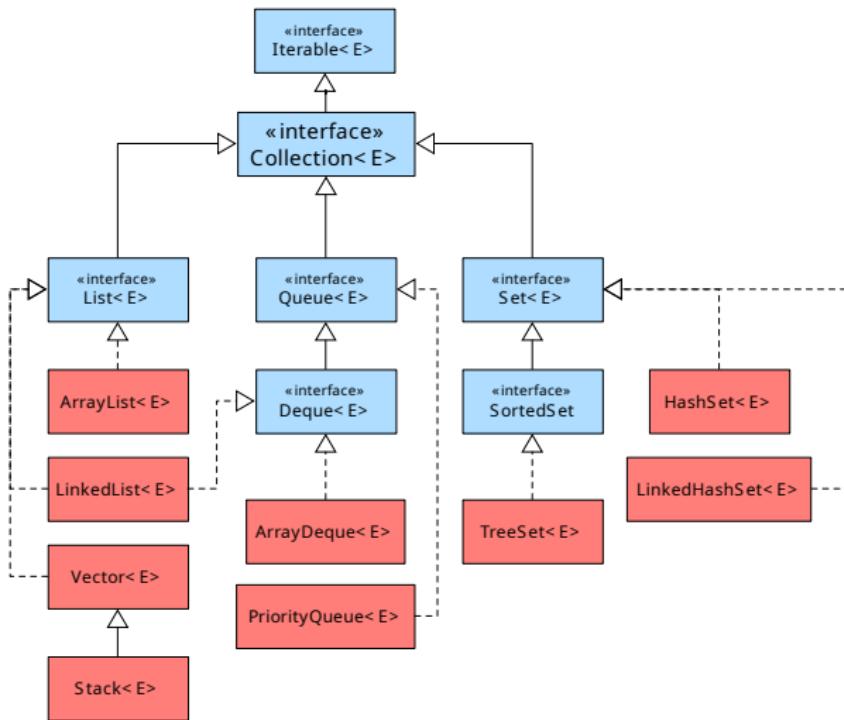


- A generic class including additional generic parameters is a subtype of another generic class as long as the former class **extends** the latter and the **shared generic parameters match**
- My2ParamList<T, S> is a subtype of MySpecializedList<T> if the shared generic parameter T is the same type
- Not shared generic parameters, such as S, **can vary independently**
- E.g. both My2ParamList<Number, Integer> and My2ParamList<Number, Double> are subtypes of MySpecializedList<Number>

# Getting Back to JCF



# Collection<E> Hierarchy Tree



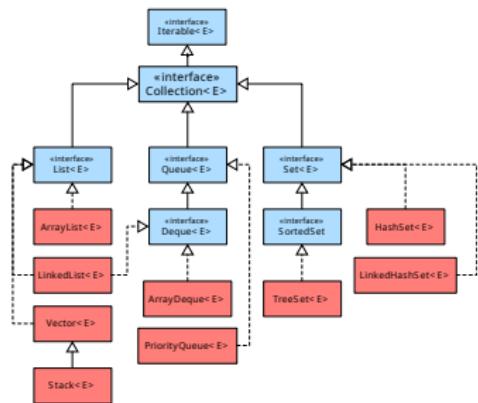
# Collection<E> Interface

- Collection<E> represents a **generalized grouping of objects of Type E**
- One of the two **main root** of JCF
- Defines features shared by all implementing classes e.g., adding/removing elements
- Can also be used in the code when maximum flexibility is needed allowing to employ any concrete implementation (except than Map<K, V>):

```

1 // Declaring object of type
 collection
2 Collection<String> c;
3
4 c = new ArrayList<>();
5 ...
6 c = new HashSet<>();

```



# Collection<E> Interface: Basic Operations

- `add(E element)` adds an element to the collection, returns true if the collection changes (*optional operation*)
- `remove(Object element)`<sup>1</sup> removes an element from the collection, returns true if the collection changes (*optional operation*)
- `contains(Object element)`<sup>1</sup> checks if the collection contains an element
- `size()` returns the number of elements
- `isEmpty()` returns true if no elements are found
- `iterator()` just another way to iterate over a collection (more on this later)



---

<sup>1</sup>Note that element is an Object because the method relies on the `equals(Object element)` method

## Collection<E> Interface: Operations on entire Collections

- `addAll(Collection<? extends E> c)` adds all of the elements in the specified collection to this collection (*optional operation*)
- `removeAll(Collection<?> c)` removes all of this collection's elements that are also contained in the specified collection (*optional operation*)
- `containsAll(Collection<?> c)` returns true if this collection contains all the items in the specified collection
- `retainAll(Collection<?> c)` removes from this collection all of its elements that are not contained in the specified collection (*optional operation*)
- `clear()` removes all elements within this collection

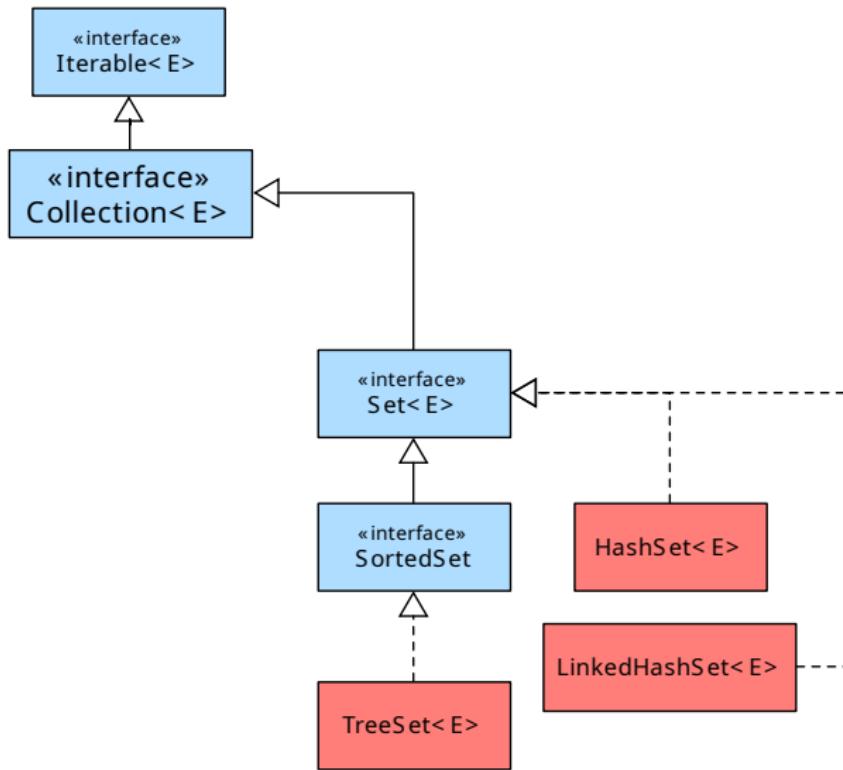


# Collection<E> Interface: Optional Operations

- Implementations of the previous methods are concretely provided by derived classes
- Derived classes are permitted to not perform one or more so-called *optional operations*, throwing an `UnsupportedOperationException` if called
- This design choice is controversial and faces a trade-off:
  - It prevents **static (compile time) type checking**: if the instance type does not support a certain method, the exception will be thrown only at runtime (this is bad!)
  - It also prevents an **explosion in the size** of the interface hierarchy (this is good!)



# Collection<E> Derived Classes: Set<E>



# Set<E> Interface

- Set<E> represents a collection that does not allow duplicate elements
- Models the mathematical set abstraction
- Set<E> can not contain:
  - null references
  - Same objects for identity (*i.e.*, two references to the same object)
  - Same objects for equality (*i.e.*, two references to two different objects *a* and *b* such as *a.equals(b)*)
- Two Set<E> instances are equal if they contain the same elements

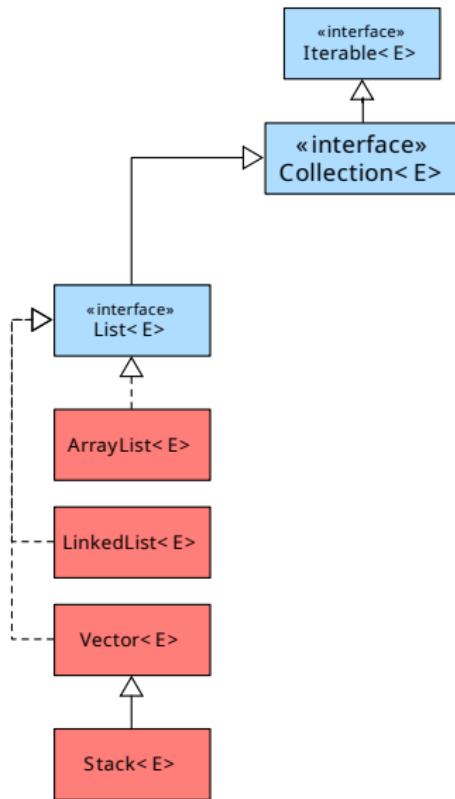


# Set<E> Interface: Some General-Purpose Implementations

- HashSet<E>
  - **Best performing** and most used implementation
  - Elements are stored in a hash table with **no guarantees concerning the order of iteration**
- TreeSet<E>
  - Elements are stored in a red-black tree basing on values following the ordering specified by a Comparator<T> object or according to the natural ordering of Comparable<T> objects
- LinkedHashSet<E>
  - Elements are stored in a hash table with a linked list running through it and ordered basing on insertion



# Collection<E> Derived Classes: List<E>



# List<E> Interface

- List<E> represents an ordered collection that can contain duplicates
  - null references are allowed
  - Two List<E> objects are equal if they contain the same (*equals(...)*) elements in the same order
- A kind of array whose size can change if needed:
  - Each element has a position in the list accessed by an index from 0 to `list.size()-1`



# List Methods

- Some noteworthy **inherits methods** from Collection<E>:
  - add(E e) and remove(Object e) methods are defined to append to the end of the list and to remove the first occurrence of an element, respectively
  - addAll(Collection<? extends E> c) append all the elements of the passed collection to the end of the current list
  - removeAll(Collection<?> c) remove from the current list all the elements that are contained in the passed collection
- Some **specific operators**:
  - get(int index) returns the element at the specified position in this list
  - set(int index, E element) replaces the element at the specified position in this list with the specified

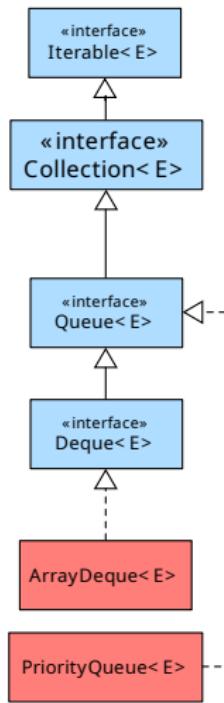


# List<E> Interface: Some General-Purpose Implementations

- `ArrayList<E>`
  - **Best performing** and most used implementation
- `LinkedList<E>`
  - Doubly linked list implementation
- `Vector` and `Stack`
  - Legacy and deprecated implementations and should not be used



# Collection<E> Derived Classes: Queue<E> & Deque<E>



# Queue<E> Interface

- Queue<E> represents an ordered fixed-size collection that can contain duplicates
- Usually used to hold multiple elements prior to processing
- Provides additional insertion, extraction, and inspection operations:
  - `add(E e)` and `offer(E e)` to add an element
  - `remove()` and `poll()` to remove an element at the head
  - `element()` and `peek()` to inspect an element at the head without removing it
- Does not allow the insertion of null elements while duplicated ones are allowed
- Typically elements are ordered in a FIFO manner: all new elements are inserted at the tail of the queue

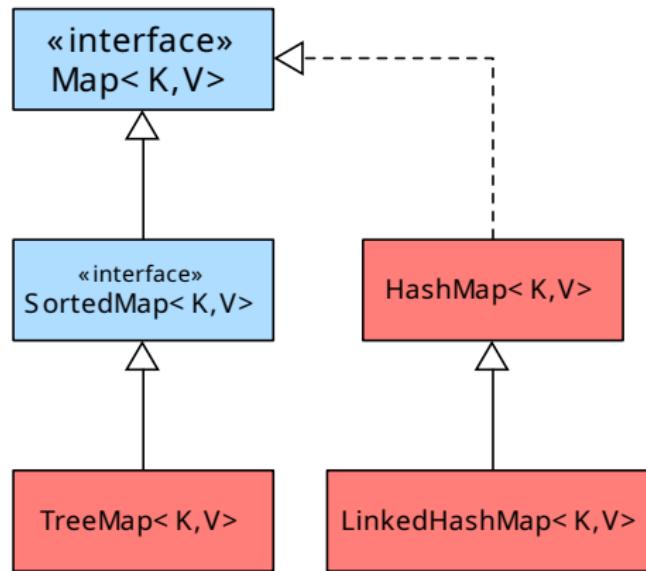


# Deque<E> Interface

- Deque<E>, as subtype of the Queue<E> interface, represents a double-ended queue
- Elements can be inserted, retrieved and removed at both ends
- Provides additional insertion, extraction, and inspection operations:
  - addFirst(E e), offerFirst(E e), addLast(E e), offerLast(E e) to add an element to the head/tail
  - removeFirst(), pollFirst(), removeLast(), pollLast() to remove an element at the head/tail
  - getFirst(), peekFirst(), getLast(), peekLast() to inspect an element at the head/tail
- Can be used both as FIFO and LIFO



# Map<E> Interface



# Map<K, V> Interface

- Map<K, V> represents an object that maps keys into values **generalized grouping of objects of Type E**
- One of the two **main root** of JCF
- Models the mathematical function abstraction  $K \rightarrow V$ :
  - K represents the type of keys held by the map
  - V represents the type of the values that the keys are mapped to
  - A key can map to at most one value
- Cannot contain duplicate keys: if the map previously contained a mapping for the key, the old value is replaced by the specified value
- Two Map<K, V> instances are equal if they represent the same key value mapping



# Map<K, V> Interface: Basic Operations 1/2

- `put(K key, V value)` inserts an entry in the map
- `remove(Object key)` deletes an entry for the passed key
- `get(Object key)` returns the object that contains the value associated with the key
- `isEmpty()` returns true if the map is empty
- `size()` returns the number of entries in the map
- `clear()` resets the map
- `containsKey(Object key)` returns true if some key equal to the passed key exists within the map
- `containsValue(Object value)` returns true if some value equal to the passed value exists within the map



# Map<K,V> Interface: Basic Operations 2/2

- `keySet()` returns the `Set<E>` view containing all the keys
- `values()` returns a `Collection<E>` view of the values contained in the map
- `entrySet()` returns the `Set<E>` view containing all the keys and values



# Map<K, V>: Some General-Purpose Implementations

- `HashMap<K, V>`
  - Most used implementation
- `TreeMap<K, V>`
  - Elements are stored following the ordering specified by a `Comparator<T>` object or according to the `Comparable<T>` natural ordering of its keys
- `LinkedHashMap<E>` ...

```
1 Map<String, Integer> map = new HashMap<String, Integer>();
2 map.put("one", 1);
3 map.put("two", 2);
4 int value = map.get("one"); // returns the Integer referred by the key "one"
5
6 // returns a Set of keys (i.e., [one, two, three])
7 Set<String> keys = map.keySet();
8
9 // returns a Collection of values (i.e., [1, 2, 3])
10 Collection<Integer> values = map.values();
11
12 // returns a Set of entries (i.e., [one=1, two=2, three=3])
13 Set<Entry<String, Integer>> entrySet = map.entrySet();
```



# Useful Static Methods

- `Set<E>`, `List<E>`, and `Map<K,V>` provide some useful static methods to create **on-the-fly immutable** sets, lists, and maps, respectively
- `Set.of(E...)`: provides a convenient way to create immutable sets
- `List.of(E...)`: provides a convenient way to create immutable lists
- `Map.of(K,V)`: provides a convenient way to create immutable maps

```
1 List<String> list = List.of("Hello", "world");
2
3 Set<String> set = Set.of("Hello", "world");
4
5 Map<String, Integer> map = Map.of("First", 1, "Second", 2);
```



# Iterating Over Collections

- Using for/while loops:

```
1 List<String> days = List.of("Monday", "
2 Tuesday", "Wednesday",
3 "Thursday", "Friday", "Saturday", "Sunday"
4);
5
6 for(int i = 0; i < days.size(); i++){
7 System.out.println(days.get(i));
8 }
```

- Using foreach loop:

```
1 for(String day : days) {
2 System.out.println(day);
3 }
```

- Using an Iterator<E>: more on the next slides



# Iterating Over Collections: the Iterator<E>

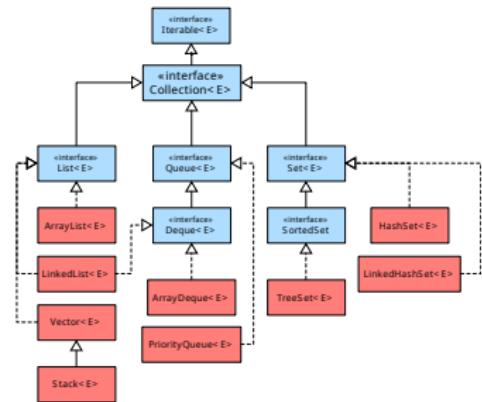
- `Iterator<E>`: interface which belongs to JCF that allows to traverse a collection and access/remove data elements
- Each collection can obtain an instance of `Iterator<E>` over its elements through the `iterator()` method inherited by `Collection<E>` from the `Iterable<T>` interface
- While there are elements in the collection, the `hasNext()` method of the iterator returns true; calling the `next()` method returns the next element of the collection

```
1 List<String> days = List.of("Monday", "Tuesday", "Wednesday",
2 "Thursday", "Friday", "Saturday", "Sunday");
3
4 // iterator
5 for(Iterator<String> i = days.iterator(); i.hasNext();) {
6 System.out.println(i.next());
}
```



# JCF Iterator Pattern Implementation

- Note that `Collection<E>` implements the **Iterator pattern**
- Represent the **aggregated class** implementing the `Iterable<E>` interface
- Each collection class provides an `iterator()` method that returns an iterator to access each element of the collection



# Form Arrays to Collections and vice-versa

- `Array.asList(T ...a)` static method that returns a fixed-size list backed by the specific array
- `toArray()` is a method declared in the `Collection<E>` interface that returns an `Object[]` array containing all of the elements of a specified collection

```
1 List<String> days = List.of("Monday", "Tuesday", "Wednesday",
2 "Thursday", "Friday", "Saturday", "Sunday");
3
4 // from a list of strings to an array of objects
5 Object[] daysArray = days.toArray();
6
7 // ... or making the type of the array explicitly...
8 String[] daysArray = (String[]) days.toArray(new String[0]);
9
10 // from array to list
11 days = Arrays.asList(daysArray);
```



# Collections: an Utility Class

- Provides **static methods for polymorphic algorithms** that operate on or return collections
- `Collections.sort(List<T>)` and `Collections.sort(List<T>, Comparator<T>)` sort the specified list into ascending order, according to the natural ordering of its elements (which must implement the `Comparable<T>` interface) or to the order induced by a specified `Comparator<T>`
- `Collections.reverse(List<?> list)` reverses the order of the elements in the specified list
- `Collections.max(...)/min(...)` return maximum/minimum elements of the given collection, according to the natural ordering of its elements (which must implement the `Comparable<T>` interface) or to the order induced by a specified `Comparator<T>`



# Sorting with the Comparable<T> Interface

- Comparable<T>: an interface that requires the implementation of the (only) method `compareTo(T o)`
- Imposes a total ordering on the objects of each classes that implements it
- And this ordering is referred to as the **class's natural ordering** while the method `compareTo(T o)` is referred to as its **natural comparison method**
- `compareTo(T o)` should return a negative integer, zero, or a positive integer as the current object is less than, equal to, or greater than the passed object
- Implementing Comparable<T> allows:
  - calling `Collections.sort(List<T> list)`
  - calling `Arrays.sort([])`
  - ordering `TreeMap<E>` elements by value
  - ordering `TreeSet<E>` elements by value



# Sorting with the Comparable<T> Interface

- Definition of a Comparable Class:

```
1 public class Person implements Comparable<Person> {
2 private String taxcode;
3 private String name;
4 private String surname;
5
6 // defines a natural ordering between objects of type Person,
7 // based on taxcode
8 public int compareTo(Person o) {
9 // exploits the compareTo implementation provided by
10 String
11 return this.taxcode.compareTo(o.taxcode);
12 }
13 }
```

- Usage:

```
1 // an unordered set
2 Set<Person> people = new HashSet<Person>();
3 people.add(new Person("RSS", "MARIO", "ROSSI"));
4 people.add(new Person("BNC", "PAOLO", "BIANCHI"));
5
6 // an ordered set
7 Set<Person> orderedPeople = new TreeSet<Person>(people);
```



# Sorting with the Comparator<T> Interface

- Comparator<T>: interface that requires the implementation of the method `compare(T o1, T o2)`
- `compare(T o1, T o2)` method compares two arguments and returns a negative integer, zero, or a positive integer as the first argument is less than, equal or greater than the second object
- Implementing Comparator<T> allows to:
  - Define multiple ordering criteria
  - Precise control over the sort order, such as `Collections.sort(List<T>, Comparator<T>)` or `Arrays.sort(T[], Comparator<T>)`
  - Control the order of some data structures, such as sorted sets/maps (e.g., `TreeSet<E>`, `TreeMap<K, V>`)
  - Provide an ordering for collections of objects that don't have a natural ordering



# Sorting with the Comparator<T> Interface

- **Definition of Comparable Classes:**

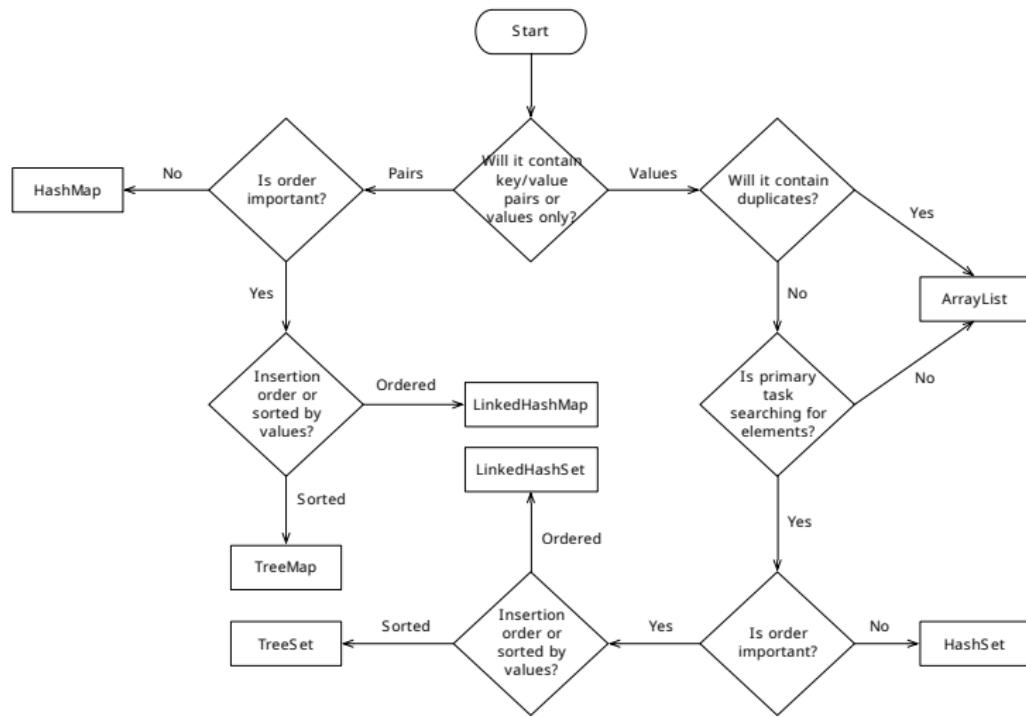
```
1 // a concrete comparator of Person objects, based on surname
2 public class PersonComparatorBySurname implements Comparator<Person> {
3 public int compare(Person o1, Person o2) {
4 // exploits the compareTo implementation provided by String
5 return o1.surname.compareTo(o2.surname);
6 }
7 }
8 // another concrete comparator of Person objects, based on name
9 public class PersonComparatorByName implements Comparator<Person> {
10 public int compare(Person o1, Person o2) {
11 // exploits the compareTo implementation provided by String
12 return o1.name.compareTo(o2.name);
13 }
14 }
```

- **Usage:**

```
1 // calling code
2 Set<Person> people = new HashSet<Person>(); // an unordered set
3 people.add(new Person("RSS", "MARIO", "ROSSI"));
4 people.add(new Person("BNC", "PAOLO", "BIANCHI"));
5 // an empty tree set, sorted according to the specified comparator, is first
6 // instantiated
7 TreeSet<Person> orderedBySurname = new TreeSet<Person>(
8 new PersonComparatorBySurname());
9 // the empty tree set is filled with all the elements of an unordered set
10 orderedBySurname.addAll(people); // the set is now ordered by Person.surname
11 // a new tree set with a different ordering criterion (i.e., by name)
12 TreeSet<Person> orderedByName = new TreeSet<Person>(
13 new PersonComparatorByName());
14 orderedByName.addAll(people); // the set is now ordered by Person.name
```



# Collections Summary



# References

- former slides by professor Fulvio Patara
- Design patterns: elements of reusable object-oriented software, Gamma, Helm, Johnson and Vlissides
- Java Documentation
- DZone article:  
<https://dzone.com/articles/how-do-generic-subtypes-work>



# Software Testing Principles and Java Testing Tools

## Software Engineering Course

Leonardo Scommegna

[leonardo.scommegna@unifi.it](mailto:leonardo.scommegna@unifi.it)

Università degli Studi di Firenze  
Software Technologies Lab



# Outline

- 1 Introduction
- 2 xUnit/JUnit
- 3 Object Oriented Testing
  - Tests Granularity
  - A Structural Example: Unit Tests
  - A Functional example: Generating Test Cases From Use Cases



# Introduction



# Software Testing Definition

- “*The process of analyzing a software item to detect the differences between actual and required conditions (that is, defects) and to evaluate the software item features*”<sup>1</sup>
- Verification Method aimed at **identifying defects** of a *System Under Test* (SUT) through its execution and comparing the actual behavior with the expected
- An individual instance of test is also called **test case**
- As opposed to static inspection, it is a **dynamic approach** that requires the execution of the source code
- “*Testing can prove the presence of defects, but not their absence*”  
(E.Dijkstra, “Notes On Structured Programming,” 1970)

---

<sup>1</sup>ANSI/IEEE 1059 standard

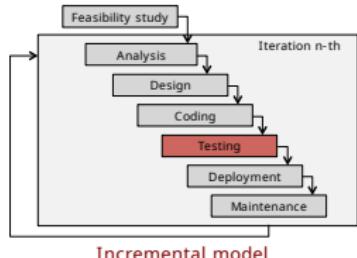


# Software Testing Purposes

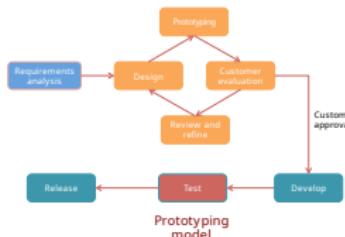
- **Defect detection:** ensures that the code does what is expected at different level of abstraction (e.g., state testing, behavioral testing)
- **Software regression:** supports the process of SUT extension or refactoring
- **Automation:** running all the tests of an application may take some time (hours for big and real life applications), automating this process allows to delegate the execution to a server (continuous integration)
- **Executable documentation:** readable tests act as documentation providing executable example of the application, in addition, some tests will fail when the application will be changed, forcing the developer to fix and keep the suite up to date.



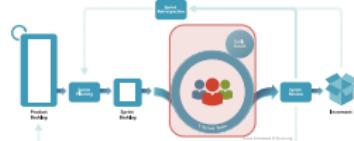
# Software Testing in Development Process



Incremental model

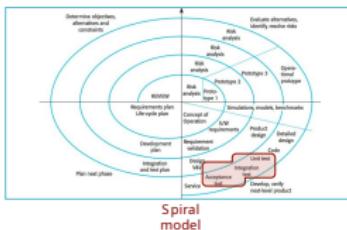


SCRUM FRAMEWORK

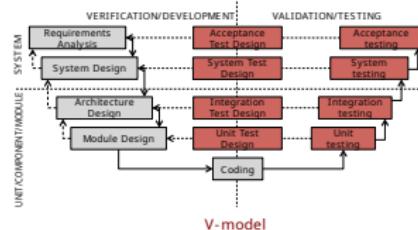


Scrum

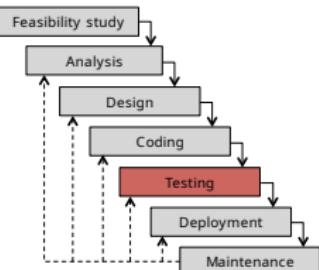
Scrum.org



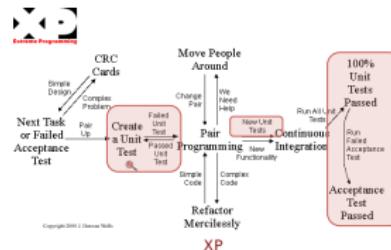
Spiral model



V-model



Waterfall model



SW Testing

# xUnit/JUnit



# xUnit/JUnit

- **xUnit:**

- Collective name for several testing frameworks with similar structure and functionalities
- Many xUnit frameworks exist for various programming languages and development platforms

- **JUnit:**

- Main **Testing Framework** for Java
- Member of the xUnit family with whom it shares the structure and various components
- The “*Unit*” in “*JUnit*” is misleading: is **not only for unit testing**<sup>2</sup>
- There are two major version: JUnit 4 and Junit 5, only Junit 4 is covered here since it is more documented
- This is not about simply learning testing with Java but learn **xUnit workflow** so that you can also test with other languages

---

<sup>2</sup>More on *unit testing* later



# Generic Structure of a Test

- **Setup:**

- Create the environment for the test to be executed
- Instantiate the state of the SUT **before** any testing activities
- This set of preconditions is called **Test Fixture**
- a set of test cases sharing the same fixture is a **Test Suite**

- **Exercise:**

- The test case is executed by the test runner of xUnit/JUnit

- **Verify:**

- verify if the outcome matches the expected behavior of the SUT e.g., checking the outcome of a method or the final state of the SUT

- **Teardown:**

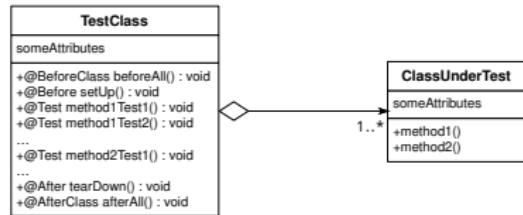
- Clean up the environment bringing it back to the initial state e.g., if a file is modified during the exercise phase, it should be restored afterward

- Test cases should be **independent** of each other: a test case should not rely on side effects of another test case



# Structure of a Test in JUnit

- A test suite is represented by a Java class
- Methods of the class identify:
  - Setup process
  - Test cases
  - Teardown process



# Test Case in JUnit

- Represented by a public void method annotated with @Test (from org.junit.Test)
- The JUnit runner will execute all the tests with **no specific order**

```
1 public class ExampleTestSuite{
2 @Test
3 public void aTest(){
4 ...
5 }
6 @Test
7 public void anotherTest(){
8 ...
9 }
10 }
```



# Test Method in JUnit

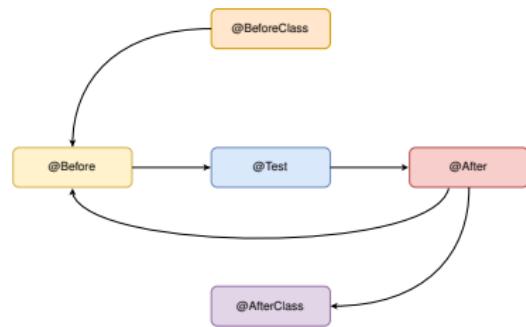
- Should implement both the **Exercise** and the **Verify** step
- Junit provide a bunch of static methods to simplify the Verify step, e.g.:
  - `assertEquals(expected, actual)`
  - `assertTrue(boolean condition)`
  - `.assertNull(expression)`

```
1 @Test
2 public void aTest(){
3 // exercise
4 ...
5 sut.makeSideEffect();
6 ...
7
8 // verify
9 assertEquals(5, sut.getItem());
10 }
```



# Setup and Teardown Phases in JUnit

- Methods annotation for **Setup** phase:
  - @BeforeClass: the method is executed only once and before any other method
  - @Before: the method is executed before each individual test method
  
- Methods annotation for **Teardown** phase:
  - @AfterClass: the method is executed only once and after any other method
  - @After: the method is executed after each individual test method



# Execution Results

- xUnit/JUnit follow the “*all or nothing*” policy:
  - a test case success occurs if and only if the execution completes successfully with all assertions satisfied
  - a test suite success occurs if and only if all test cases complete successfully

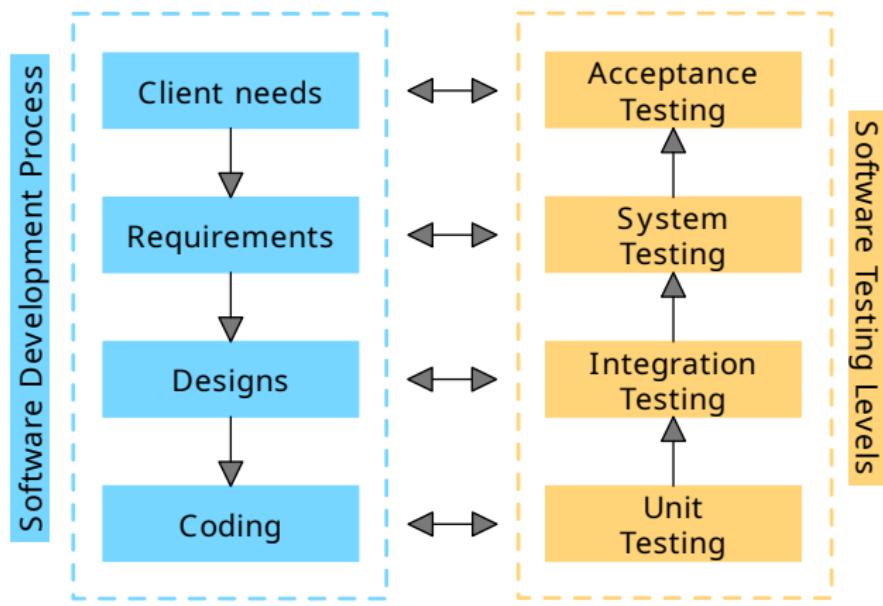


# Object Oriented Testing



# Software Testing Granularity

- Testing is conducted at different stages of the software development process at different levels of granularity

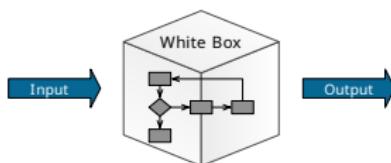


# Software Testing Granularity

- **Unit Testing:** individual functionalities of each unit of software (*i.e.*, classes and methods) are tested in isolation
- **Integration Testing:** single units are combined together to verify whether they still work as expected
- **End-to-End (E2E) Testing:** the whole application is tested: verifying that all components interact correctly. Often called also **System** testing
- **Acceptance Testing:** the system is tested by users and customers to assess whether it is acceptable for delivery



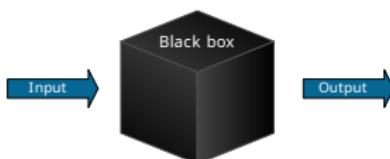
# Unit Testing



- Tests are executed in a **White-Box** Perspective
- A unit test is usually **fast** to execute and it is responsible to verify the behavior of an **individual** software unit (*i.e.*, a class) in isolation
- Tests are based on the deep and detailed knowledge of the **internal logic** of a component's code
- A proper unit testing process requires the support of other techniques:
  - **Mocking:** with *Mockito Framework* for Java
  - **Code Coverage:** with *JaCoCo Library* for Java
  - **Mutation Testing:** with *PIT Framework* for Java



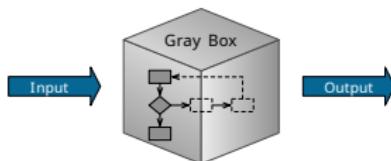
# E2E Testing



- Tests are executed in a **Black-Box Perspective**
- An E2E test is usually **slow** to execute and it is responsible to verify the behavior of whole application relying on the specification
- Tests interact only with the **external interface** of the application (the user interface) ignoring low level details
- A proper E2E testing process requires the support of a **user interaction simulator**:
  - *AssertJ Swing* for Java SE application
  - *Selenium Web Driver* for Java Web Application



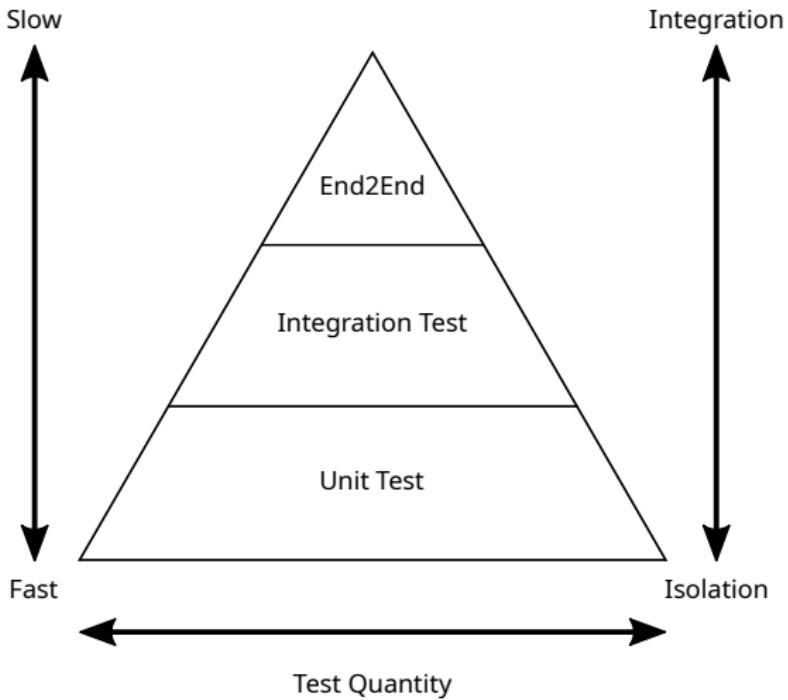
# Integration Testing



- Tests are executed in a **Gray-Box** Perspective
- An integration test is usually **slower** than a unit test but **faster** than a E2E test
- It is responsible to verify the integration between **two or more** software units
- Integration tests interact with the **class interfaces** relying on internal details but not as much as unit tests
- This testing process could require to **mock some components** in order to isolate the units under test

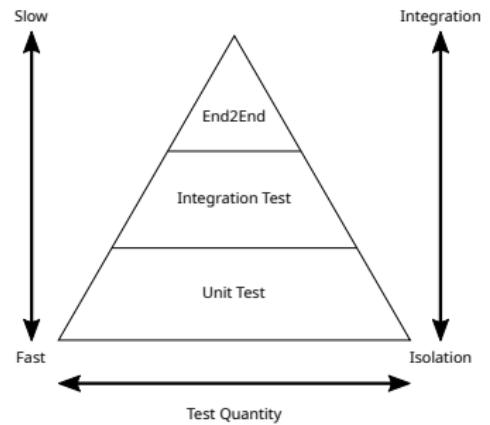


# The Test Pyramid

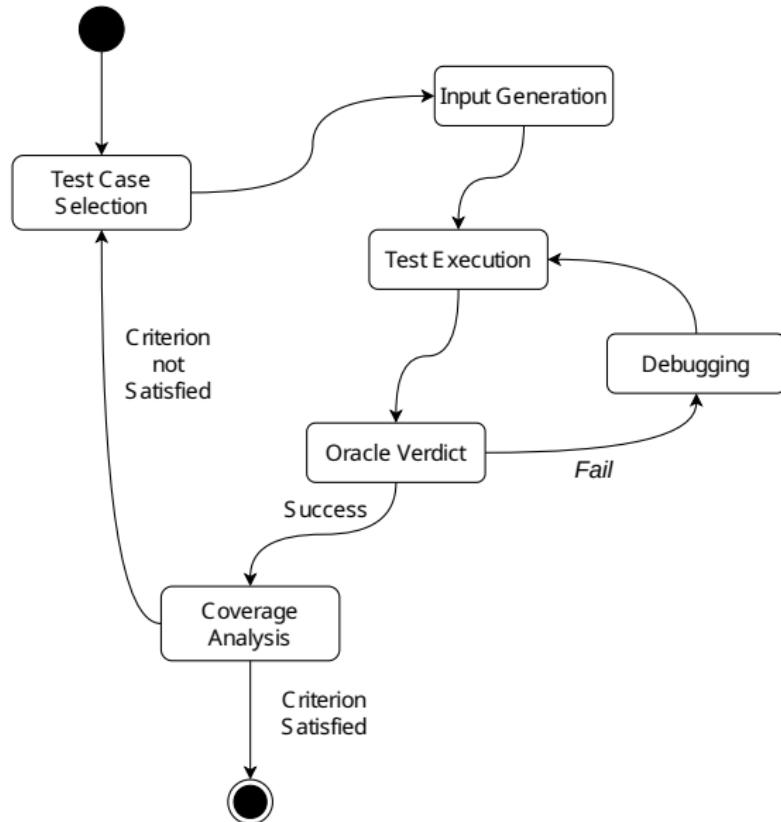


# The Test Pyramid

- Unit Tests should test **all the important code** i.e., *POJO* classes and *getters/setters* can be ignored
- Integration Tests should test the **core collaborations** between components
- e2e Tests should only test **some important scenarios**

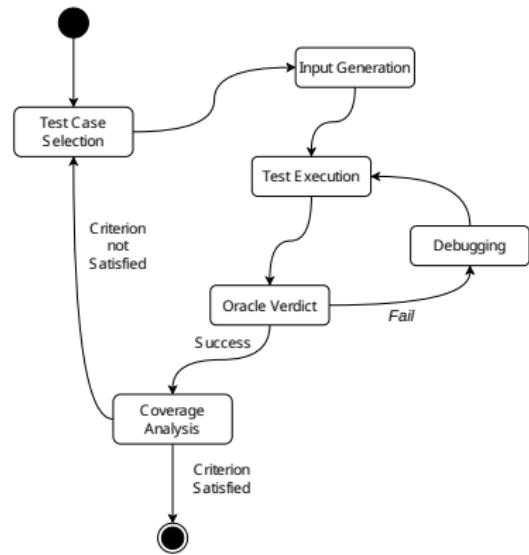


# Testing Process Activities



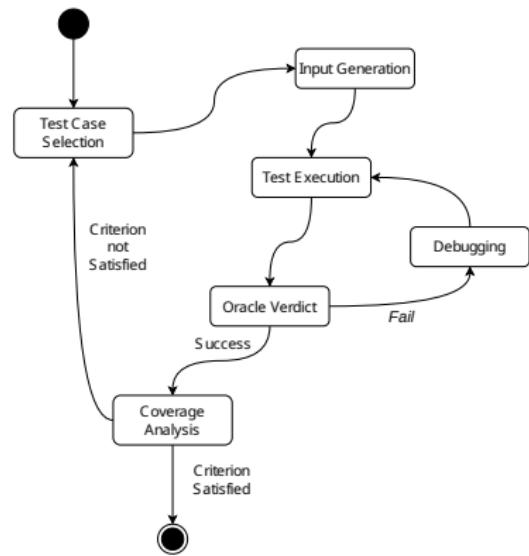
# Testing Process Activities 1/2

- **Test case selection:** define a suite of tests able to detect possible Faults  
*(more on this later)*
- **Input Generation:** identify inputs that let the system run along a test case. It is a kind of undecidable problem that could also address unfeasibility issues
- **Test Execution:** drive the execution of the test suite



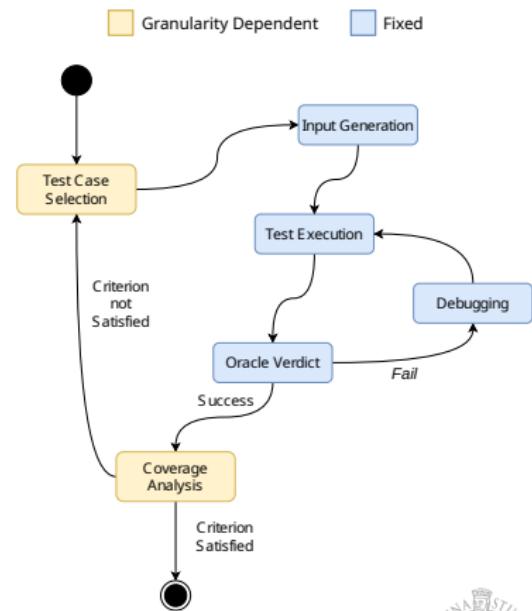
# Testing Process Activities 2/2

- **Oracle Verdict:** decides whether the SUT passes a test, i.e., test results comply with functional requirements. It is the most hard part to automate
- **Debugging:** trace back observed functional failures to structural faults
- **Coverage Analysis:** evaluate how test results cover the space of SUT behaviors (*more on this later*)



# How Testing Activities infer Test Cases Granularity

- The process to obtain a test case of any perspective **remains unchanged**
- The combination of the **test case selection** and the **coverage analysis** phases determines the level of granularity of the test case



# Test Case Selection and Coverage Analysis

- **Test Case Selection:**

- Identify a suite of tests able to **detect possible faults**
- Relies on an **abstraction** of the SUT

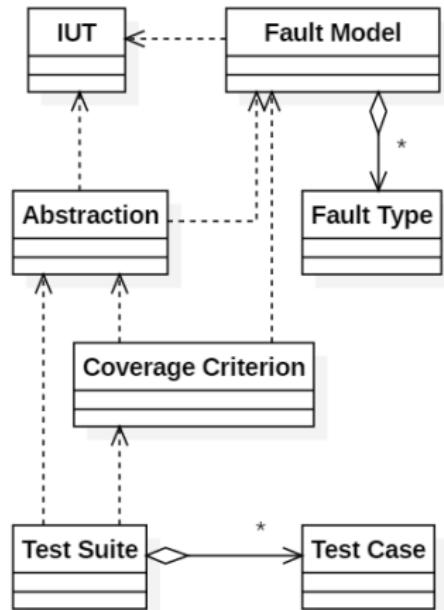
- **Coverage Analysis:**

- **evaluate** how test results cover the space of SUT behaviors
- Relies on a coverage criterion measured on an **abstraction** of the SUT
- Provides a **degree of confidence** on the absence of defects in the SUT
- Both procedures are **based on abstraction** properly: test case selection operates *a-priori* while coverage analysis operates *a-posteriori*



# Abstraction and Coverage Criterion

- Abstraction and coverage criterion depend on a **Fault Model**
- A Fault Model is made up of **types of faults** in the SUT that are being fought
- Fault model depends on SUT **structural and functional characteristics**



# Abstraction and Perspectives

- Abstractions can take different perspectives:
  - **Functional** (black box) refers to the SUT specification (e.g. use case diagrams, conceptual class diag., ...)
  - **Structural** (white box) refers to the SUT implementation (e.g. class diag, source or binary code, ...)
  - **Architectural** (grey box) refers to SUT architecture (e.g. architectural design, ...)
- Test case selection and coverage analysis can **operate on different abstractions** mixing perspectives:
  - e.g., select test cases from use cases, and then evaluate coverage on the code



# Relation between Test Pyramid and Abstraction

- Abstraction of different perspectives infer a **specific granularity**:
  - Structural abstractions will generate **fine-grained tests** (e.g., unit tests)
  - Functional and architectural abstractions will generate more **integrated tests** (e.g., integration tests or e2e tests)
  - Mixing perspectives (e.g., functional test case selection with architectural coverage analysis) will generate **gray box tests**
- Usually the **more coarse** the abstraction is, the **fewer tests** will be required to satisfy the coverage criterion
- To test the entire application the testing process should be **repeated at different level**
- Note that this implicitly recreates the **ratios** suggested in the test pyramid



# Testing Methodology for Unit Testing

- Unit testing analyses each component in isolation:
  - **Abstraction:** the source code of the component
  - **Coverage Criterion:** code coverage
- Tests will be selected in order to achieve a **coverage target**:
  - Usually 100% of the whole source code are required  
(also 80% is a popular target)
  - *POJO* classes and *getters/setters* are excluded from coverage



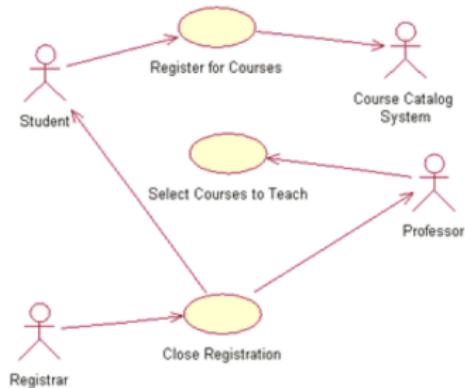
# More on Code Coverage

- Measures how many **lines of code** are executed in a test suite
- Gives an **idea** of how many parts of the code are tested
- Usually complemented by mutation testing to further strengthen test case selection
- An high percentage of code coverage **lowers the probability** of containing an undetected sw defects in the code
- The code coverage standard tool in Java is **JaCoCo** (Java Code Coverage)
- Well integrated in most popular IDEs (e.g., *Eclipse*, *IntelliJ*, *NetBeans*)



# Use Case Diagrams

- Visually represent **use cases**
- Main characters:
  - actors
  - use cases
- Different **level of abstraction**  
(we focuses on user goal level)
- Provide the “big picture” of functionalities



# Use Case Templates

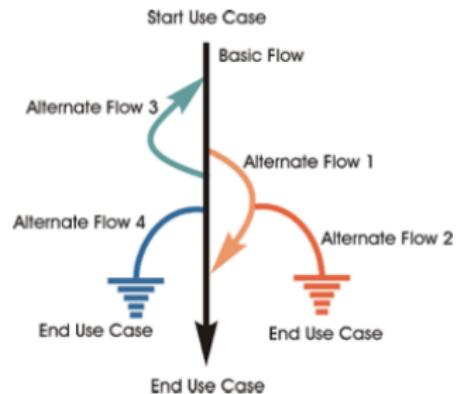
- Each use case also requires an adequate **textual documentation**
- Usually formatted in the so called: **use case templates**
- The core part is the **flow description**:
  - Normal flow
  - Alternative flows

| Use Case Section     | Description                                                                                                                                                                                                                   |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Name                 | An appropriate name for the use case (see <a href="#">Leslie Probasco's article</a> in the March issue of <i>The Rational Edge</i> ).                                                                                         |
| Brief Description    | A brief description of the use case's role and purpose.                                                                                                                                                                       |
| Flow of Events       | A textual description of what the system does with regard to the use case (not how specific problems are solved by the system). The description should be understandable to the customer.                                     |
| Special Requirements | A textual description that collects all requirements, such as non-functional requirements, on the use case, that are not considered in the use-case model, but that need to be taken care of during design or implementation. |
| Preconditions        | A textual description that defines any constraints on the system at the time the use case may start.                                                                                                                          |
| Post conditions      | A textual description that defines any constraints on the system at the time the use case will terminate.                                                                                                                     |



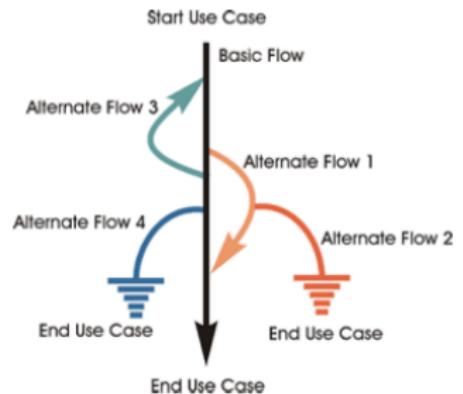
# Use Case Flows

- Basic and Alternative flows can be represented as **edges of a graph**
- Alternative flows can **return to the basic flow** of events
- ... Can also **terminate** the use case



# Use Case Abstraction

- The obtained graph can be used as **functional abstraction**
- On top of that, different coverage criteria could be required
- a test case is represented by a **use case scenario**



# References

- [Martin Fowler's Website](#)
- Slides on Testing by Prof. E. Vicario
- Slides on JUnit by F. Patara, PhD



## Software Engineering - A.A. 20/21

### Activity diagrams in process analysis and specification



Enrico Vicario

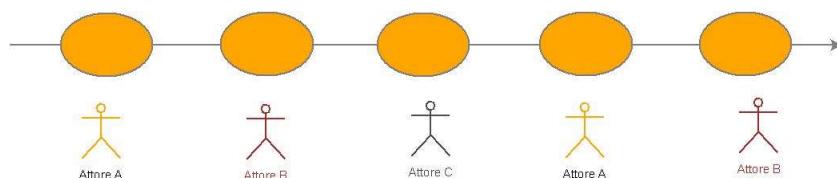
Dipartimento di Ingegneria dell'Informazione  
Laboratorio Scienza e Tecnologia del Software  
Università di Firenze

[enrico.vicario@unifi.it](mailto:enrico.vicario@unifi.it), [stlab.dinfo.unifi.it/vicario](http://stlab.dinfo.unifi.it/vicario)

1/42

### Procedura

- una sequenza di attività coordinate
  - svolte in tempi diversi,  
per effetto della intrinseca durata del processo  
(e.g. una sperimentazione clinica si svolge nell'arco di mesi e anni)
  - da uno più attori (ruoli),  
per separazione delle competenze o comunque delle responsabilità  
(e.g. il promotore, la segreteria del Comitato Etico, ...)



- è un problema ricorrente
  - pervasivo in servizi, processi amministrativi, gestionali, produttivi, ...
  - tipico nella pubblica amministrazione, ma non solo

2/42

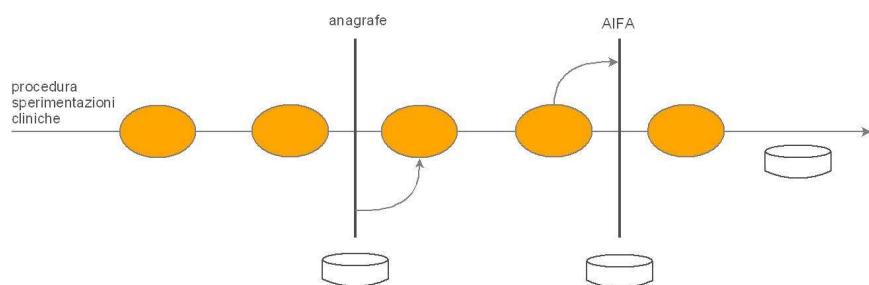
## Criticità nella gestione di procedure 1/2

- la procedura ha uno stato
  - non si esaurisce in una transazione, e si svolge in un arco di tempo
  - ciascuna attività si svolge nel contesto creato dalle precedenti
  - le diverse attività operano su dati e documenti condivisi e incrementati
- le responsabilità sono frammentate
  - le attività sono svolte da più soggetti (ruoli, uffici), ciascuno portatore di una propria prospettiva
  - mancano visione globale e controllo centralizzato sul flusso degli eventi
- numero di procedure e istanze
  - ciascuna procedura è concorrentemente eseguita su più casi specifici
  - una amministrazione esegue tipicamente più procedure
  - la forma di ciascuna evolve con la riorganizzazione dei processi

3/42

## Criticità nella gestione di procedure 2/2

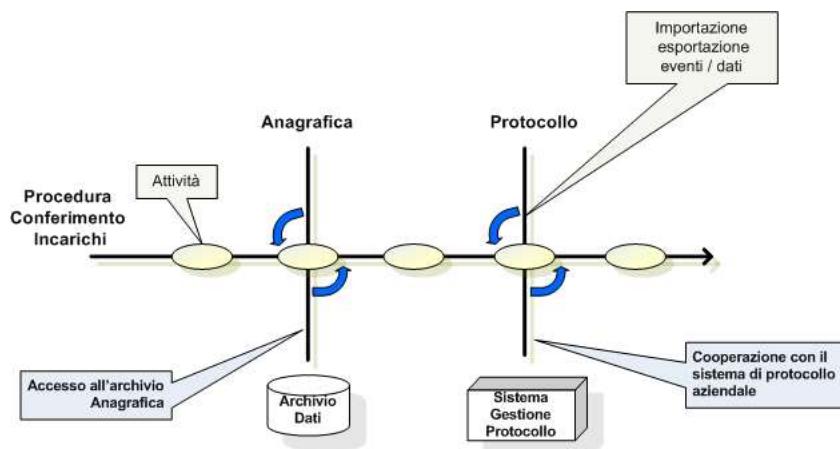
- La procedura interseca punti di cooperazione
  - sistemi informativi infrastrutturali
    - e.g. il protocollo, un'anagrafe, un identity provider, ...
  - sistemi informativi applicativi
    - e.g. una applicazione di gestione dei dati di una sperimentazione...
  - debito informativo verso sistemi esterni
    - e.g. Regione Toscana, AIFA, ...



4/42

### ... procedura orizzontale e applicativi verticali

- cooperazione con sistemi verticali che gestiscono in maniera autonoma informazione rilevante rispetto a singoli passi della procedura



5/42

### 3 livelli di capacità nella gestione

- processo esplicitato
  - esiste una rappresentazione documentata e condivisa della procedura
  - sostiene la (ri)organizzazione amministrativa, la formazione e la allocazione delle risorse umane
- processo attuato (enacted)
  - esiste un sistema informativo che coordina l'attuazione della procedura
  - fornisce efficienza e garantisce un livello di qualità ripetibile
  - favorisce la trasmissione tra centro e periferia
- processo ottimizzato
  - rispetto a requisiti di qualità (efficacia) e all'uso delle risorse (efficienza)
  - sottende capacità di valutazione ex-ante/ex-post e di evoluzione

6/42

## Processo esplicitato

- Esiste una rappresentazione documentata e condivisa della organizzazione della procedura
  - esiste un modello documentato e condiviso della procedura
  - sostiene la (ri)organizzazione amministrativa, la formazione e la allocazione delle risorse umane
- La rappresentazione può essere espressa attraverso schemi visuali standard
  - UML - Unified Modeling Language
  - Artefatti tecnici ma comprensibili
  - use case diagram: attori e attività
  - activity diagram: attività , sequenza e allocazione delle responsabilità
  - class diagram (modello concettuale): entità e relazioni nel dominio

7/42

## Activity Diagrams

- Derivato da formalismi SDL e Reti di Petri
- Adatto a descrivere procedure, workflow and business modeling
  - In qualche modo collegabile a data flow diagrams dell'analisi strutturata
- Adatto alla analisi di uno use case
  - Favorisce la identificazione delle operazioni, non degli oggetti a cui allocarle
  - Viceversa, può elaborare sulla base dei casi d'uso

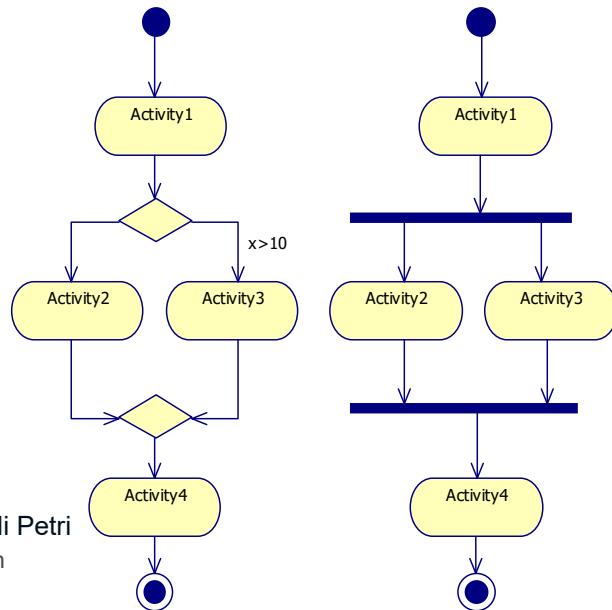
8/42

## Flusso di controllo e concorrenza

- Activity
- Sequence
- Start/End
- Branch
- Fork/join

### Modello delle Reti di Petri

- concetto del token

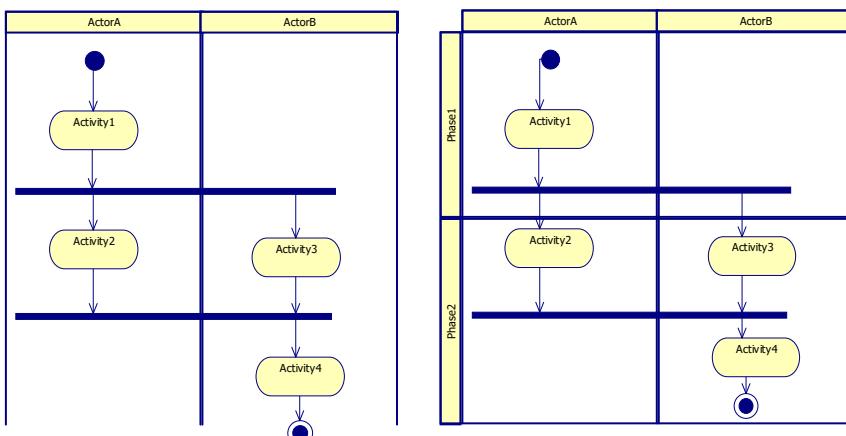


9/42

## Allocazione ad attori e fasi

### Swimlanes

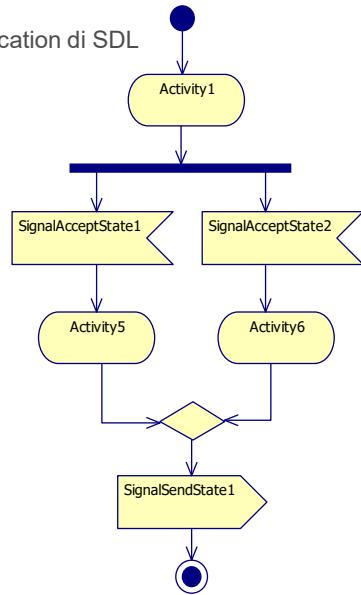
- In più direzioni (partizioni)



10/42

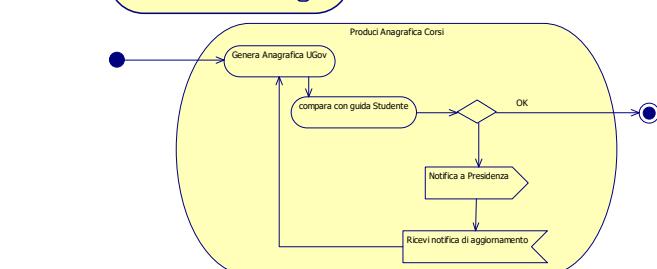
■ Signals

- sottende il modello della Process Specification di SDL
- comunicazione tra processi sequenziali



11/42

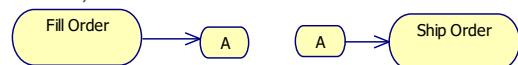
■ Sub-Aattività



■ Connettori



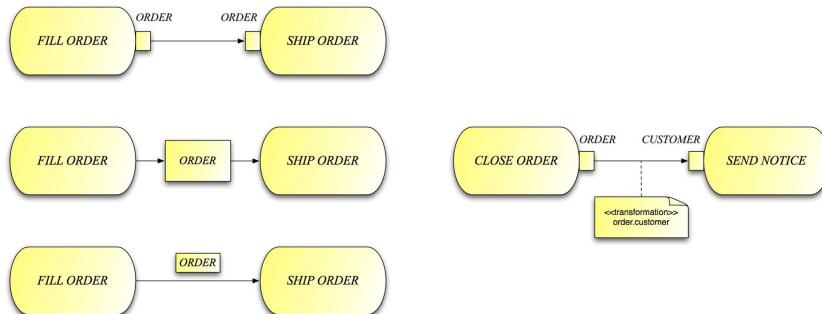
- ... in starUML (vers. 5.0.2 del 2005)



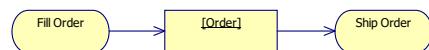
12/42

■ Pin e oggetti

- Oggetti o token
- Bilanciamento tra porte di ingresso e uscita
- Eventuali trasformazioni possibili



- In StarUML (versione 5.0.2 del 2005)



- Reti di Petri colorate

13/42

■ Come si arriva all'activity diagram ?

- Class diagram + attori + use cases

■ Osservazioni

- Le attività corrispondono ai casi d'uso?
- Stiamo descrivendo o specificando un processo?
- Il processo materiale e la sua impronta nel sistema informativo
- Schede CRC nell'identificazione degli attori

14/42

## Esempio: conferimento di un contratto

- Procedura conferimento di un contratto avviata da un docente
  - Fase del bando
  - Selezione e arruolamento
  - Chiusura e rinnovo
- Passi della analisi
  - Specifica preliminare
  - Identificazione di classi, attori e casi d'uso
  - Organizzazione in un activity diagram

15/42

## Specifiche preliminare: fase del bando

- Un docente presenta richiesta di attivazione al Consiglio di Dipartimento, indicando il titolo, il tema della ricerca, i prerequisiti (e.g. tipo di laurea, votazione, esperienza pregressa in specifici ambiti, ...), il periodo di svolgimento, la motivazione della necessità di un contratto, i fondi su cui intende finanziare il contratto.
- La richiesta viene immessa nell'ordine del giorno della prossima riunione del consiglio di dipartimento a cui afferisce il docente.
- Il Consiglio nomina la commissione formata dal docente stesso (nel ruolo di responsabile scientifico) e altri due docenti afferenti a questo o ad altro dipartimento. Nella nomina possono essere indicati fino a due membri supplenti.
- I docenti incaricati ricevono notifica, e accettano l'incarico (eccezione: se non accettano la richiesta viene inviata ai membri supplenti)
- Il docente responsabile dell'assegno comunica ai membri della commissione la data della prova. La data deve essere almeno 40 giorni avanti.
- Il bando per l'assegno viene comunicato all'ufficio Ricerca dell'Ateneo, che pubblica il titolo, la descrizione della ricerca, i requisiti di partecipazione, i termini per la presentazione della domanda.
- L'Ufficio Ricerca anche invia per via telematica ai membri della commissione la convocazione per la prova di selezione.

16/42

## Specifiche preliminari: selezione e arruolamento

- Ciascun candidato, presenta domanda per via telematica inserendo i propri dati anagrafici, descrivendo titoli e allegati, allegando files. Il sistema produce una versione elettronica della domanda (che include l'elenco degli allegati ma non gli allegati stessi). Il candidato stampa la domanda e la presenta in forma cartacea firmata e con marca da bollo all'ufficio Ricerca.
- Entro la data della prova di selezione, l'ufficio verifica quali domande possono essere ammesse in base a vari criteri: effettiva ricezione della domanda in forma cartacea, adeguatezza dei titoli presentati, ...
- L'Ufficio comunica al Responsabile scientifico l'elenco dei candidati ammessi e invia ai membri della commissione un promemoria per la prova di selezione assieme a un fac-simile del verbale.
- In occasione della selezione viene redatto un verbale. Il sistema pre-compila tutte le parti già note (incluso descrizione del titolo, membri della commissione, nomi dei partecipanti, ...) . I commissari compilano on-line la valutazione dei titoli e del colloquio per ciascun candidato che si è presentato alla prova, decretano il vincitore della selezione.
- Il sistema comunica a tutti i partecipanti un estratto del verbale che include l'indicazione del vincitore. Il Sistema comunica al vincitore l'invito a presentarsi all'ufficio contratti per la firma.
- Il sistema notifica all'ufficio contratti i termini del contratto e il riferimento del responsabile scientifico e della unità amministrativa.
- Quando il vincitore firma il contratto, l'ufficio contratti notifica l'avvio del contratto di collaborazione all'ufficio stipendi.

17/42

## Specifiche preliminari: chiusura e rinnovo

- Quando mancano 60 giorni al termine del contratto, il sistema invia un avviso al responsabile della ricerca e al titolare del contratto.
- Il titolare del contratto invia una relazione delle attività svolte che viene controfirmata dal responsabile scientifico e accompagnata da una valutazione.
- La relazione e' inviata all'Ufficio ricerca in forma elettronica entro 30 giorni dalla data di scadenza del contratto.
- Il responsabile scientifico ha facoltà di richiedere il rinnovo del contratto allo stesso soggetto che ne attualmente titolare per un periodo pari a quello del primo incarico, senza ulteriore approvazione del consiglio di dipartimento e senza ulteriore bando e selezione. In tal caso il sistema invia al titolare del contratto una notifica a cui il titolare risponde indicando la sua eventuale accettazione; la accettazione e' inviata all'ufficio stipendi.
- Il titolare del contratto può recedere inviando una comunicazione all'ufficio Ricerca. Ne ricevono notifica il responsabile scientifico e l'ufficio stipendi.
- Viceversa anche il responsabile scientifico può chiedere l'interruzione del contratto dove risultino applicabili i criteri indicati nel contratto stesso. La richiesta è notificata all'ufficio affari legali, all'ufficio ricerca, all'ufficio stipendi, e al titolare del contratto. La procedura di rescissione non è qui descritta.

18/42

## Identificazione oggetti e attributi: fase del bando

- Un docente presenta richiesta di attivazione al Consiglio di Dipartimento, indicando il titolo, il tema della ricerca, i prerequisiti (e.g. tipo di laurea, votazione, esperienza pregressa in specifici ambiti, ...), il periodo di svolgimento, la motivazione della necessità di un contratto, i fondi su cui intende finanziare il contratto. La richiesta viene inserita nell'ordine del giorno della prossima riunione del consiglio di dipartimento a cui afferisce il docente.
- Il Consiglio nomina la commissione formata dal docente stesso (nel ruolo di responsabile scientifico) e altri due docenti afferenti a questo o ad altro dipartimento. Nella nomina possono essere indicati fino a due membri supplenti.
- I docenti incaricati ricevono notifica, e accettano l'incarico (eccezione: se non accettano la richiesta viene inviata ai membri supplenti)
- Il docente responsabile dell'assegno comunica ai membri della commissione la data della prova di selezione. La data deve essere almeno 40 giorni avanti.
- Il bando per l'assegno viene comunicato all'ufficio Ricerca dell'Ateneo, che pubblica il titolo, la descrizione della ricerca, i requisiti di partecipazione, i termini per la presentazione della domanda.
- L'Ufficio Ricerca anche invia per via telematica ai membri della commissione la convocazione per la prova di selezione.

19/42

## Identificazione oggetti e attributi: selezione e arruolamento

- Ciascun candidato, presenta domanda per via telematica inserendo i propri dati anagrafici, descrivendo titoli e allegati, allegando files. Il sistema produce una versione elettronica della domanda (che include l'elenco degli allegati ma non gli allegati stessi). Il candidato stampa la domanda e la presenta in forma cartacea firmata e con marca da bollo all'ufficio Ricerca.
- Entro la data della prova di selezione, l'ufficio verifica quali domande possono essere ammesse in base a vari criteri: effettiva ricezione della domanda in forma cartacea, adeguatezza dei titoli presentati, ...
- L'Ufficio comunica al Responsabile scientifico l'elenco dei candidati ammessi e invia ai membri della commissione un promemoria per la prova di selezione assieme a un fac-simile del verbale.
- In occasione della selezione viene redatto un verbale. Il sistema pre-compila tutte le parti già note (incluso descrizione del titolo, membri della commissione, nomi dei partecipanti, ...) . I commissari compilano on-line la valutazione dei titoli e del colloquio per ciascun candidato che si è presentato alla prova, decretano il vincitore della selezione.
- Il sistema comunica a tutti i partecipanti un estratto del verbale che include l'indicazione del vincitore. Il Sistema comunica al vincitore l'invito a presentarsi all'ufficio contratti per la firma.
- Il sistema notifica all'ufficio contratti i termini del contratto e il riferimento del responsabile scientifico e del dipartimento
- Quando il vincitore firma il contratto, l'ufficio contratti notifica l'avvio del contratto di collaborazione all'ufficio stipendi.

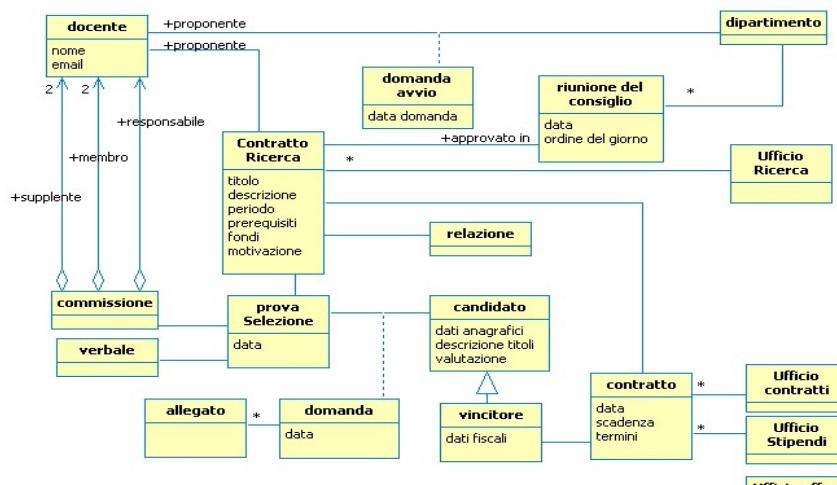
20/42

## Identificazione oggetti e attributi: chiusura e rinnovo

- Quando mancano 60 giorni al termine del contratto, il sistema invia un avviso al responsabile della ricerca e al titolare del contratto.
- Il titolare del contratto invia una **relazione delle attività** svolte che viene controfirmata dal responsabile scientifico e accompagnata da una valutazione.
- La relazione e' inviata all'Ufficio ricerca in forma elettronica entro 30 giorni dalla data di scadenza del contratto.
- Il responsabile scientifico ha facoltà di richiedere il rinnovo del contratto allo stesso soggetto che ne attualmente titolare per un periodo pari a quello del primo incarico, senza ulteriore approvazione del consiglio di dipartimento e senza ulteriore bando e selezione. In tal caso il sistema invia al titolare del contratto una notifica a cui il titolare risponde indicando la sua eventuale accettazione; la accettazione e' inviata all'ufficio stipendi.
- Il titolare del contratto può recedere inviando una comunicazione all'ufficio Ricerca. Ne ricevono notifica il responsabile scientifico e l'ufficio stipendi.
- Viceversa anche il responsabile scientifico può chiedere l'interruzione del contratto dove risultino applicabili i criteri indicati nel contratto stesso. La richiesta è notificata all'**ufficio affari legali**, all'ufficio ricerca, all'ufficio stipendi, e al titolare del contratto. La procedura di rescissione non è qui descritta.

21/42

## Class diagram concettuale



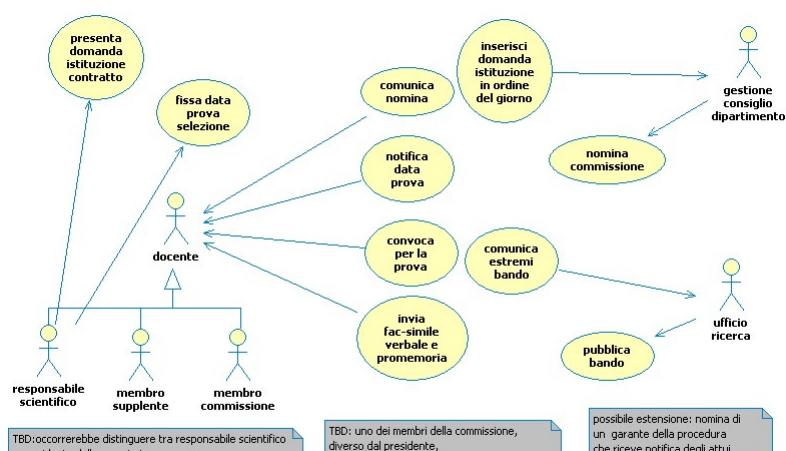
22/42

### Identificazione casi d'uso: fase del Bando

- Un docente presenta richiesta di attivazione al Consiglio di Dipartimento, indicando il titolo, il tema della ricerca, i prerequisiti (e.g. tipo di laurea, votazione, esperienza pregressa in specifici ambiti, ...), il periodo di svolgimento, la motivazione della necessità di un contratto, i fondi su cui intende finanziare il contratto. La richiesta viene inserita nell'ordine del giorno della prossima riunione del consiglio di dipartimento a cui afferisce il docente.
- Il Consiglio nomina la commissione formata dal docente stesso (nel ruolo di responsabile scientifico) e altri due docenti afferenti a questo o ad altro dipartimento. Nella nomina possono essere indicati fino a due membri supplenti.
- I docenti incaricati ricevono notifica, e accettano l'incarico (eccezione: se non accettano la richiesta viene inviata ai membri supplenti)
- Il docente responsabile dell'assegno comunica ai membri della commissione la data della prova di selezione. La data deve essere almeno 40 giorni avanti.
- Il bando per l'assegno viene comunicato all'ufficio Ricerca dell'Ateneo, che pubblica il titolo, la descrizione della ricerca, i requisiti di partecipazione, i termini per la presentazione della domanda.
- L'Ufficio Ricerca anche invia per via telematica ai membri della commissione la convocazione per la prova di selezione.

23/42

### Use case diagram: fase del Bando



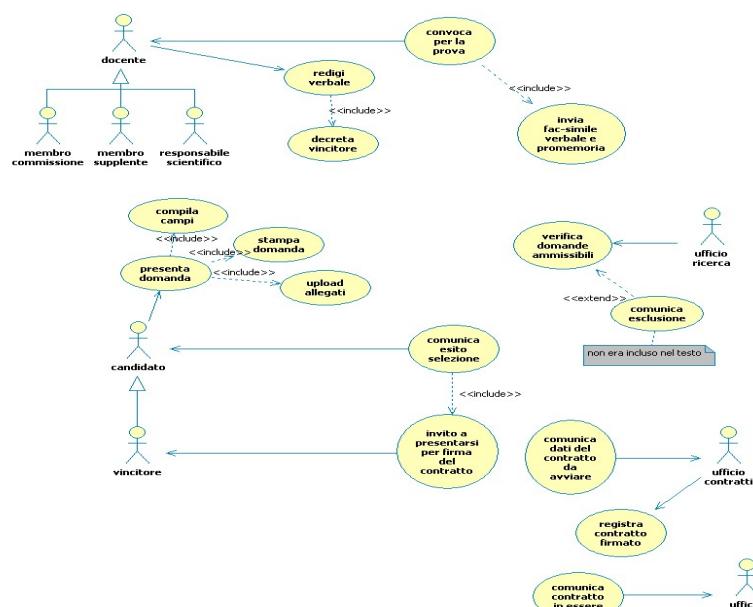
24/42

## Identificazione casi d'uso: selezione e arruolamento

- Ciascun **candidato**, **presenta domanda** per via telematica inserendo i propri **dati anagrafici**, descrivendo **titoli** e **allegati**, allegando files. Il sistema produce una versione elettronica della domanda (che include l'elenco degli allegati ma non gli allegati stessi). Il candidato **stampa** la domanda e la presenta in forma cartacea firmata e con marca da bollo all'**ufficio Ricerca**.
- Entro la data della prova di selezione, l'**ufficio verifica** quali domande possono essere ammesse in base a vari criteri: effettiva ricezione della domanda in forma cartacea, adeguatezza dei titoli presentati, ...
- L'**Ufficio comunica** al Responsabile scientifico l'elenco dei candidati ammessi e invia ai membri della commissione un promemoria per la prova di selezione assieme a un fac-simile del **verbale**.
- In occasione della selezione **viene redatto** un verbale. Il sistema pre-compila tutte le parti già note (incluso descrizione del titolo, membri della commissione, nomi dei partecipanti, ...) . I commissari compilano on-line la valutazione dei titoli e del colloquio per ciascun candidato che si è presentato alla prova, decretano il **vincitore della selezione**.
- Il sistema **comunica** a tutti i partecipanti un estratto del verbale che include l'indicazione del vincitore. Il Sistema **comunica** al vincitore l'invito a presentarsi all'**ufficio contratti** per la firma.
- Il sistema **notifica** all'**ufficio contratti** i termini del contratto e il riferimento del responsabile scientifico e del dipartimento
- Quando il vincitore firma il contratto, l'**ufficio contratti** **notifica** l'avvio del contratto di collaborazione **all'**ufficio stipendi****.

25/42

## Use case diagram: selezione e arruolamento



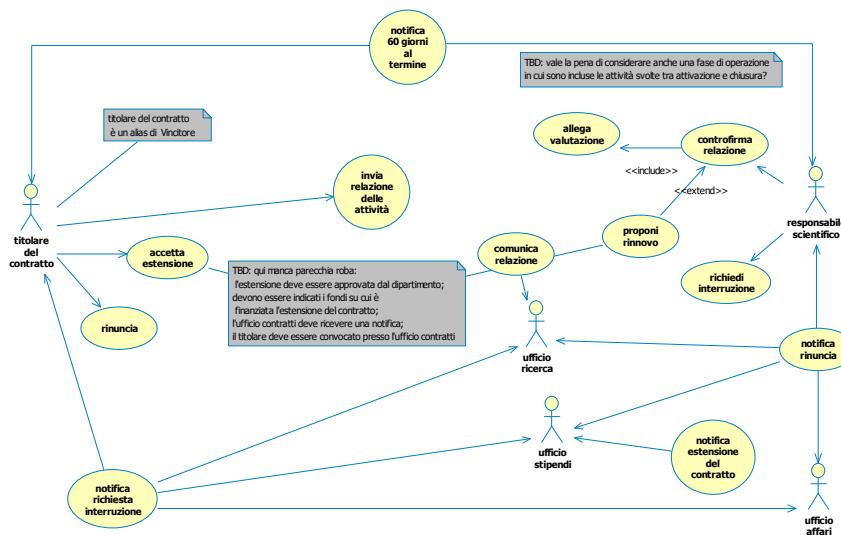
26/42

## Identificazione casi d'uso: chiusura e rinnovo

- Quando mancano 60 giorni al termine del contratto, il sistema **invia un avviso** al responsabile della ricerca e al titolare del contratto.
- Il titolare del contratto **invia** una **relazione delle attività** svolte che viene **controfirmata** dal responsabile scientifico e **accompagnata** da una valutazione.
- La relazione e' **inviata** all'Ufficio ricerca in forma elettronica entro 30 giorni dalla data di scadenza del contratto.
- Il responsabile scientifico ha facoltà di **richiedere il rinnovo** del contratto allo stesso soggetto che ne attualmente titolare per un periodo pari a quello del primo incarico, senza ulteriore approvazione del consiglio di dipartimento e senza ulteriore bando e selezione. In tal caso il sistema invia al titolare del contratto una notifica a cui il titolare risponde **indicando** la sua eventuale accettazione; la accettazione e' **inviata** all'ufficio stipendi.
- Il titolare del contratto può recedere **invitando** una comunicazione all'ufficio Ricerca. Ne ricevono notifica il responsabile scientifico e l'ufficio stipendi.
- Viceversa anche il responsabile scientifico può **chiedere** l'interruzione del contratto dove risultino applicabili i criteri indicati nel contratto stesso. La richiesta è **notificata all'ufficio affari legali**, all'ufficio ricerca, all'ufficio stipendi, e al titolare del contratto. La procedura di rescissione non è qui descritta.

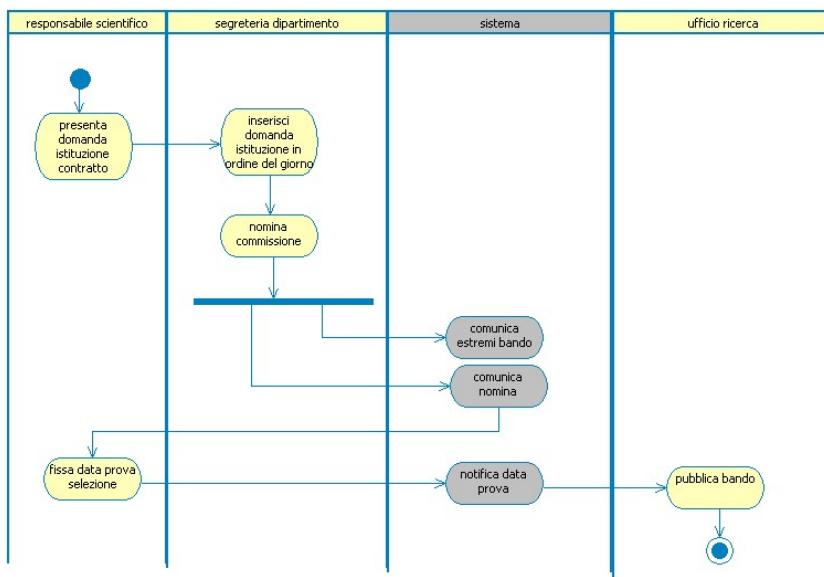
27/42

## Use case diagram: chiusura e rinnovo



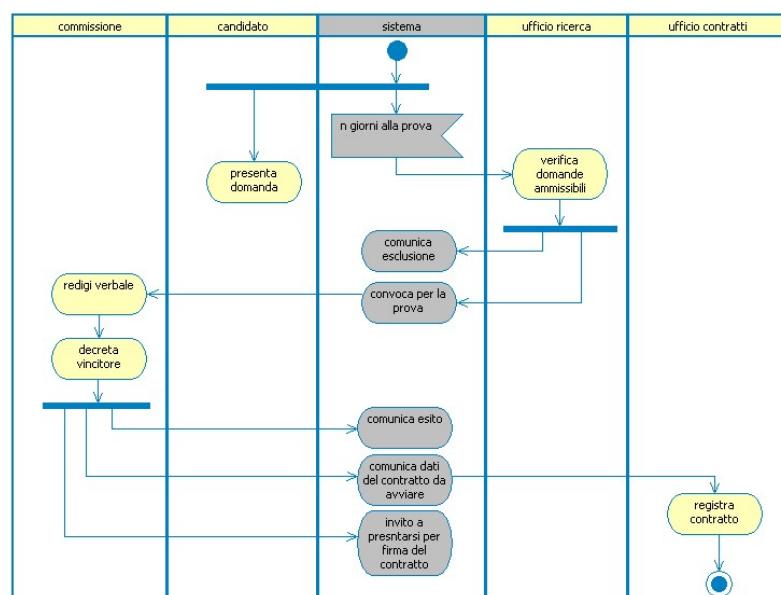
28/42

### Activity diagram: fase del bando



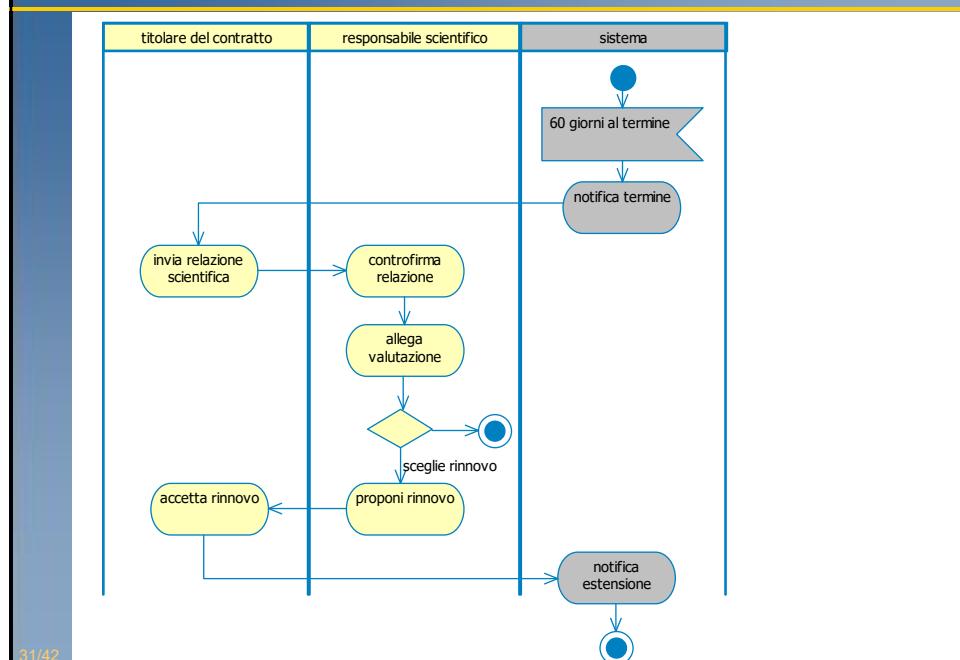
29/42

### Activity diagram: selezione e arruolamento



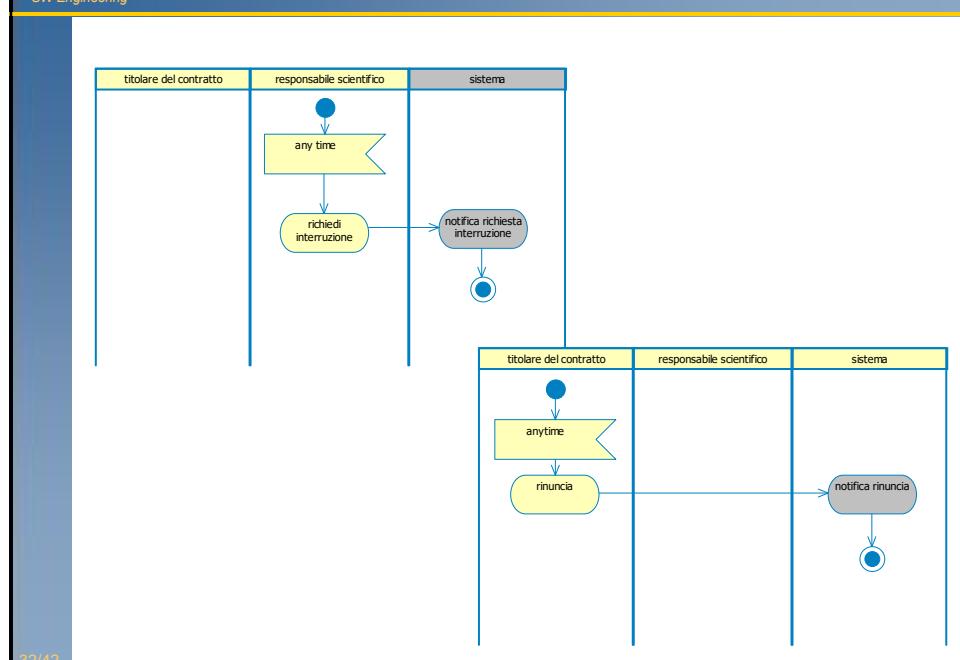
30/42

### Activity diaagram: chiusura e rinnovo 1/2



31/42

### Activity diagram: chiusura e rinnovo 2/2



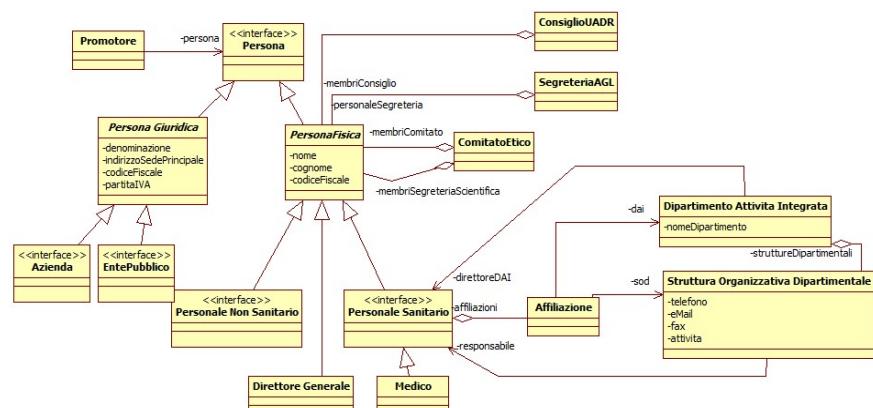
32/42

- Alistair Cockburn, "Writing effective use cases," Addison Wesley, Pearson Education, 2001.
- Martin Fowler, "UML Distilled: Guida rapida al linguaggio di modellazione standard" - terza edizione, Pearson Education Italia, Febbraio 2004 .
- Martin Fowler, "UML Distilled: a brief guide to the standard object modeling language", third edition (Addison Wesley).
- Martin Fowler, "Analysis Patterns, reusable object models", Addison Wesley 1997.
- J.Arlow, I.Neustadt, "UML e unified process," McGraw Hill, 2003.

- Processo di autorizzazione delle sperimentazioni cliniche
  - risultato di un'analisi presso il Comitato Etico di Careggi
  - tradotto poi in applicazione
- Elementi della descrizione
  - modello di dominio
  - attori
  - casi d'uso
  - activity diagram

### modello concettuale del dominio (1/3)

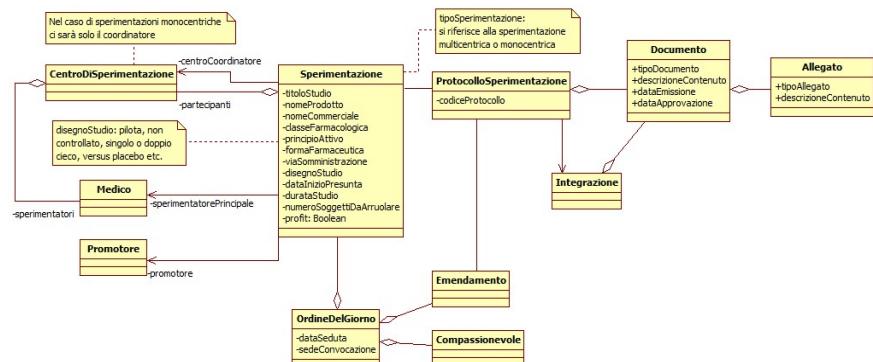
- Modello di dominio – parti coinvolte



35/42

### modello concettuale del dominio (2/3)

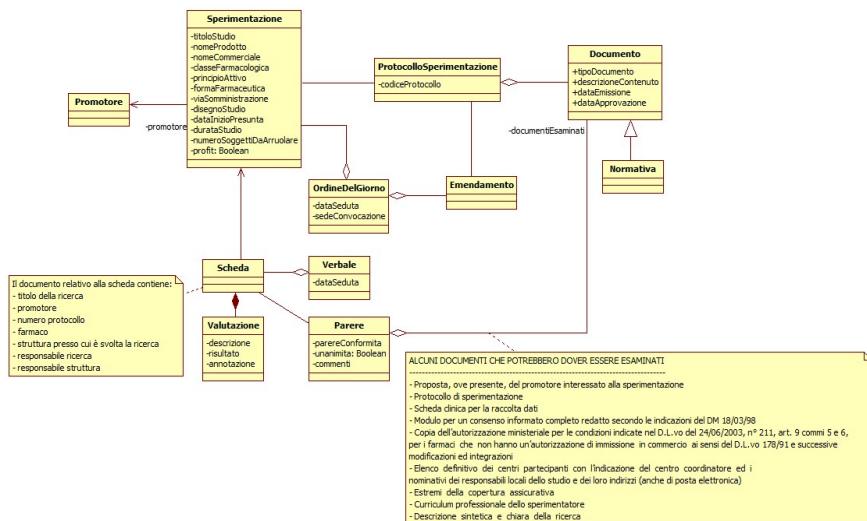
- Modello di dominio – Sperimentazione



36/42

### modello concettuale del domino (3/3)

#### Modello di dominio – Valutazione



37/42

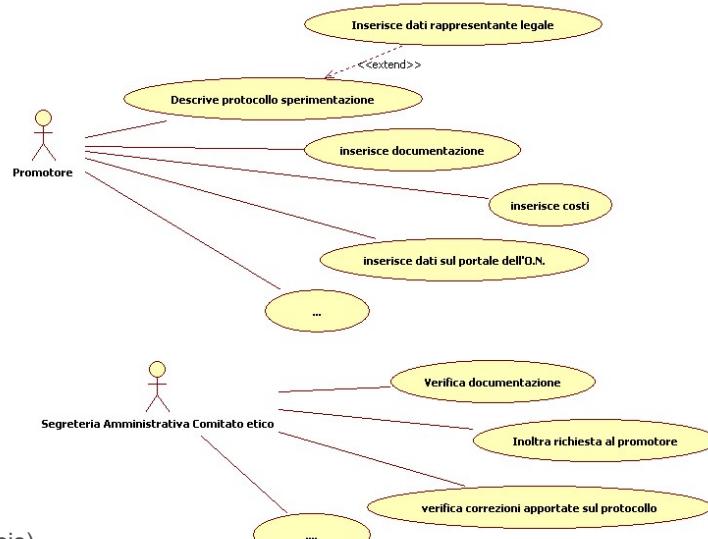
### Attori

#### Attori: ruoli coinvolti nella procedura



38/42

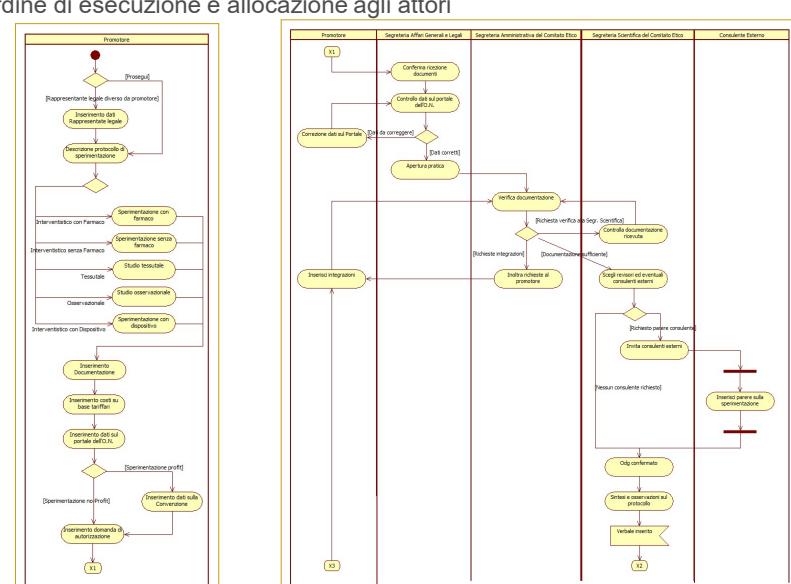
■ Casi d'uso



39/42

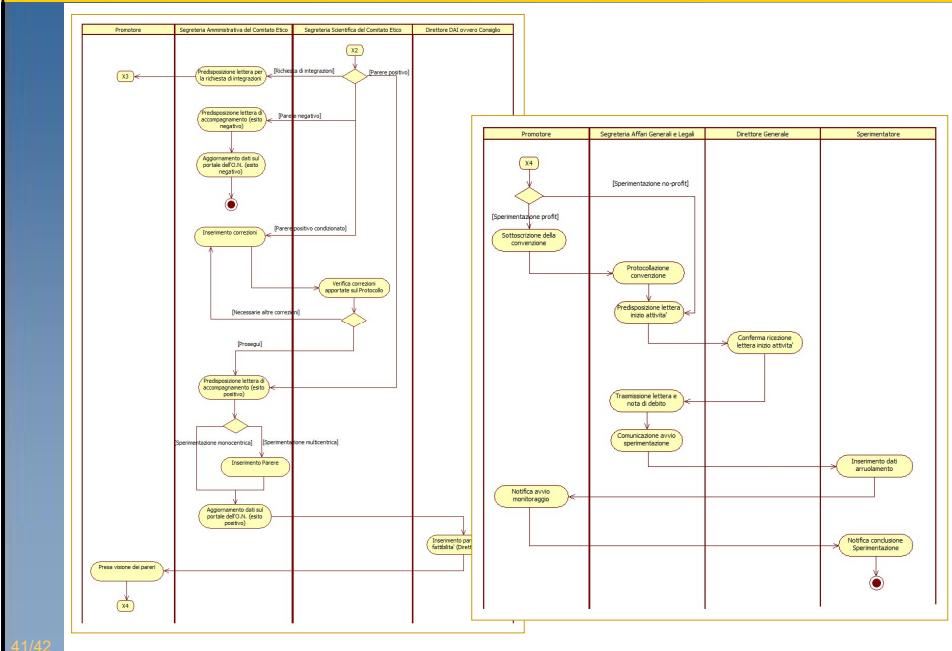
■ Modello delle attività: mappa le attività della procedura

- Ordine di esecuzione e allocazione agli attori



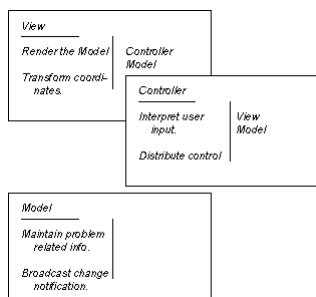
40/42

## mappa della procedura (2/2)



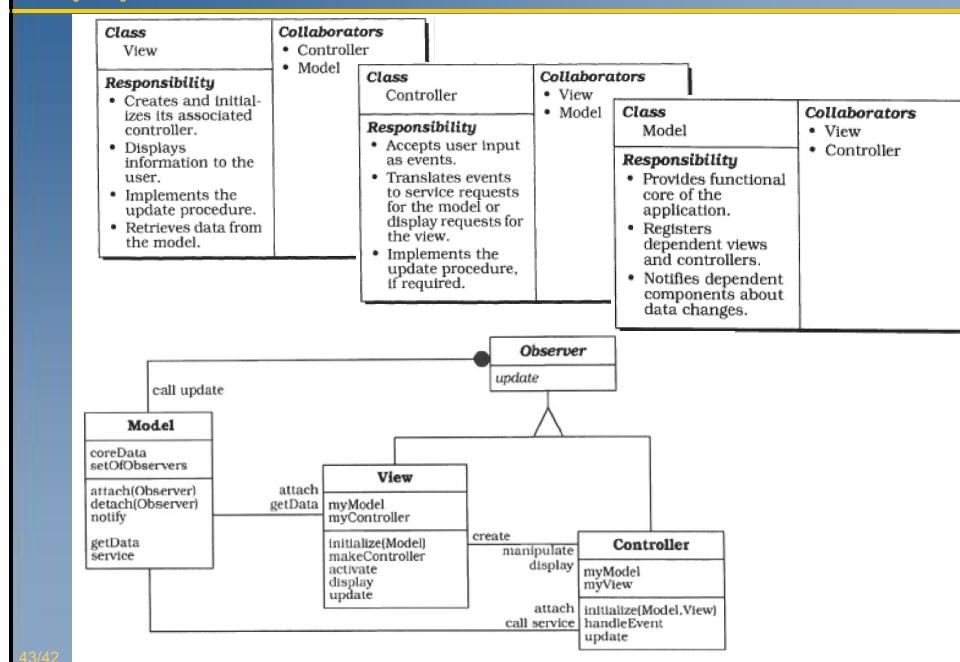
## Schede CRC

- Metodo delle schede CRC(Class-Responsibility-Collaboration)
  - Si identifica un insieme di classi
  - Per ciascuna si elencano le responsabilità salienti
  - Per ciascuna classe si elencano le collaborazioni necessarie ad assolvere la responsabilità



- Usate nella identificazione e allocazione di responsabilità
  - Un modo semplice e quindi preliminare
  - Nel disegno architettonurale, sul livello enterprise o degli oggetti
  - Applicabile anche nella analisi e la progettazione di un processo

### Schede CRC: Model View Controller [POSA]



43/42

### Schede CRC: Publish & Subscribe [MS]

| Components                   | Responsibilities                                                                                                                                                                                                                                                                | Collaborations                                                                                                                                          |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| Communication infrastructure | <ul style="list-style-type: none"> <li>Maintains the subscribers' subscriptions.</li> <li>Inspects the topic-related information or the content information that is included in each published message.</li> <li>Tранспортирует сообщение к подписанным приложениям.</li> </ul> | <ul style="list-style-type: none"> <li>The publisher publishes messages.</li> <li>The subscriber subscribes to topics and receives messages.</li> </ul> |
| Publisher                    | <ul style="list-style-type: none"> <li>Inserts topic-related information or content information in each message.</li> <li>Publishes the message to the communication infrastructure.</li> </ul>                                                                                 | The communication infrastructure transports messages to subscribers.                                                                                    |
| Subscriber                   | <ul style="list-style-type: none"> <li>Subscribes to one or more topics or message content types.</li> <li>Consumes messages published to the subscribed topics.</li> </ul>                                                                                                     | The communication infrastructure transports published messages from the publisher.                                                                      |

44/42

Software Engineering - A.A. 20/21

## Analysis Idioms in the creation of a Domain Model

Reflections from experience in OO Analysis and Design  
mainly oriented to an implementation/specification perspective



Enrico Vicario  
Dipartimento di Ingegneria dell'Informazione  
Laboratorio Tecnologie del Software  
Università di Firenze

enrico.vicario@unifi.it, stlab.dinfo.unifi.it/vicario

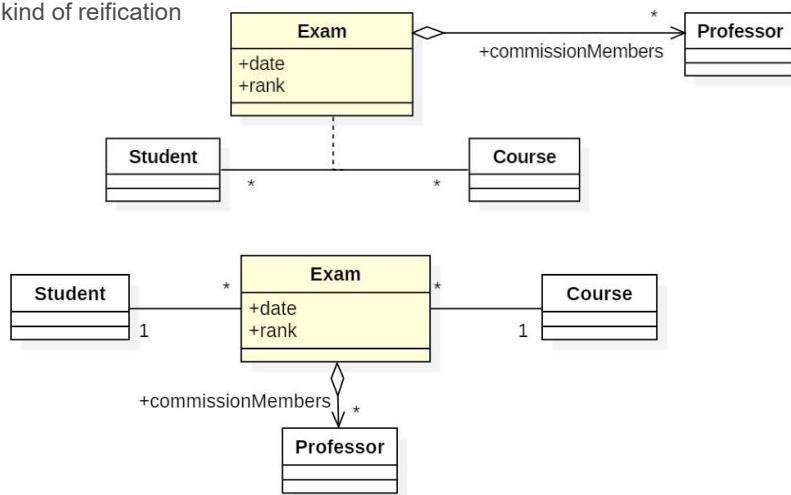
1/29

## Remarks on Analysis idioms – 1/10



### Association classes

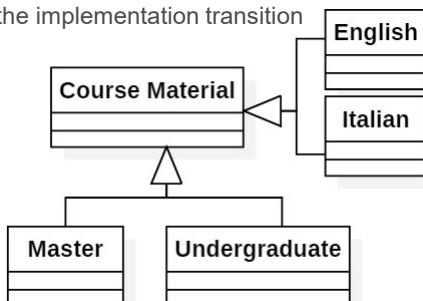
- a kind of shortcut in the analysis, later translated in the implementation
- e.g. Client, Product, Order
- e.g. Student, Course, Exam
- a kind of reification



2/97

■ Orthogonal classification

- again, a shortcut in conceptual analysis
- more complex to be translated in the implementation transition
- (Bridge Pattern)



■ Comments and constraints

- avoid complexity and stiffness
- often preparing deployment of some design pattern

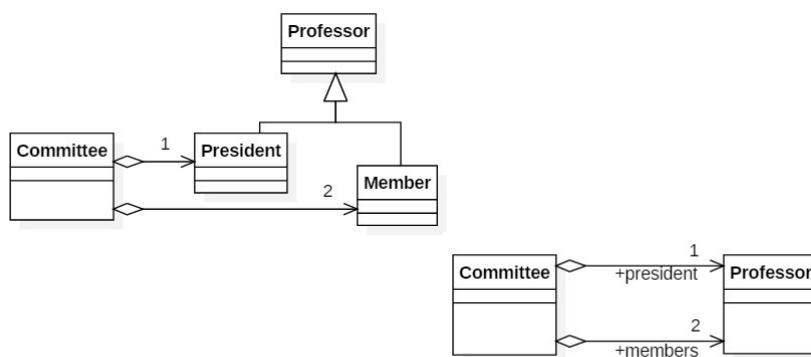
■ it is often said that: design patterns emerge  
in the transition from specification to implementation

- whereas, often Design Patterns disappear in that transition
- e.g. Composite Pattern

3/97

■ Roles are lighter than Types

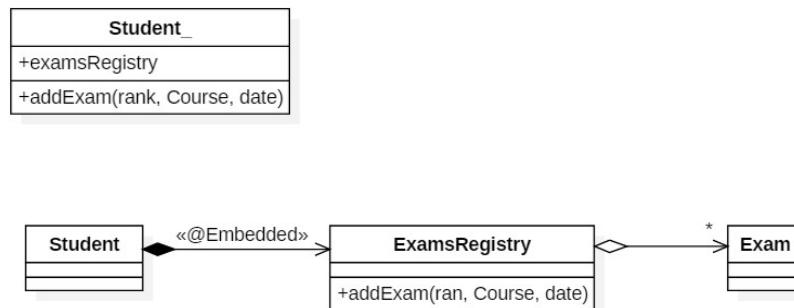
- e.g. a Committee has a President and 2 Members,  
and all of them are Professors
- ... unless the President has specific Responsibilities to be captured  
in the Digital representation (a PEC address, a nomination decree, ...)



4/97

■ Attributes and Types

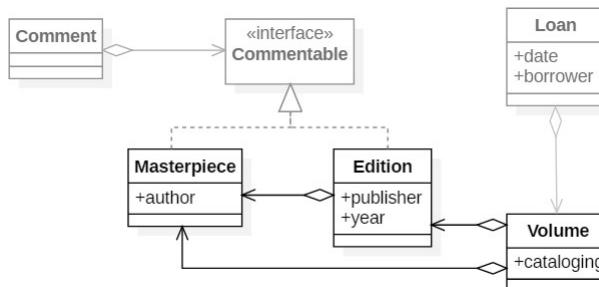
- an object shall have State, Behaviour, and Identity ... what if Identity is resolved on the database?
- e.g. Student and ExamRegistry
- ... embeddable, UML composition, offloaded to simplify, but without adding identity, affected by the architecture



5/97

■ the same term may subtend different abstractions

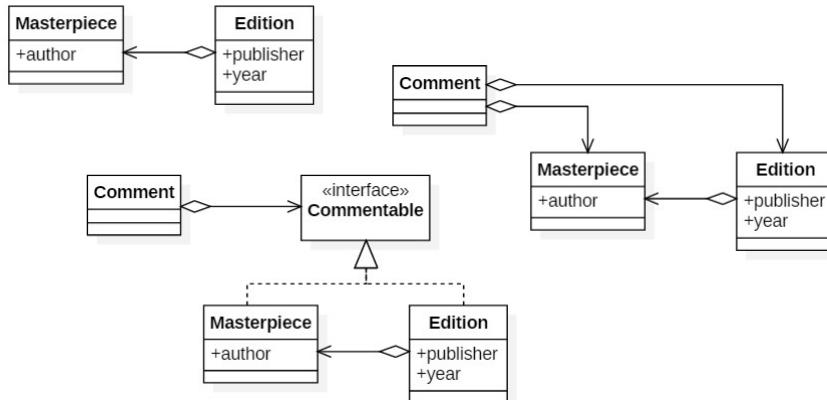
- each responsible for different attributes and operations, and references to associated types
- e.g. a MasterPiece, an Edition, a Volume (title, author, year, year, size, location on the scaffold, )
- e.g. a Course and its concrete implementation (CFU, SSD, room, timetable...)
- e.g. a ProductionProcess and a ProductionProcessInstance (the recipe, nominal and actual parameters, position in the store, ...)



6/97

## Generalization (over Specialization) – 6/10

- create a common abstraction for types that are not distinguished in some use
  - proceeds bottom-up, as an abstraction refactoring not, top-down, as a specialization
  - e.g. MasterPiece and Edition are not initially sibling subtype, the interface Commentable is created later to accommodate Comment referred to a Masterpiece or an Edition

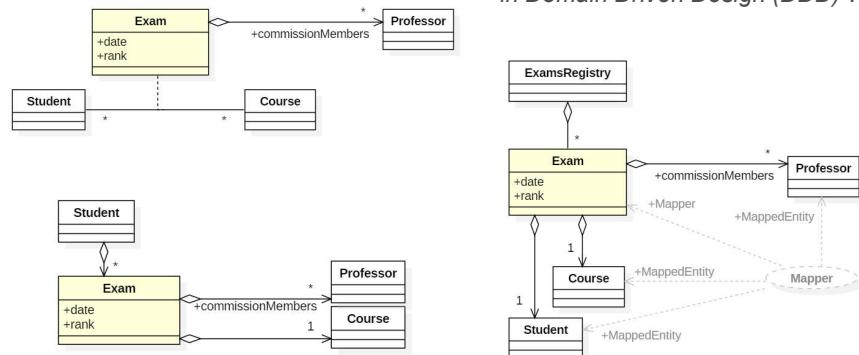


7/29

## Remarks on Analysis idioms - 7/10

- Mapper
  - a kind of reification of abstract actions, relations, ...
  - avoids charging a type with marginal responsibilities
  - permitting multiple areas of business be implemented around a common type
  - e.g. Student, Course, Exam, Commission;
  - may break navigation direction suggested by use cases (but, this may be not a problem; cmp remark on Navigation Direction)

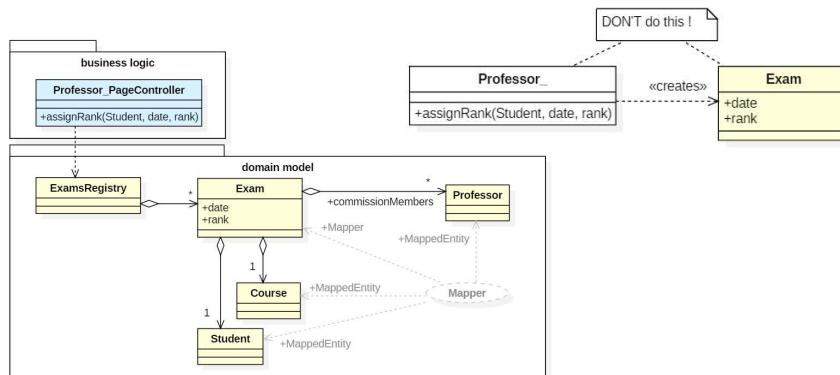
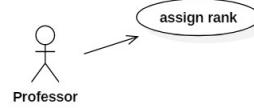
*TBD: should this be related to the Bounded Context Pattern in Domain Driven Design (DDD) ?*



8/97

■ the anti-pattern of target and agent

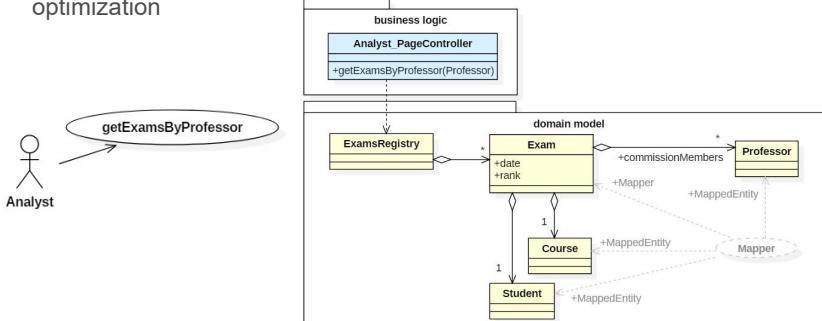
- operations shall be assigned to the class whose state is used (def/use)
- ... or they shall be captured in a use case rather than in the class diagram
- unless you need to log operations
- e.g. the Professor assigns a Rank to the Student
- e.g. the Physician administers a Drug to the Patient
- much related to the architecture, and the difference between domain model and business logic (or service layer)



9/97

■ Navigation direction

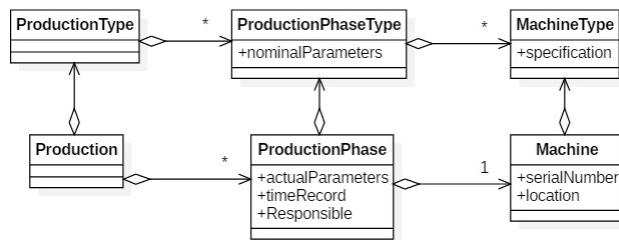
- bidirectional in the conceptual model, but better restricted in the implementation to reduce the effort in Java code, Tests, and Object Relational Mapping
- in principle, determined by expected Use Cases for the sake of efficiency
- e.g. identify all the exams registered by a Professor
- in the scope of a containing architecture can be inverted: ORMapping ensures that only "relevant" objects will be alive, reducing the issue of efficiency, and delegating to proper queries and database indexing the possible optimization



10/97

■ Composing types

- avoids subclassing (which is complex for mapping)
- may enable *type mapping* that makes the type processable as an instance (much developed later, in reflection architectures)
- ... may require correction, as specialized ops or fields may be required



## Software Engineering - A.A. 19/20

### Elements of Object Oriented Analysis



**Enrico Vicario**

Dipartimento di Ingegneria dell'Informazione  
Laboratorio Tecnologia del Software – stlab.dinfo.unifi.it  
Università di Firenze

[enrico.vicario@unifi.it](mailto:enrico.vicario@unifi.it), [stlab.dinfo.unifi.it/vicario](http://stlab.dinfo.unifi.it/vicario)

1/97

### Where are we headed now?

- Enlarge scope and abstraction
  - up to now: Java, Idioms, Design Patterns, micro architectures of few classes
  - next step: domain logic of a case
- What is a Domain Logic?
  - "an object model of the domain that incorporates both behavior and data" [Fowler, Architectural Patterns]
  - a layer of objects that model the business area you're working in
  - with objects that mimic the data in the business
  - and objects that capture the rules the business uses
  - a well understood component, appearing in any SW architecture
  - (aka domain model, business logic, ...)
- Construction involves various steps and tools (here just sketched)
  - requirements (elicitation and) analysis
  - identification of objects, classes, operations
  - semi-formal representation combining text with UML diagrams (mainly class diagr., use case diagr. and templates)
  - transition from conceptual description to specification and design
  - (integration with surrounding architecture and full stack technology)

2/97

- processo di accreditamento della qualità di un Ateneo.
  - Un Ateneo è soggetto a un insieme di requisiti di qualità, ciascuno riferito alla sede nel suo intero, oppure a uno specifico corso di studi, oppure a uno specifico dipartimento. I requisiti prescritti dalla normativa vigente sono denominati r1, r2 e r4.a per la sede, r3 per i corsi di studio, r4.b per i dipartimenti.
  - Ciascun requisito è articolato in un insieme di indicatori, per ciascuno dei quale sono identificati un numero di punti di attenzione.
  - A ciascun punto di attenzione è associata una valutazione, che può recare una prescrizione, la quale a sua volta può essere una raccomandazione o una condizione.
  - Requisiti, indicatori e punti di attenzione sono specificati in una varietà di fonti documentali che includono: i quadri SUA di un cds, le pagine web di un cds, l'ordinamento e il regolamento di un cds, il rapporto di riesame ciclico, la scheda di monitoraggio annuale, le consultazioni con le parti interessate, la valutazione della didattica, la relazione della commissione paritetica.

13/97

- Partecipano al processo di gestione dell'accreditamento un numero di gestori della qualità di Ateneo, una commissione esterna di esperti valutatori, e l'agenzia di valutazione nazionale ANVUR.
- I gestori della qualità di Ateneo hanno il compito di produrre la documentazione richiesta, poi acquisire la valutazione rilasciata dalla commissione degli esperti valutatori, e produrre delle controdeduzioni.
- La commissione degli esperti valutatori: acquisisce la documentazione; elabora una valutazione attribuendo un punteggio a ciascun punto di interesse e se necessario (in presenza di valutazioni negative) prescrive una raccomandazione (per valutazioni d debole insufficienza) o una condizione (per gravi insufficienze), infine calcola in modo automatico la valutazione degli indicatori e verifica in modo automatico il soddisfacimento dei requisiti; produce infine una relazione che viene trasmessa all'Ateneo.
- L'agenzia ANVUR acquisisce la relazione della commissione dei valutatori e poi le controdeduzioni dell'Ateneo, e sulla base di questi rilascia un suo giudizio di accreditamento della qualità dell'Ateneo.
- Si descrivano
  - i casi d'uso delineati nella descrizione attraverso uno use case diagram.
- Si definisca
  - una logica di dominio in prospettiva di specifica/implementazione capace di abilitare le funzioni identificate nei casi d'uso.

14/97

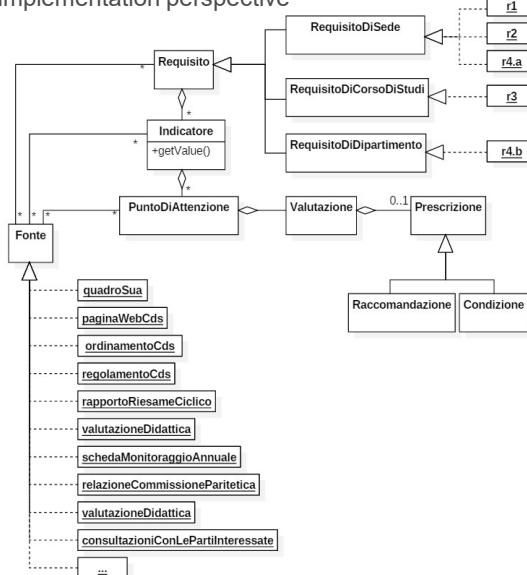
## ■ Use Case Diagrams: capture users roles and system functions



15/97

## ■ Class Diagram: capture domain entities and their relations

- in conceptual/specification/implementation perspective



16/97

- Object oriented analysis is about characterizing some conceptual domain in terms of classes, objects, and relationships
  - often is an early step in the process of building a SW application
  - but can also serve per-se, to characterize some conceptual domain
  - in both cases, a model has a pragmatic aim
- UML class diagrams provide an effective and practical notation
- ... but the real problem is
  - how to identify objects and classes to represent, and their attributes and relationships, and their operations
  - which in turn is much about identifying and allocating responsibilities, according to principles of cohesion and decoupling
- If all this is headed to build some SW
  - a subsequent problem is also how to manage the transition from conceptual to implementation models
  - ... which somehow conditions the way how concepts are initially identified

*TBD: Contrast all this with structured development*

- Starts with an initial Problem Statement
  - should be provided by the customer, yet, sometimes is suggested by the developer (in despair)
  - informal, incomplete, expressed in natural language
- ... from which an Analysis Model is elicited
  - a concise and non-ambiguous representation of essential concepts
  - something that can be discussed and that provides answers
  - replaces the problem statement
  - requires understanding of the problem statement, and abstractions
  - (in general, modeling also serves to get insight on the subject)
- The Analysis Model is composed by 3 orthogonal views
  - the Object model: the static structure and organization of classes
  - the Dynamic model: functions performed, how the state can evolve
  - the Functional model: the algorithms used for data processing

## A brief aside on SW engineering, before OMT, and also after

- *TBD: can be deferred to the time of software processes*
- **SW Requirements Specification**  
(see later about SRS - Mil-STD-498 - 1994 )
  - functional requirements: is about what a system is to do
  - architectural requirements: is about the structure of the solution
  - quality requirements: is about qualities in the function and structure  
(see later about ISO-9126 and ISO-25010)
- **Separation of functional, structural and quality perspectives is crucial**
  - a way to attain orthogonal decomposition of complexity
- **The first focus shall be on functional requirements**
  - architectural and quality requirements have not self-concreteness
- **Fuctional and structural perspectives have different relevance in different stages and for different roles**
  - the functional perspective is natural to users and customers, and prevails in the analysis stage
  - the structural perspective is for developers, and prevails in the design stage

*TBD:about rights and duties of developers and customers*

10/97

## Example of what is a problem statement (1/2)

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- (CONTINUA)

10/97

## Example of what is a problem statement (2/2)

- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

11/97

## Capture classes in a class diagram

- Objects and classes are conveniently represented by a class diagram
- ... with accompanying text
  - ... "la Rete ha più Sedi fra cui una centrale. Ciascuna Sede conserva degli Articoli, che sono la concreta realizzazione di uno stesso prodotto" ...
  - remark about Upper cases and plural terms, and the correspondence between text and diagram
- Provides a non-ambiguous basis for discussion
  - ... l'Articolo è localizzato nella Rete o in una Sede?
  - ... la Descrizione è riferita all'opera o anche all'oggetto fisico conservato?

12/97

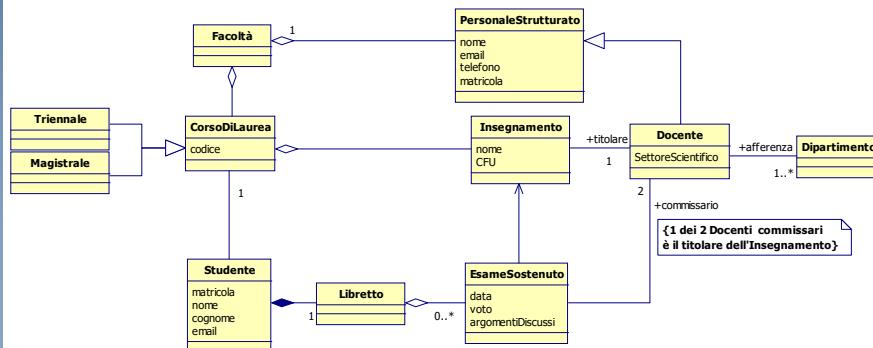
## An aside on class diagrams in the conceptual perspective

- Class Diagrams can attain 3 levels of abstraction (perhaps 2+1)
  - Implementation perspective
    - rappresenta il modo con cui è realizzato un sistema SW
    - Un oggetto nella rappresentazione corrisponde a un oggetto in un linguaggio OO (Java, c++, ...)
  - Specification perspective
    - Rappresenta le astrazioni che saranno realizzate in un sistema SW
    - Un oggetto della specifica è tipicamente realizzato attraverso più oggetti dell'implementazione
    - Non necessariamente viene realizzata con un linguaggio OO
  - Conceptual perspective
    - Descrive le entità di un dominio applicativo
    - Non necessariamente rappresentate in una realizzazione SW
- 2 representation intents
  - Specification: of something that shall be implemented
  - Description: of something that already exists
- Different combinations of 3x2 along the lifecycle of a SW system

13/42

## Class diagrams in the conceptual perspective – an example

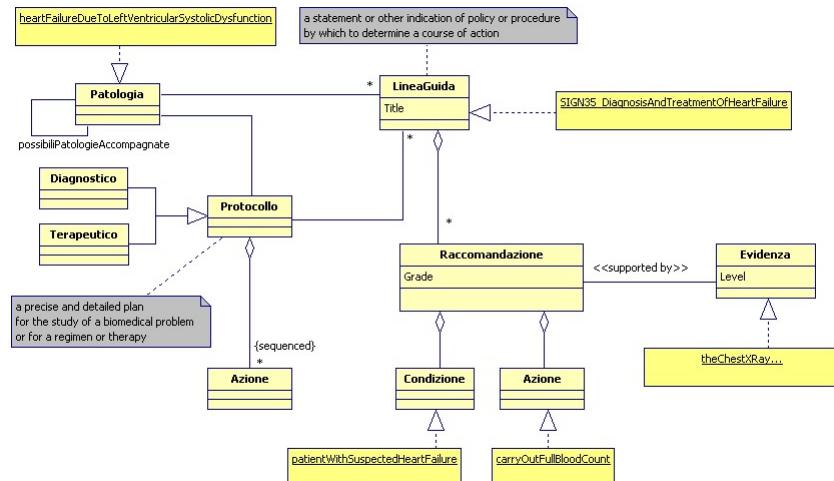
- Classes capture the concepts of some domain
  - Independently from their possible implementation within an information system
- Primarily intended for analysis and requirements definition
  - E.g. concepts involved in the process of examinations



14/42

## Class diagrams in the conceptual perspective – an example

- A class diagram in conceptual perspective: medical guidelines and protocols

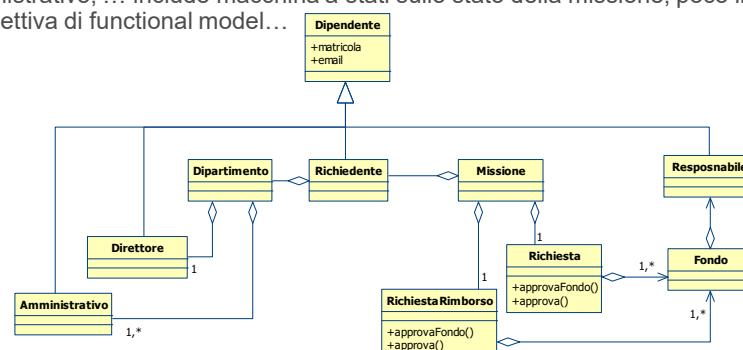


15/42

## to be skipped now, just one more example

- processo delle missioni

- ... docente, richiesta autorizzazione, fondo responsabile del fondo direttore, amministrativo, ... include macchina a stati sullo stato della missione, poco in prospettiva di functional model...



alcune note:

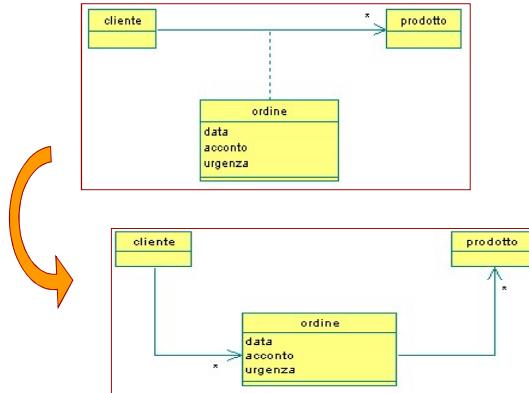
- Missione è un'astrazione di livello superiore ed è il bandolo del diagramma
- Dipendente è un concetto utile in prospettiva di implementazione, ma irriducibile in prospettiva concettuale
- il Fondo indicato alla Richiesta può essere modificato al Rimborso
- al modello si associa una macchina a stati del processo della Missione
- in ogni Dipartimento sono identificati 1 o più Amministrativi responsabili delle Missioni
- il modello sottende l'esistenza dei Controllori delle pagine
- Direttore e Amministrativo non avranno metodi, a meno di non storizzarli, se ne usa l'email
- ...

16/97

## Some more constructs: association classes

### Association classes

- Provide attributes and also methods for associations
- Used in the specification (and conceptual) perspective to drive implementation

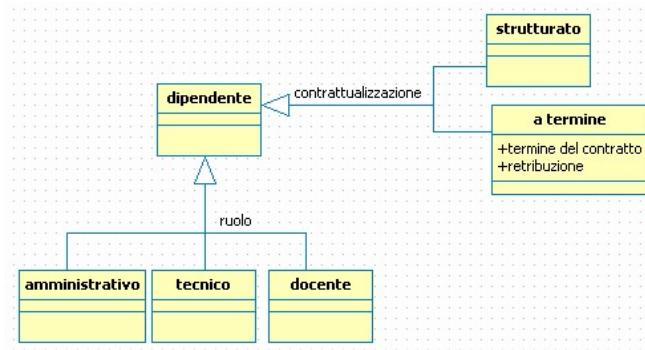


- Much useful also in the conceptual perspective

17/42

## Some more constructs: independent generalizations

- Generalization may refer to multiple orthogonal directions, with a discriminator for each direction
  - E.g. medici pediatri o di base / convenzionati o esterni
  - Requires design in the step to implementation

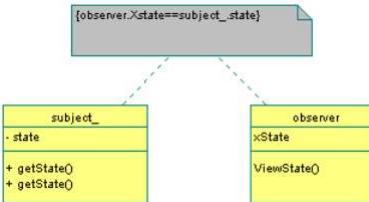


- Much useful also in the conceptual perspective

18/42

## Some more constructs: constraints, comments, annotations

- Constraints
  - Within curly brackets {}
- Comments
  - Callouts boxes with free text
- Annotations
  - Supported by the editing tool



- Much useful to avoid over-specification and detail clutter
  - Also in the conceptual perspective

19/42

## Which are the right modeling artifacts to produce?

- Different artifacts
  - suited for different stages of the SW lifecycle,
  - with different responsibility for parties (developer, customer, ...)
  - Enable incremental refinement and natural transition, possibly supported by some methodology (e.g. Iconix)
- The choice depends on process characteristics
  - duration of development process, number of involved roles and people, contractual relation among parties, ...
  - Cost of documentation maintenance, reverse engineering and Model Driven Development
  - expected lifecycle of the product, duration, maintenance and evolution
- ... which in turn should depend on the characteristics of the product
  - complexity of functions, size of implementation structure, quality attributes
- Much about agility vs discipline
  - I.e. about the trade-off between lightness and robustness
  - Well represented in eXtreme Programming vs Unified Process
  - (More answers later, when dealing with SW lifecycle)

20/42

- **Core elements**
  - Classes
    - Avoid hierarchical classification, easy to add but stiff to let evolve
    - ... and in any case work by generalization better than specialization
  - Associations
    - also characterized by names and cardinality
  - Attributes
    - Just a few, with no aim of completeness,  
often representing concepts that will later be delegated to outer classes
- **Operations emerge much later in the process of analysis and design**
  - More related to specification than to conceptual analysis
  - Closely coupled with use-cases (see later)
  - Much more difficult to identify and allocate
    - TBD: add a remark on deriving operations from data flow diagrams,  
or through robustness analysis in the iconix methodology*
- **Identification of objects and classes is a task for the analyst**
  - Not for the customer,  
who remains responsible for interpreting the model, criticizing, and validating

- An object
  - is provided with a state (attributes),  
and with operations that elaborate on the state
  - It has identity
  - it is something for what we want to have a representation available
- A class
  - is the type of one or more objects
  - some classes have a single instance (e.g. il Catalogo)
  - some have many (e.g. un Articolo)
- A nominalistic approach
  - an object is usually a noun (sostantivo)
  - also an attribute is a noun
  - a Method is a predicate, and the object is its direct object  
(complemento oggetto)
    - *TBD: remark on names, between syntax and semantics  
"stat rosa pristina nomine, nomina nuda tenemus"*

123/97

- Possible Objects (and Classes)
  - Physical or tangible objects: un Articolo, ...
  - A descriptor of some object: la Descrizione di un Articolo,...
  - A location: una Sede, ...
  - A transaction that shall be recorded: un Sollecito, un Prestito,...
  - A detail or a part of a transaction: la Tessera usata in un Prestito
  - The role of some people: Utente, PersonaleFrontOffice, ...
  - A container: Catalogo, Rete, ...
  - Something contained: Sede, ArticoloConservato, ...
  - An abstract object: Prestito, Articolo
  - An organizational unit: Sede, ...
  - An event: Restituzione (not in the model)
  - A process: la gestione dei Solleciti
  - A rule or policy: una Regola per la generazione di Solleciti
  - An index: Catalogo
  - A record: ArticoloConservato, ...
  - A document: Tessera, Commento, ...
- useful for beginners, or to begin ...

124/97

- **Attribute**
  - a value referred to an object: la collocazione di un ArticoloConservato
  - yet, this could become an autonomous object if modeled so as to assume complexity and state:  
e.g. Tessera could have initially been an attribute of Utente
- **Functoid**
  - a function encapsulated into a class
    - AlgoritmoRicerca, ha un metodo run(), non ha attributi (stato)
- **A vague abstraction, often missing a sharp definition**
  - "servizio di prestito"
  - may become an object if I'm interested in representing its concept, AND if I'm able to give it a specification (e.g. la carta dei servizi)
- **A concept pertaining to the implementation**
  - Algoritmo, funzione, CPU, database...
- **The entity that invokes a method**
  - "the doctor administers a drug to the patient"  
... is much probably a method of Patient rather than of Doctor
  - the page controller where the action is exposed

125/97

- L'identificazione delle responsabilità e loro allocazione, è il vero cuore della analisi OO
  - riguardare gli oggetti come entità capaci di assumere responsabilità anche collaborando con altre entità
- La Responsabilità si realizza nel fare e nel conoscere
  - effettuare una azione (creare un oggetto, eseguire un calcolo,...)
  - delegare una azione (provocare una azione in altri oggetti)
  - conoscere i propri dati privati
  - conoscere oggetti correlati
  - conoscere cose che può derivare o calcolare
- Tipicamente codificato nel design
  - Responsibility Driven Design (RDD)
  - General Responsibility Assignment Software Patterns (GRASP)
- Applicabile anche in analisi

126/97

## Questions and remarks (out of order)

- Is Catalogo an object?
  - has the Catalogo a state?
  - can this state be read or modified through some operation?
- The number of active Prestito for a Utente at a certain time is an object, or an attribute, or a value that can be computed?
  - if the value is logged, then becomes an object (storicizzazione, log)
  - otherwise, it is more probably an attribute or even just a value
- Do not mess reality and information about reality
  - utente can be something in the reality, or its descriptor in some application
  - both views can be relevant for the model, depending on the aim (conceptual or specification model)
  - ... about the material process and its digital twin
    - *TBD: Sorting is not work, it's mental work (T.Pynchon)*
- Operations are allocated to the objects that they modify
  - In the reality, registration of a Prestito is performed by Personale, but, the action modifies the state of the Utente (or its Tessera); So, the operation is allocated to Utente (or its Tessera)
  - do not mess interface page controllers with objects they use

127/97

## Example: a problem statement (1/2)

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- (CONTINUA)

128/97

### Example: a problem statement (2/2)

- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

129/97

### Discover Objects and functions - 1/2

X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.

X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli Utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria. La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.

Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.

130/97

X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.

Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.

Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.

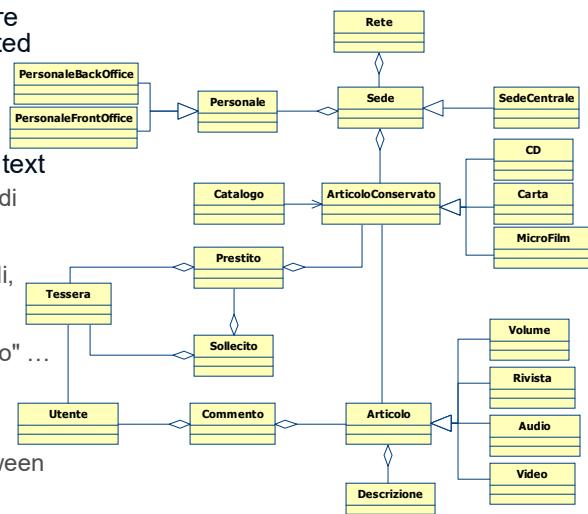
Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.

Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.

Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

131/97

- Objects and classes are conveniently represented by a class diagram



- ... with accompanying text

- ... "la Rete ha più Sedi fra cui una centrale. Ciascuna Sede conserva degli Articoli, che sono la concreta realizzazione di uno stesso prodotto" ...

- remark about Upper cases and plural terms, and the correspondence between text and diagram

- Provides a non-ambiguous basis for discussion

- ... l'articolo è localizzato nella rete o in una sede?
- La Descrizione è riferita all'opera o anche all'oggetto fisico conservato?

132/97

■ **Star UML**

- Open source, easy to use and quite complete
- for long only for Windows, now also for Linux and Mac;
- <http://staruml.io/>
- (my first choice, now at version 3.0.2)

■ **Visual Paradigm**

- Java Based and thus cross platform
- Community Edition freely distributed
- <http://www.visual-paradigm.com/>
- (Now perhaps the right first choice)

■ **Argo UML**

- Open Source, Java Based and thus cross platform
- <http://argouml.tigris.org/>

■ Martin Fowler, "UML Distilled: a brief guide to the standard object modeling language", third edition (Addison Wesley).

■ Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., Object-Oriented Modeling and Design, Prentice-Hall, 1991.

■ Martin Fowler, "Analysis Patterns, reusable object models", Addison Wesley 1997.

## One more example: assignment of a contract

TBD: meglio farlo con gli use case diagrams e gli activity diagrams

- Procedura conferimento di un contratto avviata da un docente
  - Fase del bando
  - Selezione e arruolamento
  - Chiusura e rinnovo
- Passi della analisi
  - Specifica preliminare
  - Identificazione di classi, attori e casi d'uso
  - Organizzazione in un activity diagram

35/97

## Specifiche preliminari: fase del bando

- Un docente presenta richiesta di attivazione al Consiglio di Dipartimento, indicando il titolo, il tema della ricerca, i prerequisiti (e.g. tipo di laurea, votazione, esperienza pregressa in specifici ambiti, ...), il periodo di svolgimento, la motivazione della necessità di un contratto, i fondi su cui intende finanziare il contratto.
- La richiesta viene immessa nell'ordine del giorno della prossima riunione del consiglio di dipartimento a cui afferisce il docente.
- Il Consiglio nomina la commissione formata dal docente stesso (nel ruolo di responsabile scientifico) e altri due docenti afferenti a questo o ad altro dipartimento. Nella nomina possono essere indicati fino a due membri supplenti.
- I docenti incaricati ricevono notifica, e accettano l'incarico (eccezione: se non accettano la richiesta viene inviata ai membri supplenti)
- Il docente responsabile dell'assegno comunica ai membri della commissione la data della prova. La data deve essere almeno 40 giorni avanti.
- Il bando per l'assegno viene comunicato all'ufficio Ricerca dell'Ateneo, che pubblica il titolo, la descrizione della ricerca, i requisiti di partecipazione, i termini per la presentazione della domanda.
- L'Ufficio Ricerca anche invia per via telematica ai membri della commissione la convocazione per la prova di selezione.

36/97

### Specifiche preliminari: selezione e arruolamento

- Ciascun candidato, presenta domanda per via telematica inserendo i propri dati anagrafici, descrivendo titoli e allegati, allegando files. Il sistema produce una versione elettronica della domanda (che include l'elenco degli allegati ma non gli allegati stessi). Il candidato stampa la domanda e la presenta in forma cartacea firmata e con marca da bollo all'ufficio Ricerca.
- Entro la data della prova di selezione, l'ufficio verifica quali domande possono essere ammesse in base a vari criteri: effettiva ricezione della domanda in forma cartacea, adeguatezza dei titoli presentati, ...
- L'Ufficio comunica al Responsabile scientifico l'elenco dei candidati ammessi e invia ai membri della commissione un promemoria per la prova di selezione assieme a un fac-simile del verbale.
- In occasione della selezione viene redatto un verbale. Il sistema pre-compila tutte le parti già note (incluso descrizione del titolo, membri della commissione, nomi dei partecipanti, ...) . I commissari compilano on-line la valutazione dei titoli e del colloquio per ciascun candidato che si è presentato alla prova, decretano il vincitore della selezione.
- Il sistema comunica a tutti i partecipanti un estratto del verbale che include l'indicazione del vincitore. Il Sistema comunica al vincitore l'invito a presentarsi all'ufficio contratti per la firma.
- Il sistema notifica all'ufficio contratti i termini del contratto e il riferimento del responsabile scientifico e della unità amministrativa.
- Quando il vincitore firma il contratto, l'ufficio contratti notifica l'avvio del contratto di collaborazione all'ufficio stipendi.

137/97

### Specifiche preliminari: chiusura e rinnovo

- Quando mancano 60 giorni al termine del contratto, il sistema invia un avviso al responsabile della ricerca e al titolare del contratto.
- Il titolare del contratto invia una relazione delle attività svolte che viene controfirmata dal responsabile scientifico e accompagnata da una valutazione.
- La relazione e' inviata all'Ufficio ricerca in forma elettronica entro 30 giorni dalla data di scadenza del contratto.
- Il responsabile scientifico ha facoltà di richiedere il rinnovo del contratto allo stesso soggetto che ne attualmente titolare per un periodo pari a quello del primo incarico, senza ulteriore approvazione del consiglio di dipartimento e senza ulteriore bando e selezione. In tal caso il sistema invia al titolare del contratto una notifica a cui il titolare risponde indicando la sua eventuale accettazione; la accettazione e' inviata all'ufficio stipendi.
- Il titolare del contratto può recedere inviando una comunicazione all'ufficio Ricerca. Ne ricevono notifica il responsabile scientifico e l'ufficio stipendi.
- Viceversa anche il responsabile scientifico può chiedere l'interruzione del contratto dove risultino applicabili i criteri indicati nel contratto stesso. La richiesta è notificata all'ufficio affari legali, all'ufficio ricerca, all'ufficio stipendi, e al titolare del contratto. La procedura di rescissione non è qui descritta.

138/97

## Identificazione oggetti e attributi: fase del bando

- Un docente presenta richiesta di attivazione al Consiglio di Dipartimento, indicando il titolo, il tema della ricerca, i prerequisiti (e.g. tipo di laurea, votazione, esperienza pregressa in specifici ambiti, ...), il periodo di svolgimento, la motivazione della necessità di un contratto, i fondi su cui intende finanziare il contratto. La richiesta viene inserita nell'ordine del giorno della prossima riunione del consiglio di dipartimento a cui afferisce il docente.
- Il Consiglio nomina la commissione formata dal docente stesso (nel ruolo di responsabile scientifico) e altri due docenti afferenti a questo o ad altro dipartimento. Nella nomina possono essere indicati fino a due membri supplenti.
- I docenti incaricati ricevono notifica, e accettano l'incarico (eccezione: se non accettano la richiesta viene inviata ai membri supplenti)
- Il docente responsabile dell'assegno comunica ai membri della commissione la data della prova di selezione. La data deve essere almeno 40 giorni avanti.
- Il bando per l'assegno viene comunicato all'ufficio Ricerca dell'Ateneo, che pubblica il titolo, la descrizione della ricerca, i requisiti di partecipazione, i termini per la presentazione della domanda.
- L'Ufficio Ricerca anche invia per via telematica ai membri della commissione la convocazione per la prova di selezione.

139/97

## Identificazione oggetti e attributi: selezione e arruolamento

- Ciascun candidato, presenta domanda per via telematica inserendo i propri dati anagrafici, descrivendo titoli e allegati, allegando files. Il sistema produce una versione elettronica della domanda (che include l'elenco degli allegati ma non gli allegati stessi). Il candidato stampa la domanda e la presenta in forma cartacea firmata e con marca da bollo all'ufficio Ricerca.
- Entro la data della prova di selezione, l'ufficio verifica quali domande possono essere ammesse in base a vari criteri: effettiva ricezione della domanda in forma cartacea, adeguatezza dei titoli presentati, ...
- L'Ufficio comunica al Responsabile scientifico l'elenco dei candidati ammessi e invia ai membri della commissione un promemoria per la prova di selezione assieme a un fac-simile del verbale.
- In occasione della selezione viene redatto un verbale. Il sistema pre-compila tutte le parti già note (incluso descrizione del titolo, membri della commissione, nomi dei partecipanti, ...) . I commissari compilano on-line la valutazione dei titoli e del colloquio per ciascun candidato che si è presentato alla prova, decretano il vincitore della selezione.
- Il sistema comunica a tutti i partecipanti un estratto del verbale che include l'indicazione del vincitore. Il Sistema comunica al vincitore l'invito a presentarsi all'ufficio contratti per la firma.
- Il sistema notifica all'ufficio contratti i termini del contratto e il riferimento del responsabile scientifico e del dipartimento
- Quando il vincitore firma il contratto, l'ufficio contratti notifica l'avvio del contratto di collaborazione all'ufficio stipendi.

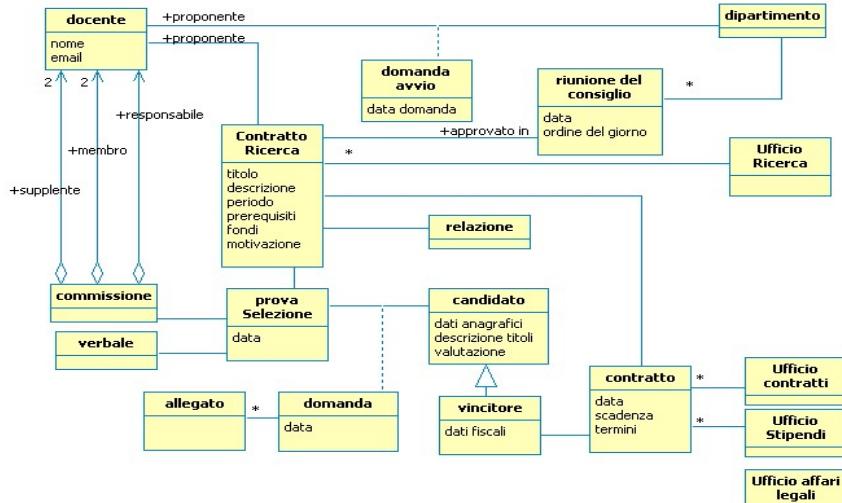
140/97

## Identificazione oggetti e attributi: chiusura e rinnovo

- Quando mancano 60 giorni al termine del contratto, il sistema invia un avviso al responsabile della ricerca e al titolare del contratto.
- Il titolare del contratto invia una **relazione delle attività** svolte che viene controfirmata dal responsabile scientifico e accompagnata da una valutazione.
- La relazione e' inviata all'Ufficio ricerca in forma elettronica entro 30 giorni dalla data di scadenza del contratto.
- Il responsabile scientifico ha facoltà di richiedere il rinnovo del contratto allo stesso soggetto che ne attualmente titolare per un periodo pari a quello del primo incarico, senza ulteriore approvazione del consiglio di dipartimento e senza ulteriore bando e selezione. In tal caso il sistema invia al titolare del contratto una notifica a cui il titolare risponde indicando la sua eventuale accettazione; la accettazione e' inviata all'ufficio stipendi.
- Il titolare del contratto può recedere inviando una comunicazione all'ufficio Ricerca. Ne ricevono notifica il responsabile scientifico e l'ufficio stipendi.
- Viceversa anche il responsabile scientifico può chiedere l'interruzione del contratto dove risultino applicabili i criteri indicati nel contratto stesso. La richiesta è notificata all'**ufficio affari legali**, all'ufficio ricerca, all'ufficio stipendi, e al titolare del contratto. La procedura di rescissione non è qui descritta.

41/97

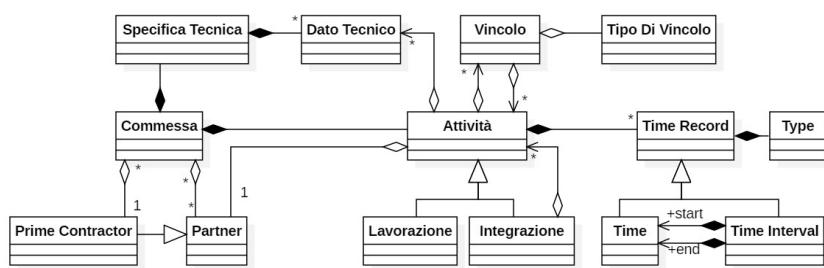
## Class diagram concettuale



42/97

- gestione di una fornitura in cui partecipano più partners.
  - Una fornitura viene eseguita da un insieme di partners, uno dei quali opera come prime contractor. La fornitura è caratterizzata da una specifica tecnica, che raccoglie una varietà di dati tecnici, anche di diverse tipologie.
  - Nell'esecuzione della fornitura vengono svolte un insieme di attività, ciascuna delle quali integra in modo ricorsivo (sub-)attività e lavorazioni di vario tipo.
  - Ciascuna attività fa riferimento ad alcuni dati tecnici e ha dei vincoli (e.g. di precedenza o di non concorrenza) rispetto ad altre attività. Ciascuna attività è associata a un insieme di dati temporali, che caratterizzano il tempo di avvio e di conclusione in diverse prospettive (nella pianificazione iniziale, nella previsione elaborata ad una certa data, nei fatti realizzati quando l'attività è completata).
  - L'applicazione permette al prime contractor di specificare i partners, specificare le attività pianificate con i loro mutui vincoli di precedenza e con la specificazione dell'intervallo nel quale la lavorazione dovrebbe essere effettuata, assegnare ciascuna attività ad un partner. A sua volta ciascun partner può aggiungere informazione sui tempi di avanzamento, aggiungendo la data di effettivo avvio o conclusione di un'attività, la previsione sul tempo di completamento.
- Si descrivano i casi d'uso delineati attraverso uno use case diagram.
- Si definisca una logica di dominio in prospettiva di specifica/implementazione che abiliti le funzioni identificate nei casi d'uso.

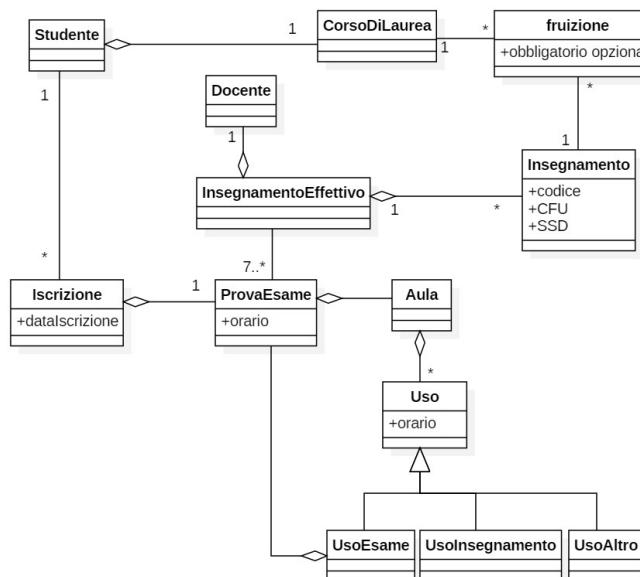
143/97



144/97

- programmazione delle prove di esame.
  - Un insegnamento è identificato da un codice, ha un settore disciplinare, un numero di CFU, ed è frutto da uno o più corsi di laurea, in modo opzionale o obbligatorio. Un insegnamento effettivo è attribuito a un docente e copre uno o più insegnamenti.
  - Per ogni insegnamento effettivo sono previsti nel corso dell'anno un numero di esami, non inferiore a 7. Ciascun esame ha un orario e un'aula.
  - Ciascuna aula ha una capienza e un insieme di prenotazioni attive che identificano gli orari ancora disponibili. Ciascuna prenotazione è associata a diversi tipi di uso (e.g. insegnamento, prova di esame, altro). Se il tipo è prova di esame, l'uso identifica anche l'esame a cui è riferito.
  - Il docente di un insegnamento effettivo può visionare la disponibilità di aule, e programmare le date degli esami selezionando l'aula di svolgimento tra quelle che sono disponibili. Il docente può vedere l'elenco degli studenti iscritti a ciascuna prova di esame.
  - Uno studente ha un numero di matricola ed è iscritto a un corso di laurea. Lo studente può vedere le date delle prove di esame per gli insegnamenti nel suo corso di laurea, e può iscriversi. Il sistema tiene traccia della data in cui è effettuata l'iscrizione.
- Si descrivano i casi d'uso delineati attraverso uno use case diagram.
- Si definisca una logica di dominio in prospettiva di specifica/implementazione che abiliti le funzioni identificate nei casi d'uso.

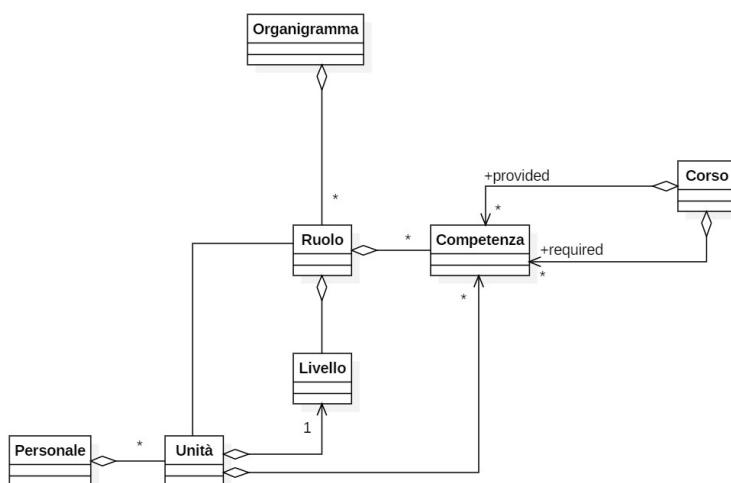
45/97



46/97

- organizzazione dei corsi di formazione per il personale di un'azienda.
  - Il personale di un'azienda è composto di un certo numero di unità, ciascuna inquadrata in un livello e ciascuna provvista di un insieme di competenze. L'organigramma dell'azienda identifica un insieme di ruoli, ciascuno associato ad un livello di inquadramento richiesto e a un insieme di competenze necessarie per un efficace svolgimento, e associa a ciascun ruolo una unità di personale. Un corso di formazione fornisce un insieme di competenze, e richiede come prerequisito la disponibilità di un altro insieme di competenze.
  - Un analista può identificare il debito di competenze, ovvero i ruoli che sono coperti da figure che non sono dotate di tutte le competenze richieste, e con questo definire i contenuti di corsi di formazione che possono sopperire. Una unità di personale può identificare corsi di formazione utili a coprire il proprio ruolo, o altri ruoli che sono ad oggi coperti con debito di competenze e può iscriversi a un corso.
- Si descrivano i casi d'uso delineati attraverso uno use case diagram.
- Si definisca una logica di dominio in prospettiva di specifica/implementazione che abiliti le funzioni identificate nei casi d'uso.
- Si delineino i metodi fondamentali per la realizzazione di alcuni casi d'uso caratterizzanti.

47/97



48/97

## Software Engineering for Embedded Systems - A.A. 19/20

### Use case diagrams and templates in the specification of functional requirements



Enrico Vicario

Dipartimento di Ingegneria dell'Informazione  
Laboratorio Tecnologie del Software – stlab.dinfo.unifi.it  
Università di Firenze

enrico.vicario@unifi.it, stlab.dinfo.unifi.it/vicario

1/42

## Casi d'uso per la definizione di requisiti funzionali

- Requisiti funzionali
  - catturano il comportamento atteso del sistema in termini di servizi, compiti, funzioni
  - bene inquadrati nel modello della Sw Requirements Specification SRS-232 (requisiti funzionali, architetturali, di qualità)
- Casi d'uso
  - pratica di successo e dominante nella rappresentazione di requisiti funzionali
  - in particolare, ma non solo, nello sviluppo object-oriented
  - conferiscono struttura e metodo all'idea di catturare i requisiti funzionali a partire da esempi e scenari di interazione e uso
- Intento della presentazione
  - Casi d'uso: diagrammi, templates e mock-ups
  - Esercizi di metodo sul loro uso: in un paio di scenari

2/42

■ **Caso d'uso (use case)**

- un insieme di interazioni tra il sistema e uno o più attori esterni al sistema finalizzate a un obiettivo (goal)
- Una funzione offerta dal sistema alla sua interfaccia

■ **Attore**

- Una entità (party) esterna al sistema che interagisce con esso
- Può essere una classe di utenti, un ruolo che gli utenti possono svolgere, un altro sottosistema, una parte del sistema non ancora sviluppata, un trigger temporale, ...
- Attore primario: ha un obiettivo che richiede il supporto del sistema
- Attore secondario: il sistema ne richiede l'assistenza

13/42

■ **Un caso d'uso è avviato da un attore, con un particolare obiettivo, e si conclude con successo quando l'obiettivo è raggiunto**

- cattura chi fa cosa con quale obiettivo attraverso quali passi

■ **Describe la sequenza di interazioni tra sistema e attori necessarie a realizzare il servizio richiesto dall'obiettivo**

- anche sequenze alternative con cui può essere raggiunto l'obiettivo
- anche sequenze che conducono a un fallimento nel completare il servizio per via di comportamenti anomali, errori o che altro
- Scenario: una specifica sequenza nell'esecuzione del caso d'uso

■ **Il sistema è trattato in prospettiva black-box/funzionale**

- le interazioni includono stimoli e risposte, descritte come percepite dall'esterno
- non tratta di come il sistema è realizzato (prospettiva white-box/strutturale)

■ **Una lista completa di casi d'uso specifica tutti i differenti modi di usare il sistema**

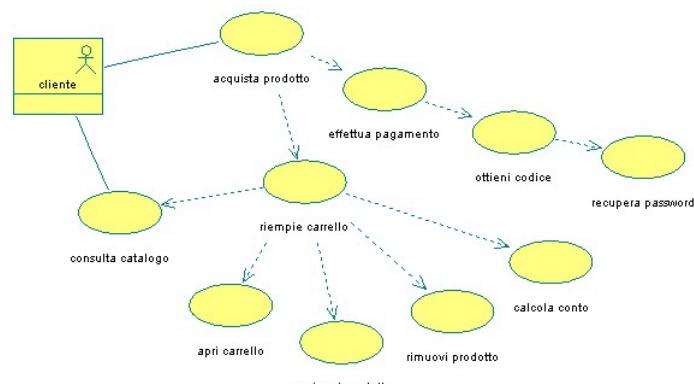
- definisce tutto il comportamento richiesto al sistema
- delimita lo scope del sistema

14/42

- Use case diagrams
  - visione di insieme, relazioni tra attori e casi d'uso, strutturazione dei casi d'uso
- Use case templates
  - Specifica testuale di singoli casi d'uso
- Mock-ups
  - Informazione e interazione sull'interfaccia grafica destinata agli utenti
  - *TBD: conviene introdurre da qui il concetto di page navigation diagram?*

15/42

- Documenta le funzionalità offerte dal sistema a una sua interfaccia



16/42

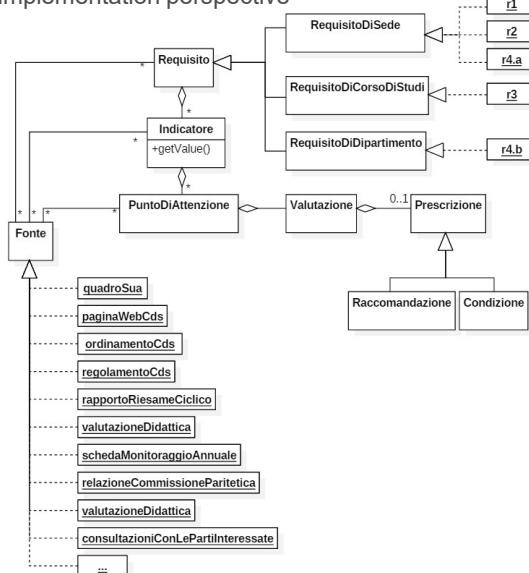
■ processo di accreditamento della qualità di un Ateneo.  
(side #1, about *Classes and Objects, and their relations*)

- Un Ateneo è soggetto a un insieme di **requisiti** di qualità, ciascuno riferito alla sede nel suo intero, oppure a uno specifico **corso di studi**, oppure a uno specifico **dipartimento**. I requisiti prescritti dalla normativa vigente sono denominati **r1, r2 e r4.a per la sede, r3 per i corsi di studio, r4.b per i dipartimenti**.
- Ciascun requisito è articolato in un insieme di **indicatori**, per ciascuno dei quale sono identificati un numero di **punti di attenzione**.
- A ciascun punto di attenzione è associata una **valutazione**, che può recare una **prescrizione**, la quale a sua volta può essere una **raccomandazione o una condizione**.
- Requisiti, indicatori e punti di attenzione sono specificati in una varietà di **fonti documentali** che includono: i quadri SUA di un cds, le pagine web di un cds, l'ordinamento e il regolamento di un cds, il rapporto di riesame ciclico, la scheda di monitoraggio annuale, le consultazioni con le parti interessate, la valutazione della didattica, la relazione della commissione paritetica.

7/97

■ Class Diagram: capture domain entities and their relations

- in conceptual/specification/implementation perspective



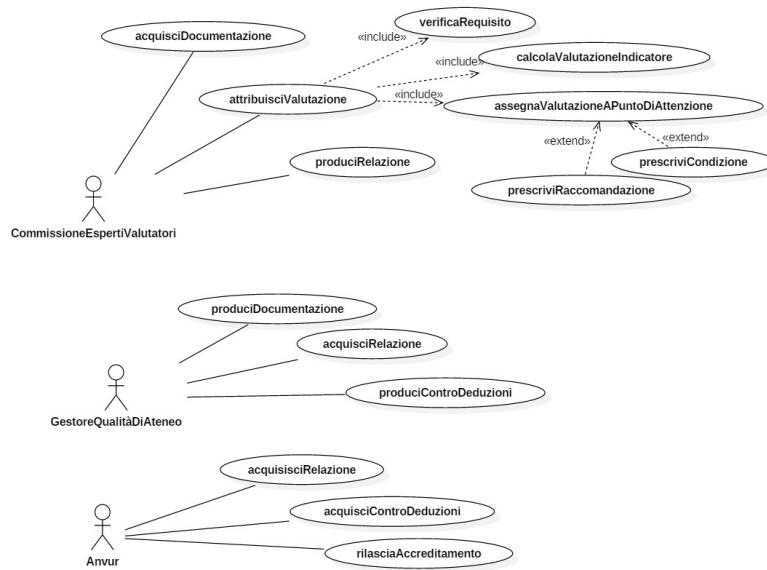
8/97

■ processo di accreditamento della qualità di un Ateneo.  
(side #2, about Roles and Use Cases)

- Partecipano al processo di gestione dell'accreditamento un numero di **gestori della qualità** di Ateneo, una **commissione esterna** di esperti valutatori, e l'agenzia di valutazione nazionale ANVUR.
- I gestori della qualità di Ateneo hanno il compito di **produrre la documentazione** richiesta, poi **acquisire la valutazione** rilasciata dalla commissione degli esperti valutatori, e **produrre delle controdeduzioni**.
- La commissione degli esperti valutatori: **acquisisce la documentazione**; **elabora una valutazione attribuendo un punteggio** a ciascun punto di interesse e se necessario (in presenza di valutazioni negative) **prescrive una raccomandazione** (per valutazioni d debole insufficienza) o **una condizione** (per gravi insufficienze), infine **calcola in modo automatico la valutazione** degli indicatori e **verifica in modo automatico il soddisfacimento dei requisiti**; **produce infine una relazione** che viene **trasmessa all'Ateneo**.
- L'agenzia ANVUR **acquisisce la relazione** della commissione dei valutatori e poi le **controdeduzioni** dell'Ateneo, e sulla base di questi **rilascia un suo giudizio** di accreditamento della qualità dell'Ateneo.

10/97

■ Use Case Diagrams: capture users roles and system functions



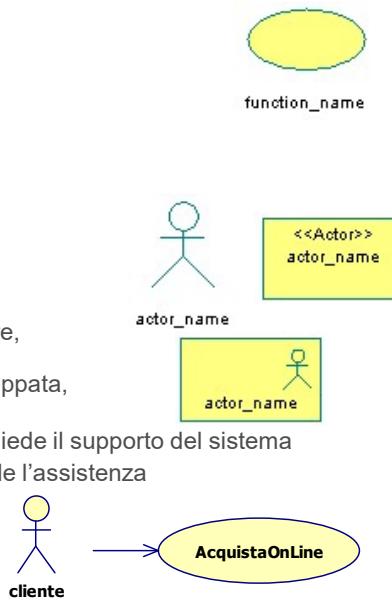
10/97

- models are pragmatic for some intent
  - classes are instrumental to use cases
  - but, a sharp specification of use cases requires a ground of concepts
  - a kind of chicken & egg problem, requiring iteration

11/42

- Caso d'uso (use case)
  - Una funzione, servizio o compito a cui partecipa il sistema
  - un insieme di interazioni tra il sistema e uno o più attori esterni al sistema finalizzate a un obiettivo (goal)
- Attore (actor)
  - Una entità (party) esterna al sistema, che partecipa a un caso d'uso
  - una classe di utenti, un ruolo che gli utenti possono svolgere, un altro sottosistema, una parte del sistema non ancora sviluppata, un trigger temporale, ...
  - Attore primario: ha un obiettivo che richiede il supporto del sistema
  - Attore secondario: il sistema ne richiede l'assistenza
- Relazione d'uso
  - Documenta una interazione tra un attore e il sistema

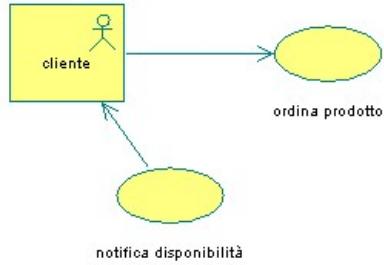
12/42



## Ancora sulla relazione d'uso

### Direzionalità

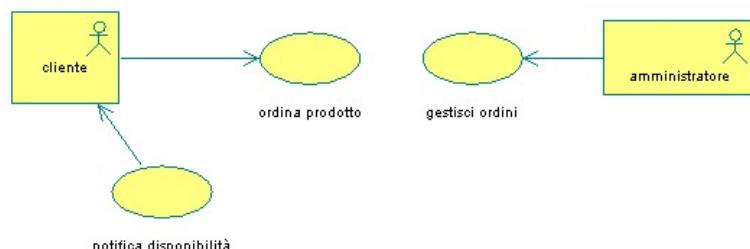
- L'attore interagisce con il caso d'uso fornendo stimoli e ricevendo risposte
- anche viceversa?  
e.g. il medico di base riceve notifica quando una visita specialistica prescritta è stata effettuata
- (... circa l'ortodossia)



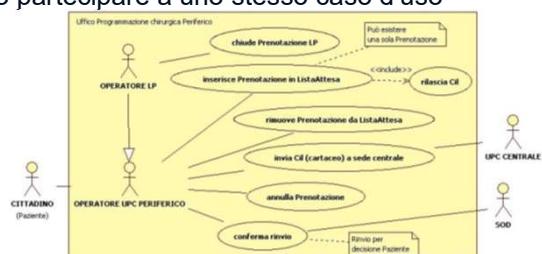
13/42

## Molteplicità di attori - 1/2

- Il sistema ha usualmente più attori coinvolti in diversi casi d'uso



- Più attori possono partecipare a uno stesso caso d'uso

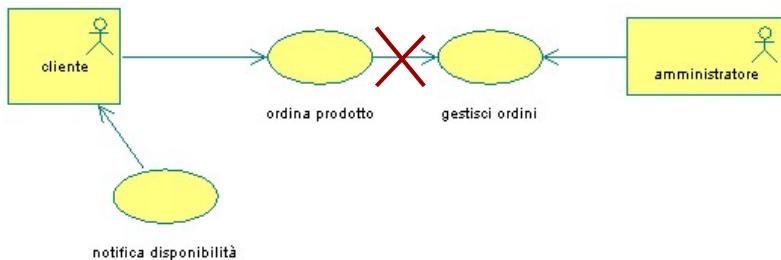


■ TBD: in the example, note the case confermaRinvio

14/42

## Breakdown funzionale vs dataflow

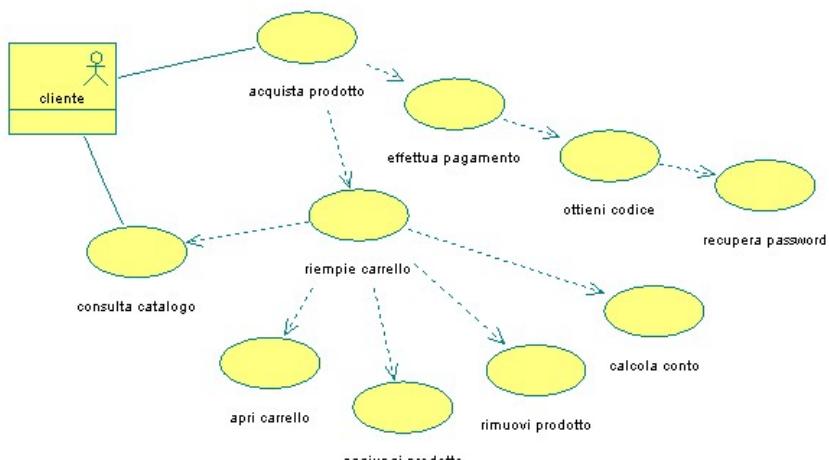
- Uno use case diagram non documenta il flusso dell'informazione
  - Né descrive relazioni di precedenza tra casi d'uso



16/42

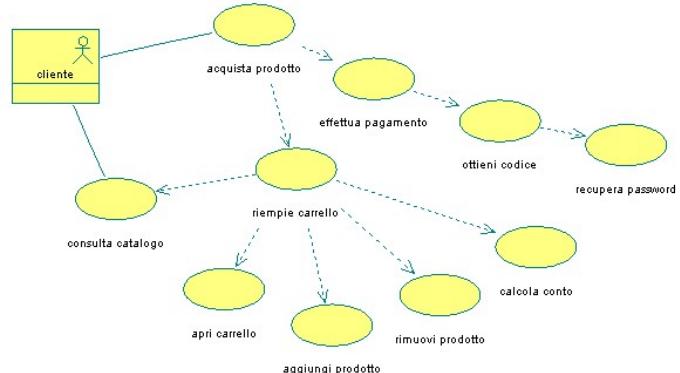
## Relazione di inclusione - 1/3

- <<include>> documenta la situazione in cui lo svolgimento di un caso include quello di uno o più sotto-casi
  - secondo un principio di breakdown funzionale



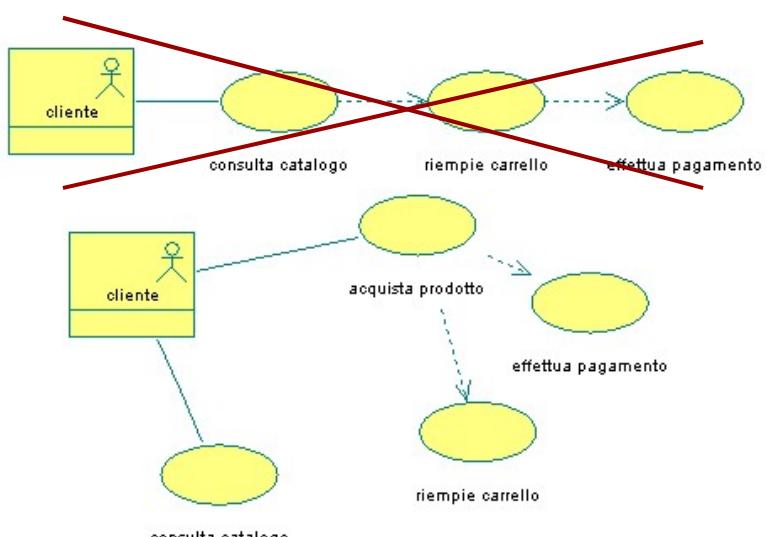
16/42

- Finalizzata a partizionare la complessità più che riusare le parti
  - Nella analisi, la documentazione, la realizzazione, il testing, ...
- Metafora nella prospettiva dello sviluppatore
  - per arrivare a dimostrare il caso devo avere implementato i sottocasi
  - il costo del caso aggrega quello dei sotto-casi
  - pianificazione e monitoraggio dell'avanzamento  
(in XP: planning game tra sviluppatore e utente)



17/42

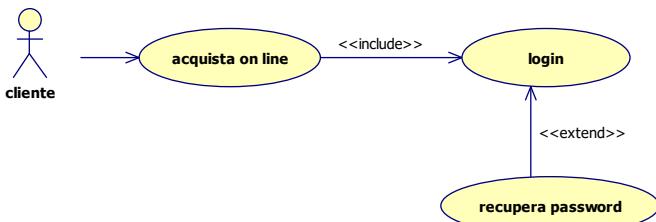
- La relazione di inclusione non documenta una procedura



18/42

### Relazione di estensione

- <<extend>> identifica sotto-casi eseguiti in circostanze speciali

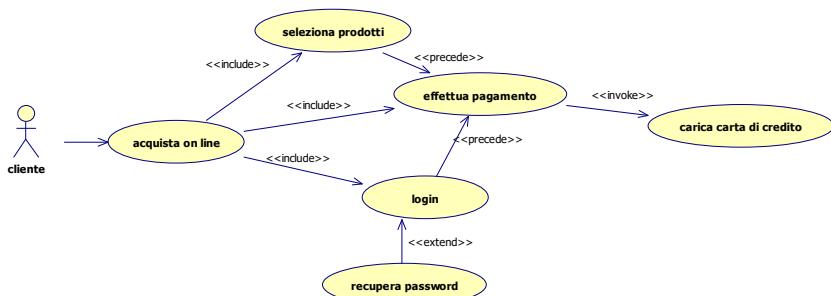


- Nella metafora dello sviluppatore
  - non essenziali per dimostrare il caso,  
ma necessarie per la completezza di funzionamento

19/42

### Invoke & Precede

- <<invoke>> e <<precede>> catturano relazioni in un modo diverso

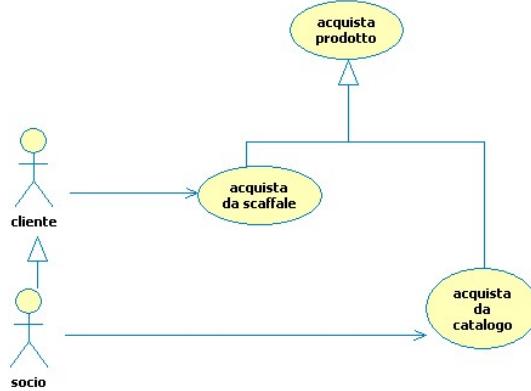


- <<include>>: descrive un sottocaso come parte di un caso più complesso
- <<extend>>: implica la definizione di un punto di estensione
- <<invoke>>: un caso ne invoca un altro
  - comprende <<include>> e <<extend>>
- <<precede>>: un caso deve terminare prima dell'avvio dell'altro
  - porta dentro un po' di procedura

20/42

## Relazione di generalizzazione

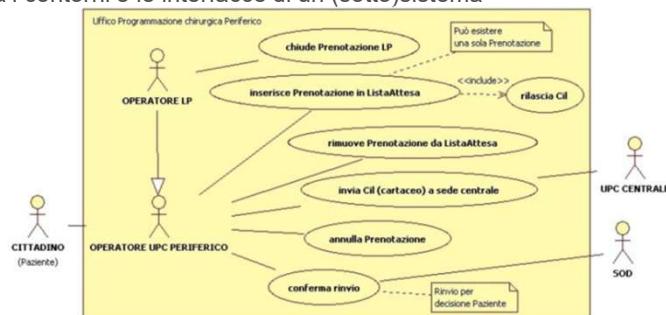
- Documenta la relazione di specializzazione (e sostituibilità)
  - Applicabile sia agli attori che ai casi d'uso
  - Da usare con giudizio



121/42

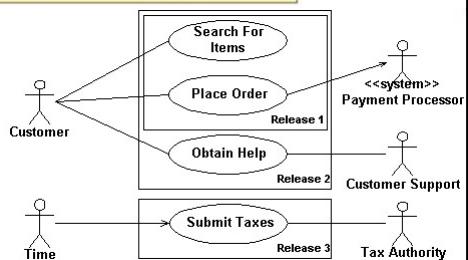
## Context and Scope

- System boundaries
  - delimita i contorni e le interfacce di un (sotto)sistema



### Scope di un caso

- il sistema, un sottosistema, una parte funzionale, una release, ...



122/42

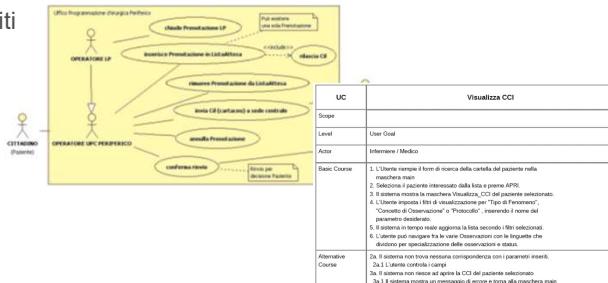
## Livelli di astrazione (level)

- High summary
  - non è una specifica "sufficiente", definisce un ambito da analizzare
  - "gestisci visite ambulatoriali", "gestisci sale operatorie", ...
- Summary
  - fornisce visione di insieme sui casi di livello user-goal
  - "CRUD visite", "CRUD ambulatori", ...
- User goal
  - specifica una interazione
  - è il livello che conviene dettagliare con template testuali
  - singoli passi a volte documentati con mock-ups
  - Il più usato
  - "prenota una visita"
- Function
  - specifica un dettaglio dell'interazione che ha qualche particolarità
  - usualmente non documentato in modo individuale
  - A volte documentato con mock-ups
  - "seleziona una prestazione su una mappa concettuale"

123/42

## 2/3: Use case templates

- ... comunque non bastano diagrammi e nomi
- I singoli casi d'uso sono documentati in forma testuale
  - Narrativa non-ambigua ma semplice
  - Riferita al linguaggio del dominio applicativo
  - Molto meglio se riferita a un modello concettuale esplicito, formalizzato e condiviso
  - Enfasi sul dialogo di interazione tra sistema e attori
- Devono abilitare il coinvolgimento di utenti e esperti di dominio
  - per seguire e validare i casi d'uso
  - per definire i requisiti



124/42

## Esempio della Cartella Clinica Informatizzata

### Template per il caso d'uso Visualizza CCI

| UC                 | Visualizza CCI                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Scope              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Level              | User Goal                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Actor              | Infermiere / Medico                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Basic Course       | <ol style="list-style-type: none"> <li>1. L'Utente riempie il form di ricerca della cartella del paziente nella maschera main</li> <li>2. Seleziona il paziente interessato dalla lista e preme APRI.</li> <li>3. Il sistema mostra la maschera Visualizza_CCI del paziente selezionato.</li> <li>4. L'Utente imposta i filtri di visualizzazione per "Tipo di Fenomeno", "Concetto di Osservazione" o "Protocollo", inserendo il nome del parametro desiderato.</li> <li>5. Il sistema in tempo reale aggiorna la lista secondo i filtri selezionati.</li> <li>6. L'utente può navigare fra le varie Osservazioni con le lingue che dividono per specializzazione delle osservazioni e status.</li> </ol> |
| Alternative Course | <ol style="list-style-type: none"> <li>2a. Il sistema non trova nessuna corrispondenza con i parametri inseriti.</li> <li>2a.1 L'utente controlla i campi</li> <li>3a. Il sistema non riesce ad aprire la CCI del paziente selezionato</li> <li>3a.1 Il sistema mostra un messaggio di errore e torna alla maschera main</li> </ol>                                                                                                                                                                                                                                                                                                                                                                        |

### Liturgia e dressing

125/42

▪ TBD: aggiungere un template in full-dress

## One more example - 1/2

|                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Use Case       | 2. Repairing Cellular Network history created 1/5/98 Derek Coleman, modified 5/5/98.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description    | Operator rectifies a report by changing parameters of a cell. sources [Operating Manual 1993], [Jones 1998].                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Assumptions    | Changes to network are always successful when applied to a network.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Actors         | Operator (primary)<br>Cellular network<br>Field maintenance engineer                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Steps          | <ol style="list-style-type: none"> <li>1. Operator notified of network problem.</li> <li>2. Operator starts repair session.</li> <li>3. REPEAT <ul style="list-style-type: none"> <li>3.1 Operator runs network diagnosis application.</li> <li>3.2 Operator identifies cells to be changed and their new parameter values.</li> <li>3.3 IN PARALLEL <ul style="list-style-type: none"> <li>3.3.1 Maintenance engineer tests network cells   </li> <li>3.3.2 Maintenance engineer sends fault reports.</li> </ul> </li> </ul> </li> </ol> <p>UNTIL no more reports of problems</p> <ol style="list-style-type: none"> <li>4. Operator closes repair session.</li> </ol> |
| Variations     | #1. System may detect fault and notify operator or<br>Field maintenance engineer may report fault to Operator.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Non-Functional | <b>Performance Mean:</b> time to repair network fault must be less than 3 hours.<br><b>Fault Tolerance:</b> A repair session must be able to tolerate failure of Operator's console.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Issues         | What are the modes of communication between field maintenance engineer and operator?                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

126/42

27/42

|                           |                                                                                            |
|---------------------------|--------------------------------------------------------------------------------------------|
| <b>Use Case Extension</b> | Repair may fail extends 2. Repairing Cellular Networks                                     |
| <b>Description</b>        | Deals with assumption that network changes can never fail.                                 |
| <b>Steps</b>              | #3.3. if the changes to network fail then the network is rolled back to its previous state |
| <b>Issues</b>             | How are failures detected? Are roll backs automatic or is Operator intervention required?  |

28/42

- **Use Case (core)**
  - nome: suggerisce l'obiettivo del caso attraverso una forma transitiva attiva che cattura ciò che avviene quando il caso è invocato, in modo memorabile;
  - numero di riferimento: per essere referenziato in altri casi e per tracciare il requisito nel corso dello sviluppo, eventualmente strutturato in forma gerarchica x.y... per localizzare a un ambito o un sottosistema;
  - **UCD# Web.12 - Acquista biglietto (su trenitalia.com)**
- **History**
  - specifica chi ha creato/modificato il caso, quando, e perché.
  - **modified by      on                  changes                            version**

|    |          |               |     |
|----|----------|---------------|-----|
| EV | 27/06/10 | first created | 1.0 |
| EV | 27/06/10 | example added | 1.1 |
- **Source**
  - origine (source): identifica da dove è stato estratto il requisito.
  - **derivato da capitolo tecnico, sez. ..., integrato con interviste a ...**

- Level (core)
  - Livello di astrazione: tipicamente user-goal, essendo high-summary e summary troppo astratti e function troppo dettagliato per un template
  - **User goal**
- Description (core)
  - obiettivo (goal): descritto nella prospettiva del contesto applicativo, possibilmente in un unico periodo;
  - **Il cliente acquista un biglietto attraverso l'interfaccia web.**
- Scope
  - Organization, System, Component, release, incremento, ...
  - **Sottosistema web**
- Actors
  - attori coinvolti nel caso d'uso, eventualmente qualificati come primari e secondari, siano essi utenti o sottosistemi
  - **cliente (primary), sottosistema di gestione delle prenotazioni (secondary),**
  - ...

29/42

- Pre-conditions (core)
  - precondizioni necessarie a completare con successo il caso
  - o comunque necessarie per applicare la specifica senza fare riferimento a eventuali estensioni
  - **il cliente dispone di un account**
- Post-conditions (core)
  - condizioni al completamento del caso, con successo o fallimento dell'obiettivo
  - **viene emessa la prenotazione e caricata la carta di credito, viene inviato un messaggio al sistema di log, se richiesto l'utente riceve un sms o una email di conferma,**
  - ...

30/42

- Normal flow (core) (aka: common flow, steps, ...)
  - Trigger: azione che avvia il caso
  - Passi: sequenza di interazioni tra attori e sistema necessarie a raggiungere l'obiettivo del caso, strutturata in uno o più passi numerati e descritti in linguaggio naturale, con eventuale uso di alternative, ripetizioni e concorrenza
  - 0. il caso inizia quando
    - il cliente apre la pagina di visualizzazione dell'orario
  - 1. il cliente indica stazione di avvio e partenza, e un orario iniziale
  - 2. il sistema mostra 8 treni per volta su richiesta dell'utente
  - 3. il cliente seleziona un treno
  - 4. il sistema mostra le opzioni di viaggio
  - 5. il cliente compila le opzioni di viaggio
  - ...
    - 15. il cliente effettua il pagamento
    - ...
      - 21. il sistema invia una notifica di completamento

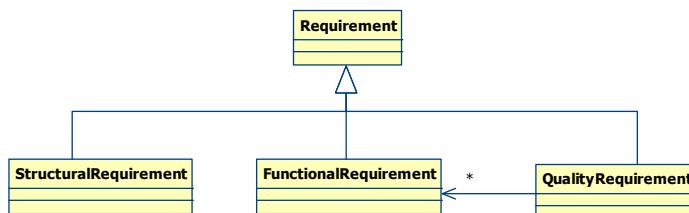
31/42

- Variazioni
  - Specifica, dove utile, varianti con cui possono essere eseguiti singoli passi
  - 15: il pagamento può essere effettuato con carta di credito o con carta postale
  - 21: la notifica può avvenire a mezzo sms o posta elettronica
- Alternative flows (core)
  - azioni che determinano diramazione rispetto alla sequenza comune
  - ciascuna seguita dalla descrizione della sequenza di passi che ne seguono
  - 15a: il cliente abbandona la transazione
  - 17a: il cliente non ricorda la password
  - ...
- Riferimenti
  - Casi superordinati o subordinati
  - include UCD# Web.8 - visualizza treni selezionati

32/42

■ Requisiti non funzionali

- requisiti architetturali o di qualità, classificati rispetto alla natura
- **Performance:** 100 transazioni al minuto con tempi di attesa <= 5 sec;  
**Accessibilità:** interfaccia AA-compliant;  
**Architettura:** riconfigurabile con fogli di stile;  
...  
▪ :-( Non sempre partizionabili sui singoli casi d'uso



133/42

■ Issues

- elenco di aspetti che devono ancora essere chiariti, note su possibili strategie di implementazione o sull'impatto verso altri casi, assunzioni fatte nello specificare il caso
- **TBD:** il cliente può applicare il procedimento a treni regionali?

■ Priorità

- criticità rispetto al piano di sviluppo

■ Data di consegna

- Data o incremento in cui è previsto (o avvenuto) il rilascio

134/42

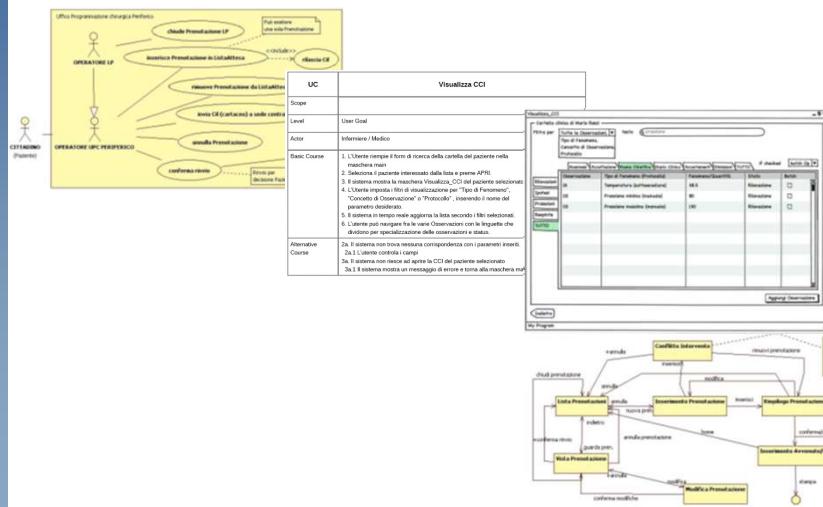
## Use cases - a cosa servono

- Non servono a
  - Descrivere procedure nel flusso di elaborazione dell'informazione
  - partizionare il processo di elaborazione all'interno del sistema
  - Progettare
- Servono a
  - discutere e definire requisiti funzionali
  - definire le interfacce di un (sotto)sistema
  - partizionare e strutturare le funzionalità (breakdown)
- E poi anche a
  - pianificare e monitorare lo sviluppo
  - pianificare test in prospettiva funzionale (acceptance testing)
  - sostenere un piano di usability engineering
  - specificare condizioni di workload
- Sono facili da mantenere
- Relazione con le classi del modello concettuale
  - Nel testo degli use case templates

135/42

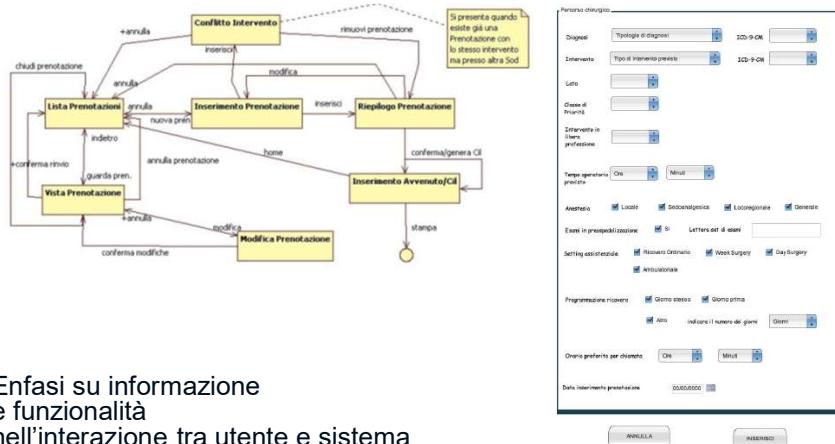
## 3/3: Mock-ups

- mock-ups
  - prototipo dell'interfaccia grafica nei passi di uno use case template
  - accompagnati da un page navigation diagram



136/42

- Prototipo delle interfacce grafiche esposte all'utente
  - Page Navigation Diagram: struttura di navigazione delle pagine
  - Layout delle singole pagine
  - E.g. lista,inserimento,confitto,riepilogo,avvenuto



- Enfasi su informazione e funzionalità nell'interazione tra utente e sistema
  - Astrazione rispetto ai dettagli del design grafico

37/42

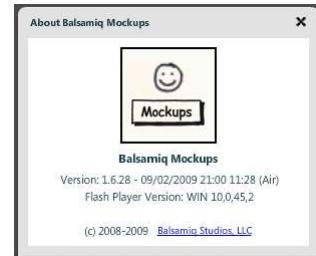
### Esempio: visualizza Cartella Clinica Informatizzata

The screenshot shows a window titled "Visualizza\_CCI" displaying a medical chart for "Cartella clinica di Mario Rossi". The chart lists observations (Osservazioni) such as Temperature (sottoascellare), Minimum pressure (manuale), and Maximum pressure (manuale). The interface includes a sidebar for filtering by observation type (Rilevazioni, Ipotesi, Proiezioni, Resposte, TUTTO) and a search bar for "pressione". A toolbar at the top includes buttons for Anamnesi, Accettazione, Esame Obiettivo, Diario Clinico, Accertamenti, Dimissioni, and TUTTO. A "batch Op" dropdown is also present. At the bottom, there are buttons for "Indietro" (Back), "My Program", and "Aggiungi Osservazione" (Add Observation).

38/42

## Motivazioni per l'uso di Mock-ups

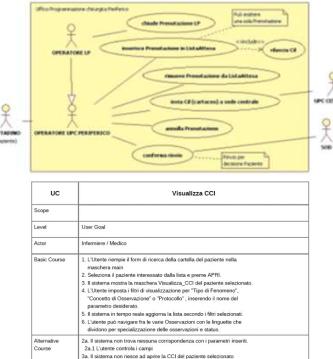
- Favoriscono il coinvolgimento di utenti e esperti di dominio
  - Concretezza della rappresentazione
  - Basso costo di evoluzione
  - Stimola critica e discussione
- Diverse finalizzazioni nelle fasi del ciclo di vita
  - progetto: Early prototype (low-fidelity), Valutazione di idee progettuali alternative, usability engineering, ...
  - analisi: validazione dei requisiti (high fidelity)
  - Test di accettazione: supporto alla verifica di completezza
- Supportato da strumenti



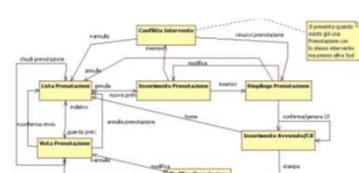
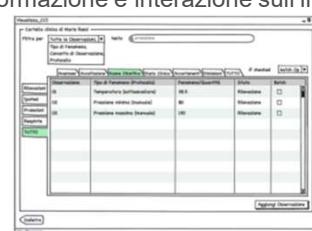
39/42

## Sintesi: rappresentazione composta

- Use case diagrams
  - visione di insieme,
  - relazioni tra attori e casi d'uso,
  - strutturazione dei casi d'uso
- Use case templates
  - Specifica testuale di singoli casi d'uso
- Mock-ups e page navigation diagram
  - Informazione e interazione sull'interfaccia grafica destinata agli utenti



| UC                 | Visualizza CCI                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Scope              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Level              | User Goal                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Actor              | Intervenire / Medico                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Basic Course       | <ol style="list-style-type: none"><li>L'utente compie il tasto di ricerca della cartella del paziente nella tastiera mani.</li><li>L'utente inserisce il numero della cartella e premi APP.</li><li>L'utente mostra la macchina Visualizza_CCI del paziente selezionato.</li><li>L'utente impone i filtri di visualizzazione per "Tipo di Intervento": "Cognitivo".</li><li>L'utente inserisce il codice del paziente nel campo del paziente desiderato.</li><li>L'utente può scegliere la linea seconda filtri selezionati.</li><li>L'utente può scegliere la linea seconda filtri selezionati.</li></ol> |
| Alternative Course | <ol style="list-style-type: none"><li>2x L'utente non trova nessuna corrispondenza con i parametri inseriti.</li><li>2x L'utente controlla i campi.</li><li>2x L'utente inserisce un codice in CCI del paziente selezionato.</li><li>2x L'utente mostra un messaggio di errore e torna alla macchina mani.</li></ol>                                                                                                                                                                                                                                                                                       |



40/42

- Alistair Cockburn, "Writing effective use cases," Addison Wesley, Pearson Education, 2001.
- Martin Fowler, "UML Distilled: Guida rapida al linguaggio di modellazione standard" - terza edizione, Pearson Education Italia, Febbraio 2004 .
- Martin Fowler, "UML Distilled: a brief guide to the standard object modeling language", third edition (Addison Wesley).
- Martin Fowler, "Analysis Patterns, reusable object models", Addison Wesley 1997.
- J.Arlow, I.Neustadt, "UML e Unified Process," McGraw Hill, 2003.

## Software Engineering - A.A. 20/21

### **Il ciclo di vita dei sistemi SW modelli e concetti per lo sviluppo di SW intensive systems**



**Enrico Vicario**  
Dipartimento di Ingegneria dell'Informazione  
Laboratorio Tecnologie del Software – stlab.dinfo.unifi.it  
Università di Firenze

[enrico.vicario@unifi.it](mailto:enrico.vicario@unifi.it), [stlab.dinfo.unifi.it/vicario](http://stlab.dinfo.unifi.it/vicario)

## ■ Software intensive system

- "any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole"  
[IEEE-Std-1471: Recommended Practice for Architecture Description of Software-Intensive Systems ]

## ■ Applications vs Systems

- much about the integration of SW components with or within hardware, electronic or electro-mechanical units, ...

*Oedipa: "Sorting isn't work?"*

*Koteks: "It's mental work,  
but not work in the thermodynamic sense."  
(T.Pynchon)*

- E.g.: una cartella clinica, l'applicazione di gestione degli orari, ...
- E.g.: lo scheduler di un sistema elettromeccanico per immunodiagnistica, un Centralized Traffic Control System ferroviario, ...

## ■ Prodotto

- un artefatto costruito nel corso dello sviluppo
- il programma eseguibile, il codice sorgente, ...
- un modello, la documentazione, un manuale, una suite di test, ...

## ■ Processo

- un insieme organizzato di attività che conducono alla creazione di un prodotto
- ... ha a che fare con come si fanno le cose, più che con le cose

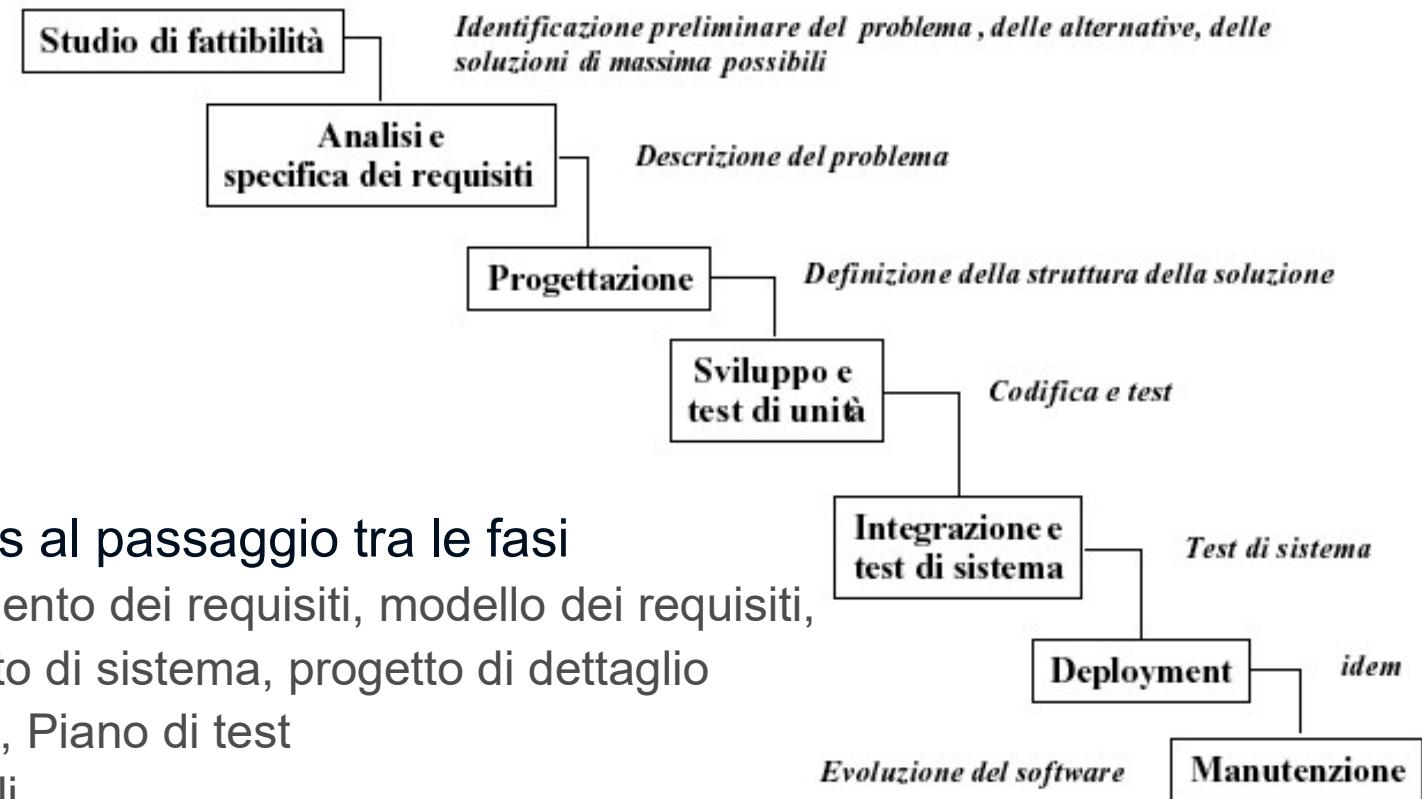
## ■ Software Development Lifecycle (SDLC)

- the process through which a software system is developed, acquired, configured, maintained, ...

- Lifecycle model
  - un modello per il processo di sviluppo
  - fasi, attività e regole di transizione
  - artefatti intermedi e loro relazioni
  - metodi, tecniche, strumenti
  - ruoli
- Waterfall (1970)
  - Vmodel (1991)
  - Spiral (1988)
- Unified Process - UP (1999)
  - Mills (1980)
- eXtreme Programming - XP (1999)

- E' il primo modello di ciclo di vita dopo la crisi del SW
- identifica attività specializzate, separando competenze e ruoli
  - Studio di fattibilità: consenso, convenienza, capacità di assorbire l'investimento, packaging/sviluppo, in-house/outsourcing,
  - Analisi: documento dei requisiti (SRS), modello dei requisiti
  - Progetto: architettura di deployment, architettura software, progetto di dettaglio
  - Implementazione
  - Testing: di unità o di integrazione, di accettazione
  - Deployment: preparazione dei siti, formazione, installazione
  - Manutenzione: errori, adattamento all'ambiente, evoluzione delle funzionalità
- Le attività sono organizzate in modo sequenziale
  - ... con limitato overlap e backtrack

- Schema sequenziale con limitato overlap e backtrack



- Milestones al passaggio tra le fasi

- Documento dei requisiti, modello dei requisiti,
- Progetto di sistema, progetto di dettaglio
- Codice, Piano di test
- Manuali

- Abilita la specializzazione dei compiti

- consolidata attraverso artefatti intermedi
- ... anche troppo

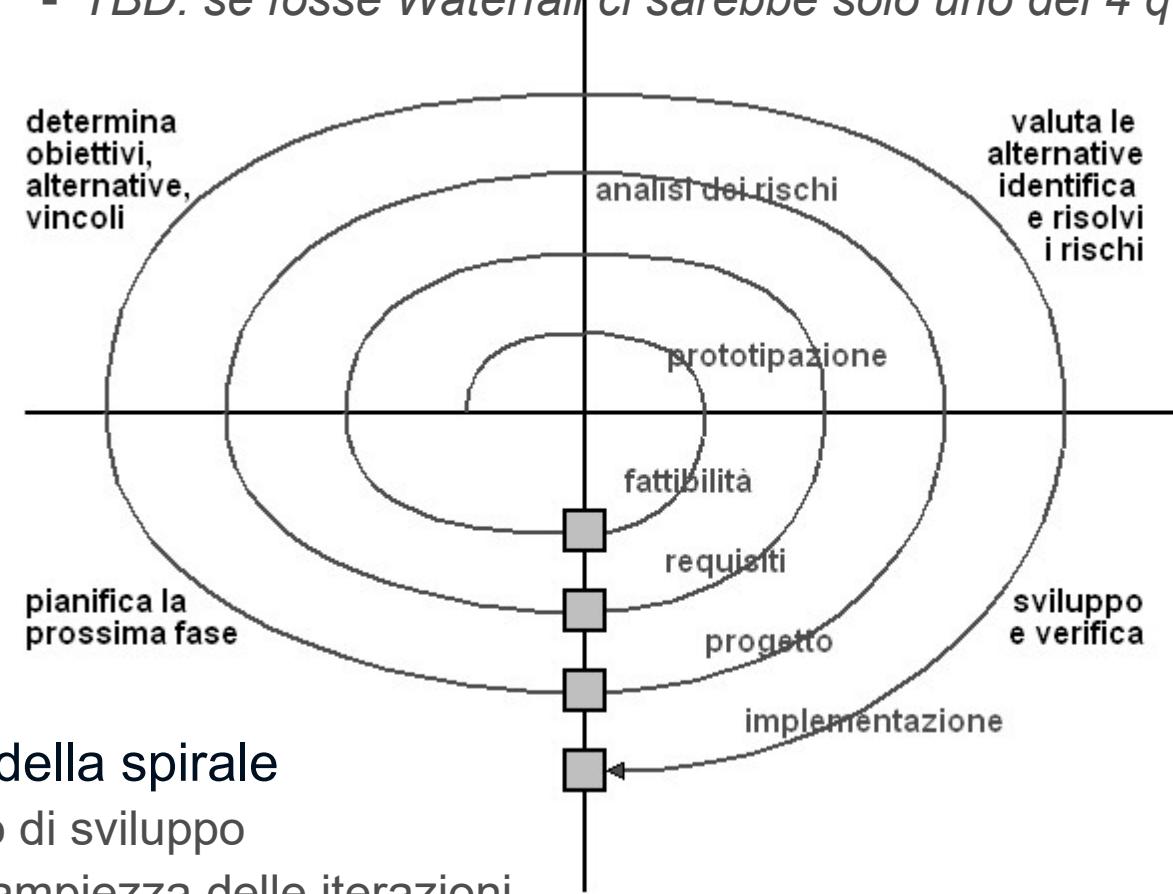
- Merito storico di identificare attività specializzate
- Tuttora ampiamente in uso
- E' il modello che piace al corporate management
  - Semplifica le attività di Contract e Management Views (IEEE 12207)
  - Si presta bene a prevedere una successione di artefatti che documentano l'avanzamento dello sviluppo
- Ma non al development team
  - Non rappresenta bisogni della Engineering View (IEEE 12207)
  - Congelamento dei requisiti e delle scelte del progetto
  - Innaturale completamento delle fasi
  - Up-front
- E' scarsamente flessibile rispetto alla evoluzione dei requisiti e alla comprensione incrementale del progetto

## ■ ISO/IEC/IEEE 12207:2017

### “Systems and software engineering — Software life cycle processes”

- a framework for software life cycle processes, with well defined terminology, identifying processes, activities, and tasks to be applied during acquisition of a software system, product or service and during supply, development, operation, maintenance and disposal of software products and the software portion of any system, whether performed internally or externally to an organization.
- may be applied in conjunction with ISO/IEC/IEEE 15288:2015, “Systems and software engineering—*System* life cycle processes”, depending on the emphasis on *software* or *systems* engineering, respectively
- divides software life cycle processes into four main process groups
  - agreement (agreement btw supplier and acquirer, actions for initiating a project, development of a project management plan, ...)
  - organizational project-enabling (infrastructure, human resources, knowledge management, ...)
  - technical management (... planning, assessment, and control during the life cycle, ensuring quality along the way)
  - technical processes (all development and technical activities during operation, maintenance, and disposal)

- Ogni iterazione percorre regioni di attività comuni
  - Produce un prodotto di maggiore livello
  - Documento dei requisiti, modello di progetto, codice, ...
  - È una evoluzione del waterfall che include la gestione del processo
    - *TBD: se fosse Waterfall ci sarebbe solo uno dei 4 quadranti*



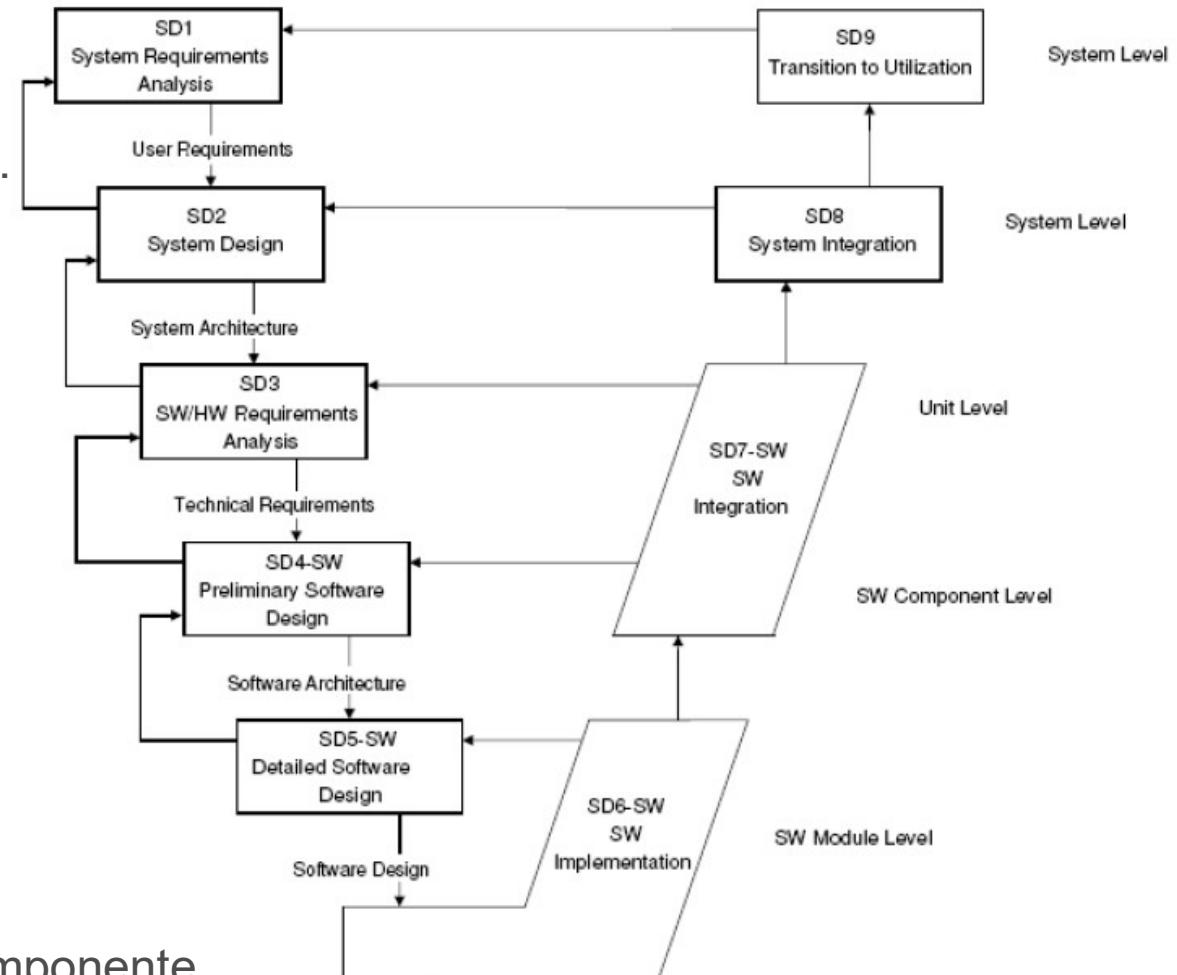
## Risk driven

- Prototyping

## Interpretazione della spirale

- costo e tempo di sviluppo
- Frequenza e ampiezza delle iterazioni
- *TBD: sull'asse degli artefatti, ci potrebbero essere gli artefatti Mil-std-498*

- per sistemi industriali
  - difesa, avionica, ferrovie, automotive, dispositivi biomedici, ...
  - nato come standard federale tedesco
- E' una riorganizzazione del modello waterfall
- due dimensioni
  - in verticale: sistema/componente
  - In orizzontale: sviluppo/validazione
- feedback
  - tra livelli corrispondenti anziché successivi



### ■ Verification

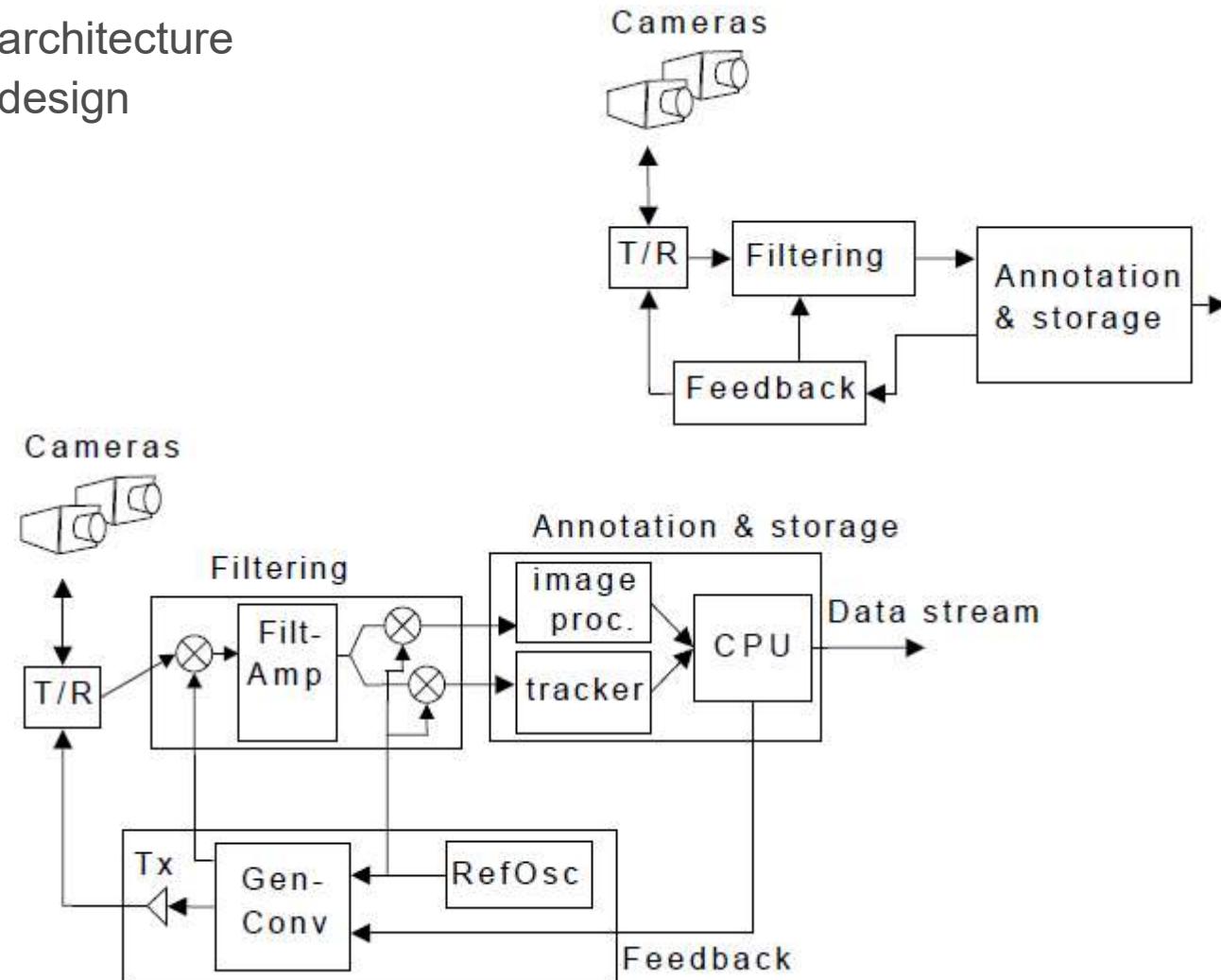
- ensuring that the software has been built correctly.
- “Are we building the product right ?“

### ■ Validation

- ensuring that the software meets the requirements, both the stated and implied.
- “Are we building the right product ?“

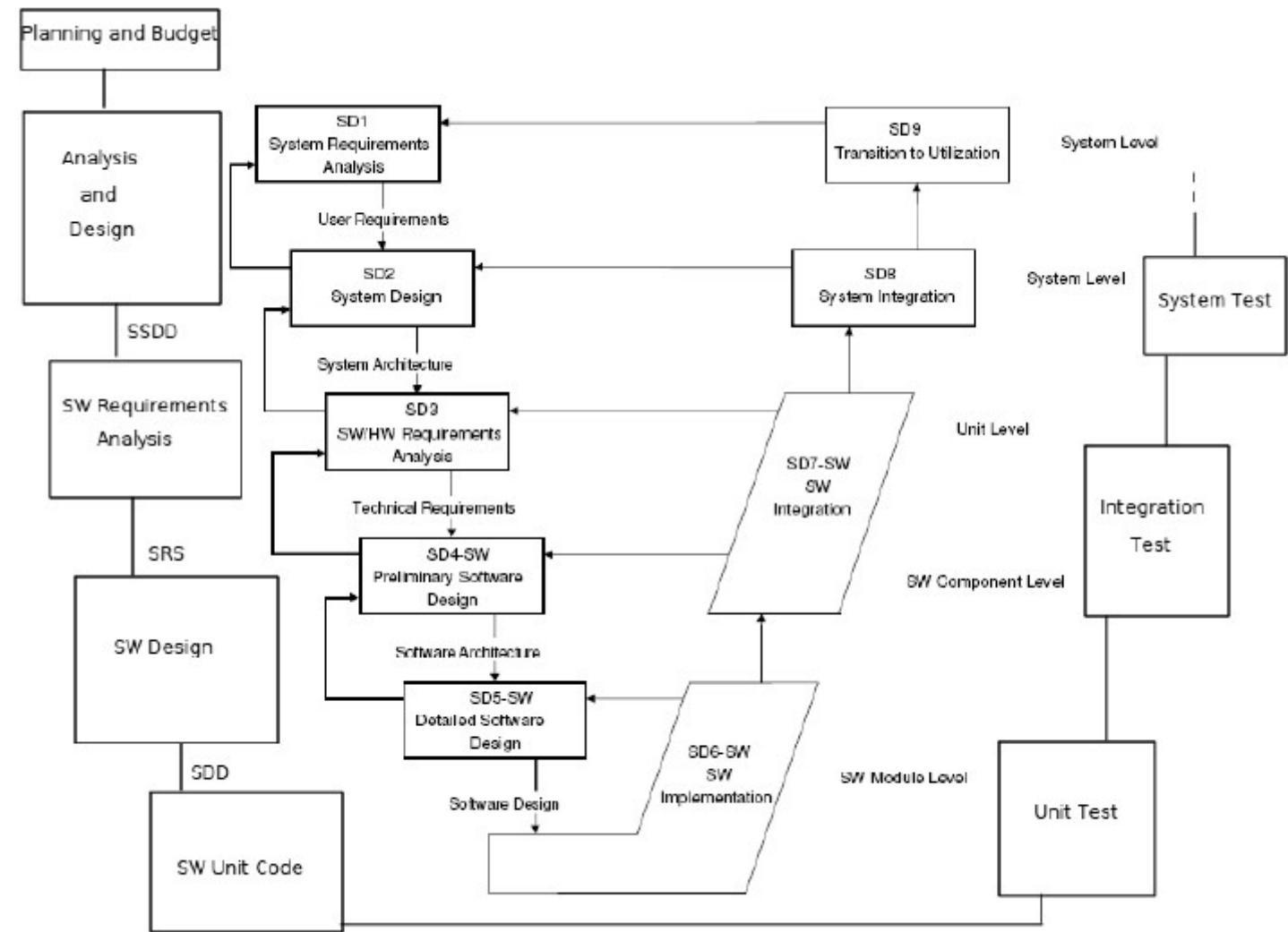
## ■ Enfasi su sistema e sottosistemi

- System architecture
- System design



## ■ HW e SW Configuration Items

- Tailoring
- Association with Mil-std-498 documental process



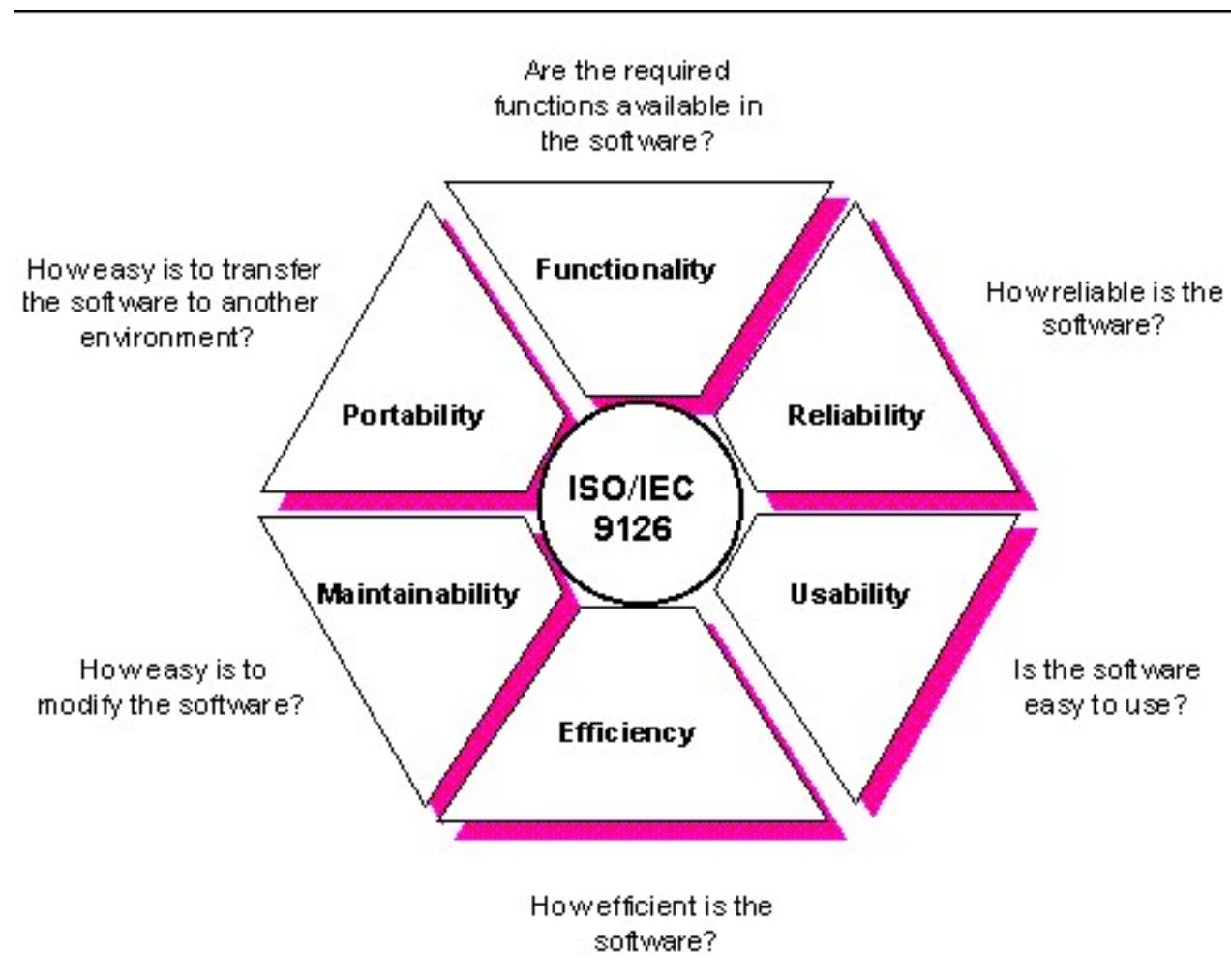
- **MIL-STD-498**
  - A military standard of the Department Of Defense (DOD) aiming at uniform requirements for software development and documentation
  - 1994, for temporary use, then embodied in other standards and practices
  - identifies 22 Data Item Descriptions which basically track the lifecycle
- **Planning**
- **Requirements**
  - SRS: SW Requirements Specification
  - IRS: Interface Requirements Specification
- **Design**
  - SSDD: System Subsystem Design Description
  - SDD: SW Design Description
- **Qualification/Test**
  - STP: SW Test Plan
  - STD: SW test description
  - STR: SW Test Report
- **User manuals**
- **Support Manuals**
- **Software**

*TBD: notes on the SRS –  
refer to the annotated document srs\_template.doc*

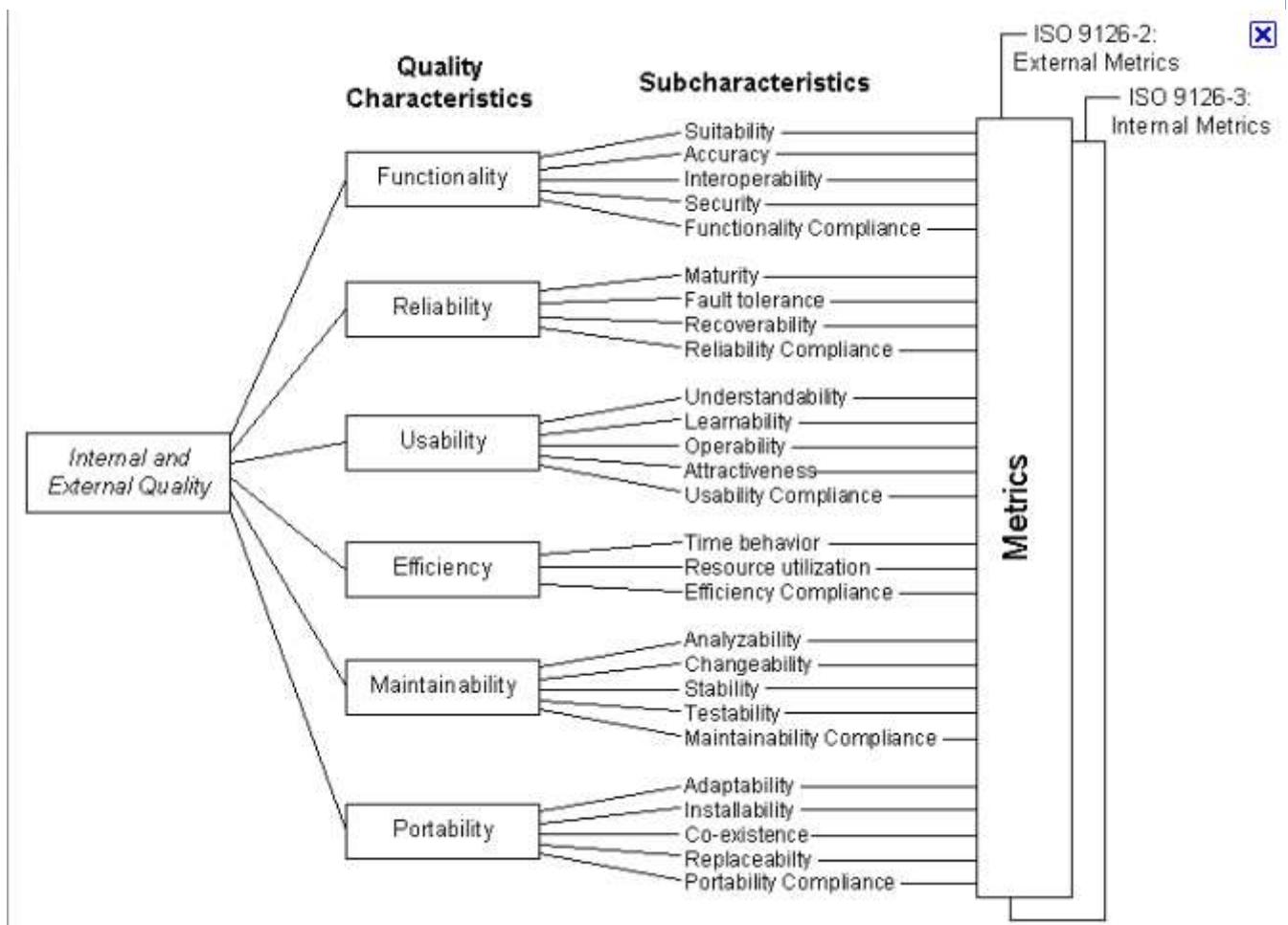
- Requisiti di qualità: attributi del modo con cui sono offerte le funzioni

## ISO9126

- the first model of quality of Software (recently evolved to ISO:25000)
- identifies 6 characteristics



- Requisiti di qualità: attributi del modo con cui sono offerte le funzioni
- ISO9126: qualità del SW scomposta in 6 caratteristiche
  - ... e 27 sottocaratteristiche



... con intento di

- completezza
- e indipendenza  
(no overlap)

- ISO/IEC 25010 “System and software quality models”
  - belongs to ISO25000 SQuaRe: Systems and software Quality Requirements
  - ... an evolution of ISO 9126 following occurred changes
- defines quality in terms of 8 characteristics



- <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>

- **Functional suitability:** degree a product or system provides functions that meet stated and implied needs when used under specified conditions
- **Performance efficiency:** amount of resources used under stated conditions
- **Compatibility:** degree a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment
- **Usability:** degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. (see also ISO 9241)
- **Reliability:** degree to which a system, product or component performs specified functions under specified conditions for a specified period of time
- **Security:** degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
- **Maintainability:** degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements
- **Portability:** Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.



- distinguishes quality of the product
  - Qualità interna, relativa a proprietà statiche sul codice software verificabili con analizzatori o ispezioni;
  - Qualità esterna, verificabile da analizzatori e da tecnici con test dinamici in ambienti simulati;
- ... and quality in use
  - Qualità in uso, verificabile in ambienti reali o simulati con la partecipazione di utenti che enfatizzano le difficoltà o facilità di interazione uomo-computer.

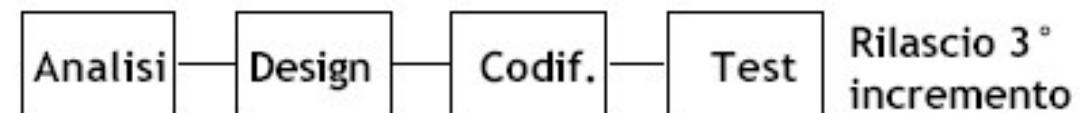
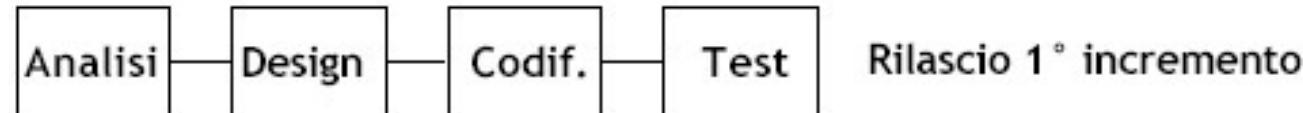
## ■ discussion

- about Reliability and Maintainability, Dependability, and RAM-S
- about further evolution to encompass Machine Learning components
- ...



- Break the sequential structure of development
  - ... while maintaining continuity in the process
- Some preliminary main concepts
  - increment and iteration
  - software architecture
  - executable software architecture

- First embodied in the Mills lifecycle model (1980)
- Sovrapposizione di specifica e sviluppo
  - Parallelizzazione di attività diverse relative a diversi incrementi
  - Rilascio per incrementi successivi ciascuno in sequenza waterfall e ciascuno operativo



- Vantaggi e limiti
  - Definizione incrementale dei requisiti
  - Utenti coinvolti per feedback e testing
  - Consolidamento del testing sui primi incrementi
  - Precedenza ai componenti più critici
  - Difficoltà contrattuale  
(è la rivincita del team di sviluppo sul corporate management)

### ■ Software architecture

- assumptions about how a system will be decomposed,  
what will be the interaction mechanisms,  
the protocols for sharing resources,  
how data will be persisted and exchanged,  
interaction between data representation and presentation,  
... TBD

- "Software architecture encompasses the set of significant decisions about the organization of a software system including
  - the selection of the structural elements and their interfaces by which the system is composed;
  - behavior as specified in collaboration among those elements;
  - composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization.

### Software architecture also involves

- functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns."

[P.Kruchten, G.Booch, K.Bittner, R.Reitman], [M.Shaw,D.Garlan 1996]

### common recurring themes when explaining architecture:

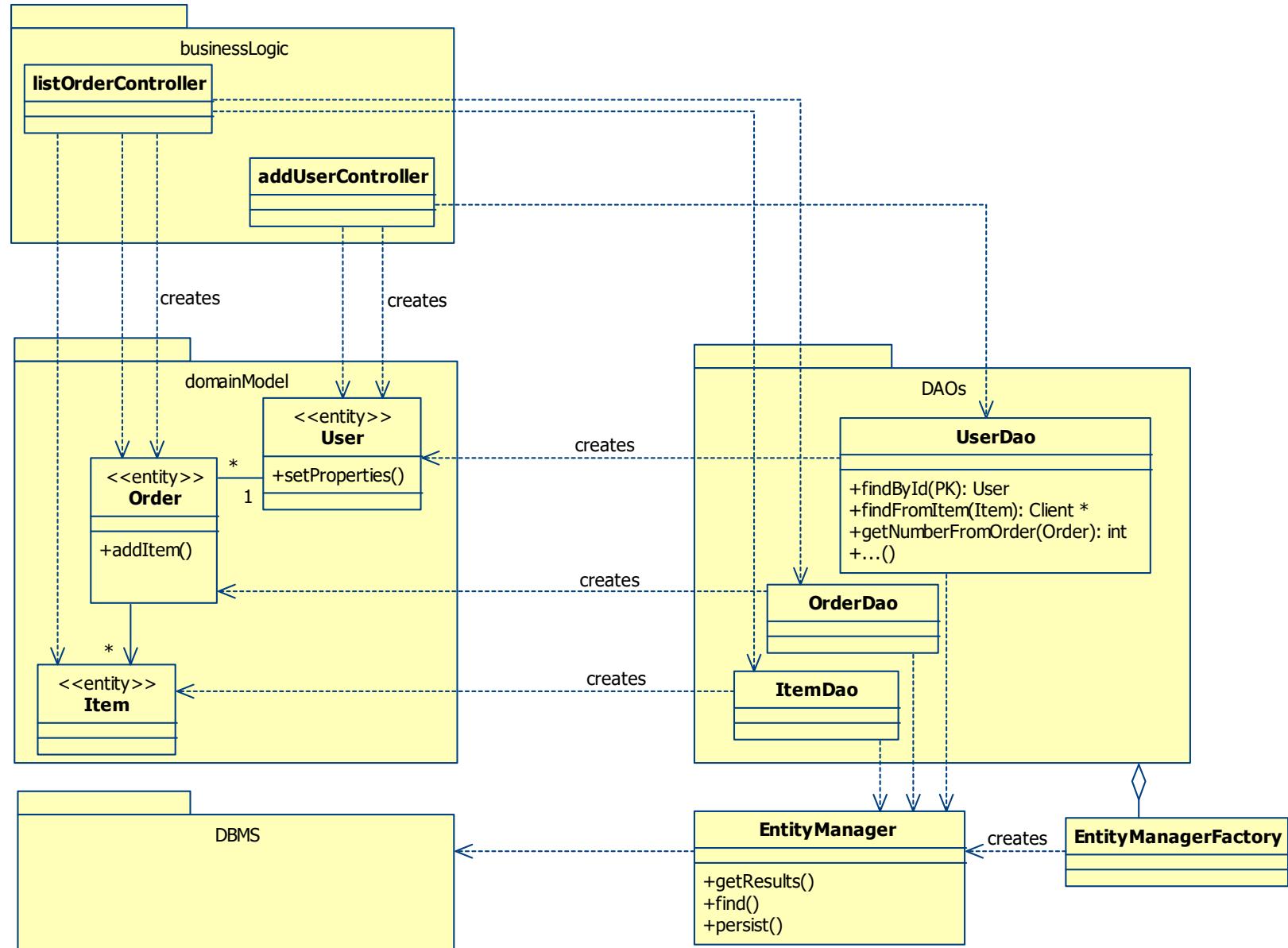
- "The highest-level breakdown of a system into its parts;  
the decisions that are hard to change;  
there are multiple architectures in a system;  
what is architecturally significant can change over a system's lifetime;  
and, in the end, architecture boils down to whatever the important stuff is."

[Martin Fowler, Patterns of Enterprise Application Architecture]

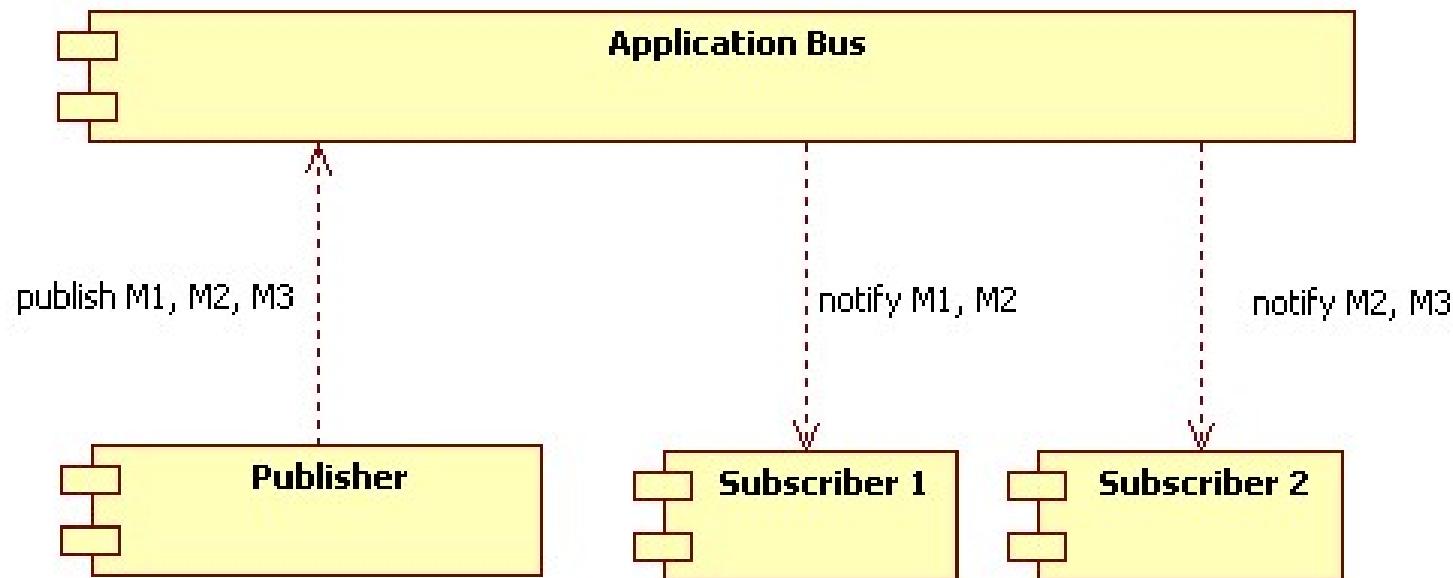
### Executable software architecture

- an executable implementation of the architectural concepts
  - It is made of code, not only of documents, and can be executed
  - ... but, it is not yet charged of actual functional capabilities
  - It provides a kind of backbone,  
where functional capabilities can be added
  - ... might be a reference implementation with fake functional capabilities
- 
- *TBD (obsoleted): per introdurre il concetto di executable architecture, viene bene di fare riferimento a un sistema di gestione della mobilità: ci sono InformationProviders, PredictionEngines e DecisionSupportSystes, integrati su un ESB, in modo simile a come funziona un Observer.*
    - *Ho un'architettura eseguibile il giorno che ho dispiegato un ESB e ho implementato ping-pong-pang*

## ... a concrete example: J2EE architecture



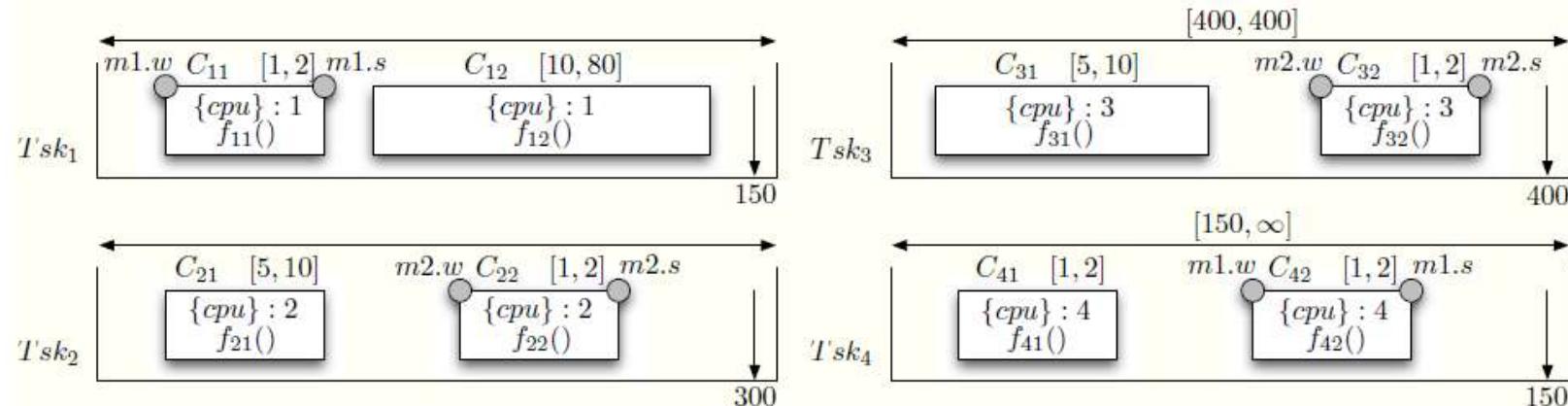
- Un sistema di cooperazione publish & subscribe
  - Also Known As: broker, event channel
  - Equivalente architetturale del pattern observer
- Participants:
  - Publisher: pubblica un evento su un topic sul broker
  - Subscriber: sottoscrive un topic sul broker, accetta notifiche dal broker
  - Broker: accetta eventi e li inoltra a tutti sottoscrittori del topic



# SKIP: Concept of executable architecture: another example

## Architettura di un real time task-set

- Un set di tasks concorrenti
- Ciascun task rilascia jobs, composti da chunks
- priority driven preemptive scheduling
- chunks sincronizzati su risorse (semafori)



## I chunks realizzano funzioni

- Ciascuno implementato da una funzione c (entrypoint)
- ... che può chiamare altre funzioni, ma non modifica l'uso delle risorse

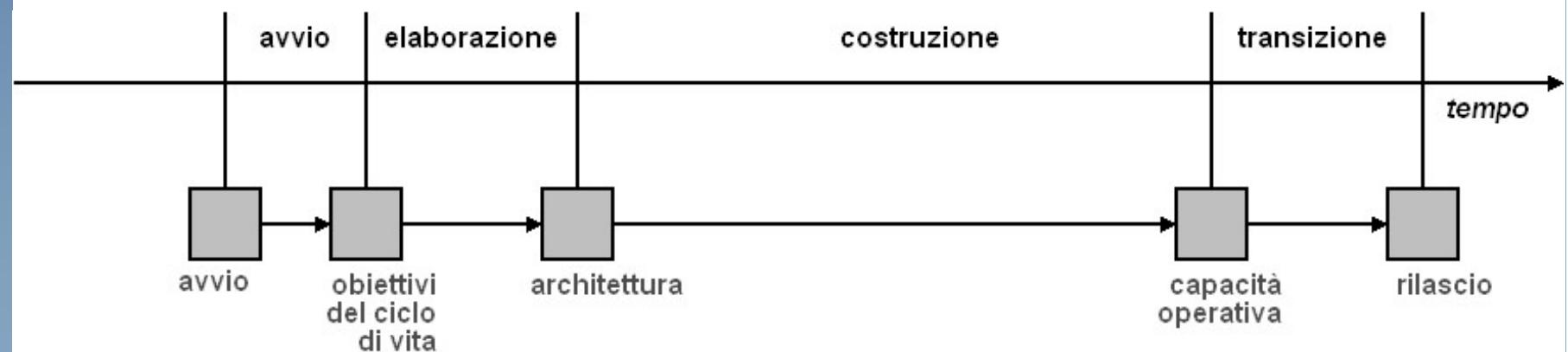
## Executable architecture

- I chunks sono emulati da una funzione busy-sleep
- E' implementata l'architettura ma manca completamente la funzionalità

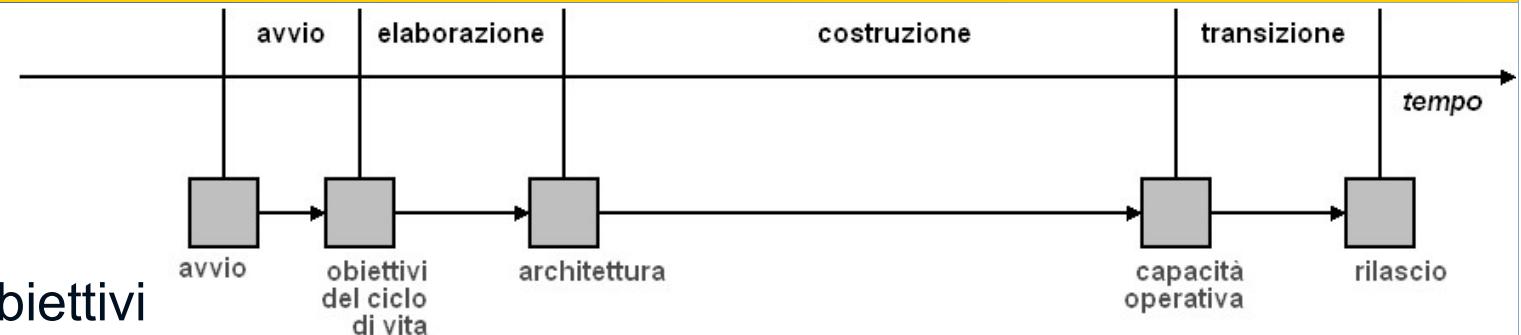
- *TBD: ... when here, you may want to skip UP and present first XP*

- Imperniato attorno ad un approccio OO
  - Fortemente basato su UML e sui tools che lo supportano
- Assiomi
  - Procede in maniera iterativa con incrementi operativi
  - Incentrato su una *architettura* fissata dal primo incremento
  - Pilotato da *casi d'uso* e dai fattori di rischio

- Il ciclo di vita è articolato in 4 fasi
- ... e ciascuna fase ha un milestone
  - Avvio (inception) -> obiettivi
  - Elaborazione -> architettura eseguibile (e 80% dei casi d'uso)
  - Costruzione -> capacità operativa
  - Transizione -> rilascio



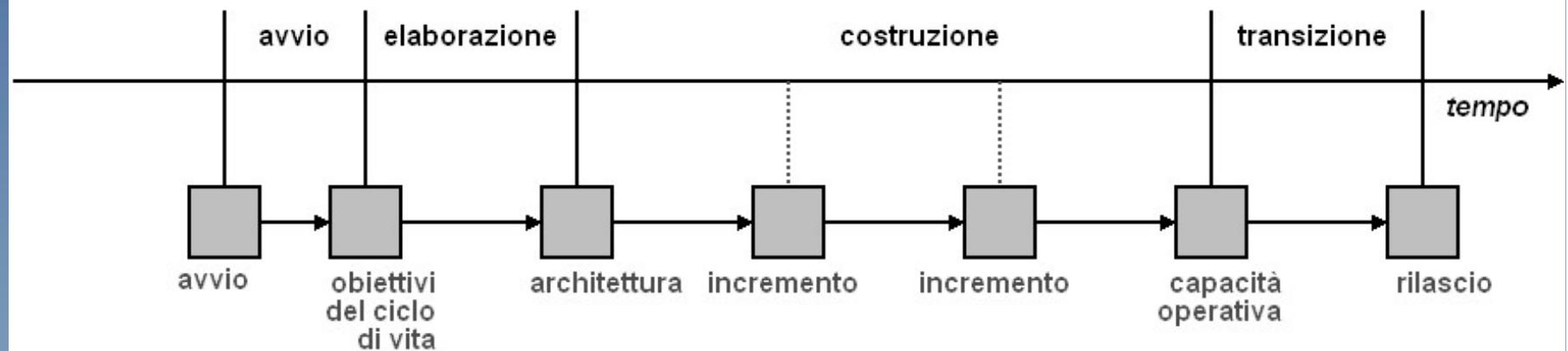
*TBD: Tradurre tutto in Inglese  
(e.g. avvio al posto di inception è insopportabile)*



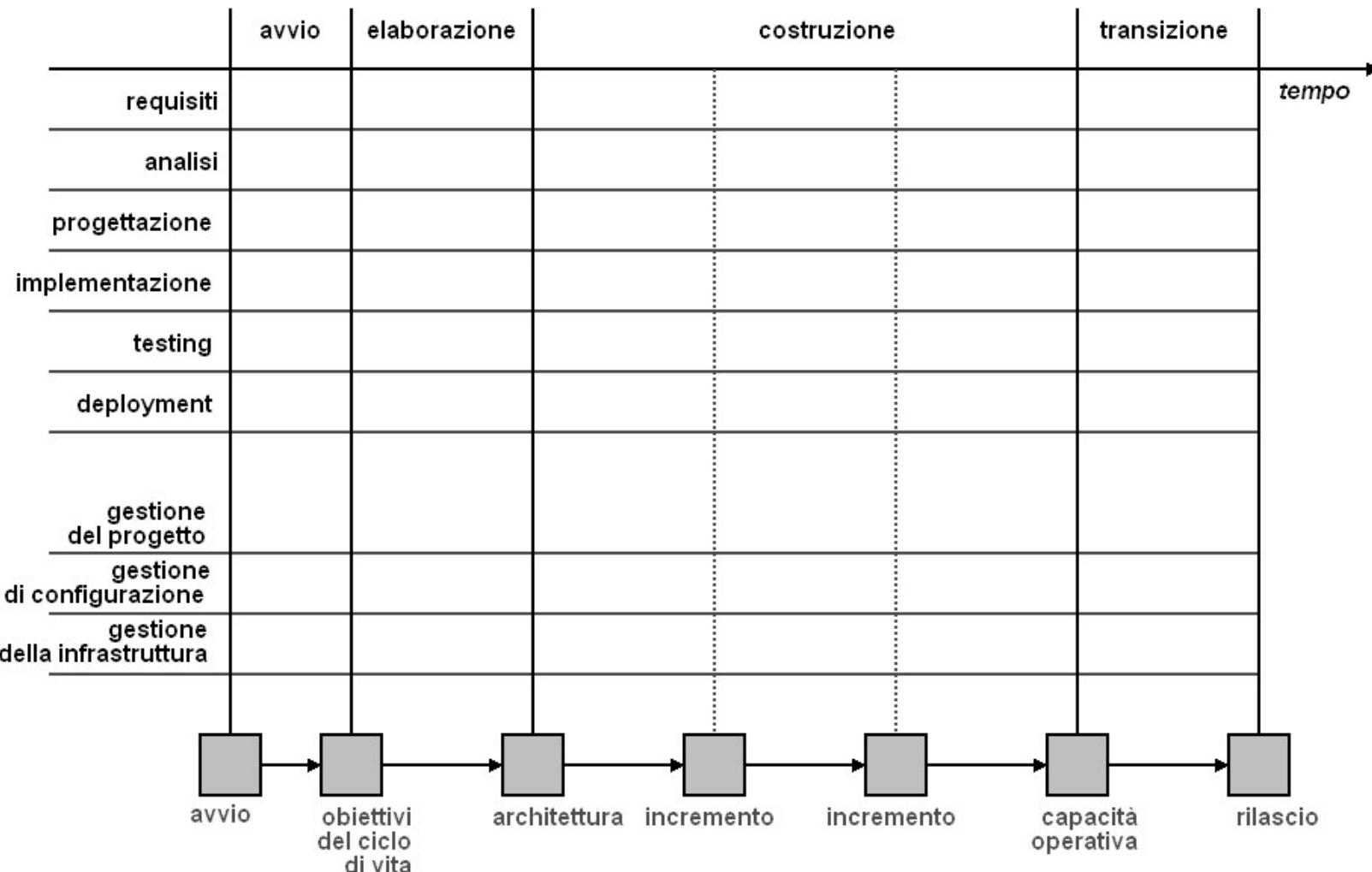
- Avvio -> obiettivi
  - Obiettivi e delimitazione del progetto
  - Fattibilità, requisiti essenziali, rischi maggiori, consenso manageriale
- Elaborazione -> architettura
  - Definizione dell'80% dei casi d'uso
  - Baseline eseguibile: architettura di base
  - Pianificazione della costruzione, valutazione delle risorse necessarie, perfezionamento analisi dei rischi, definizione di attributi di qualità
- Costruzione -> capacità operativa
  - Completamento dei requisiti e del modello dell'analisi
  - Evoluzione della baseline in coerenza con la architettura iniziale
  - Capacità operativa iniziale: beta-test
- Transizione -> rilascio
  - Ambiente di produzione, test, e manutenzione; infrastruttura nei siti utente; help desk I e II livello, trouble ticketing, metriche di qualità, manuali, preparazione, consulenza, (conduzione)...
  - Correzione difetti , adattamento, Debriefing      *TBD: often underestimated*

- Ciascuna fase articolata in iterazioni

- Ciascuna iterazione organizzata come un sotto-progetto
- durata di 1-2-3 mesi per iterazione
- tipicamente più iterazioni nella fase di costruzione



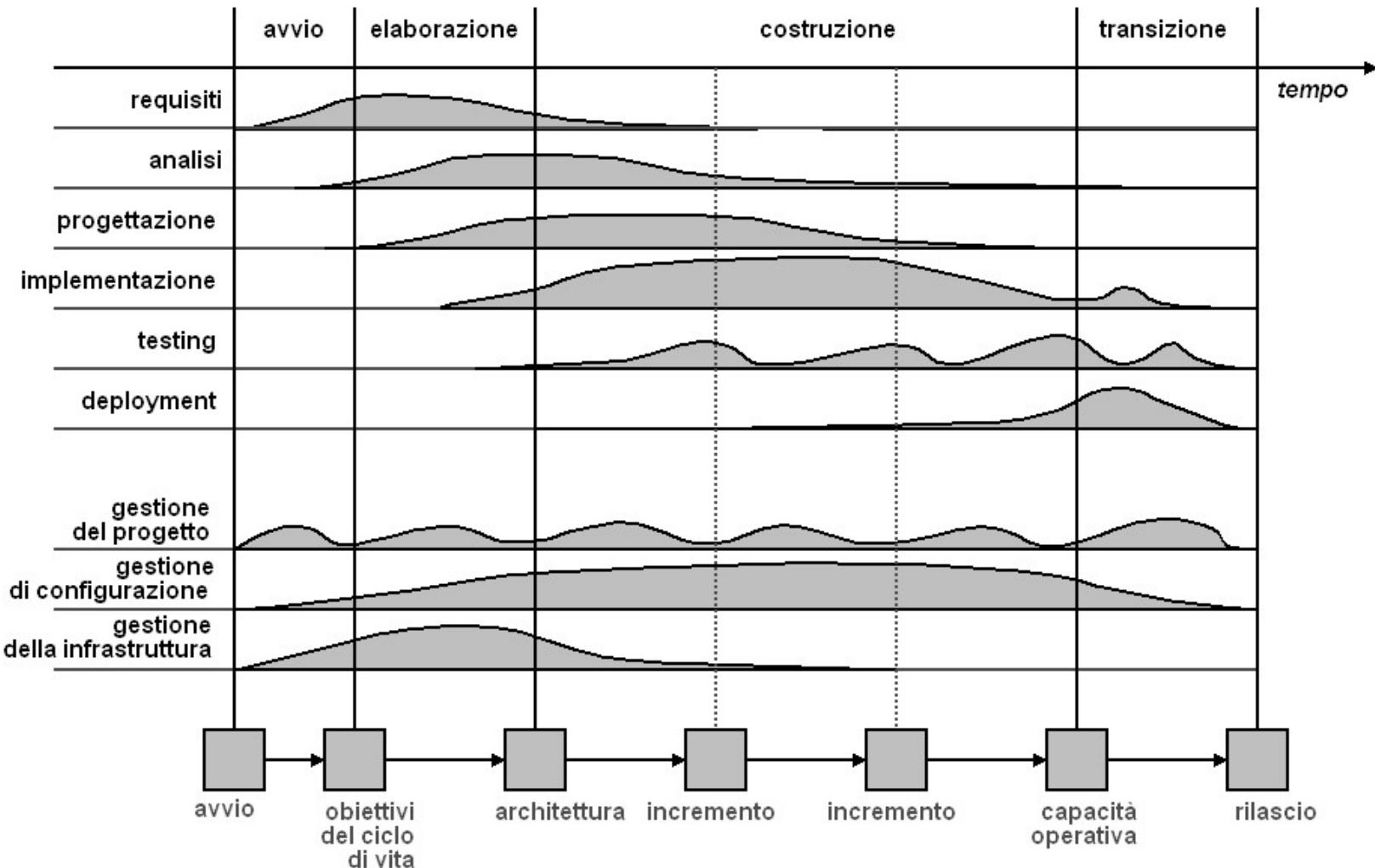
- Il processo avanza con attività che attraversano fasi e iterazioni
  - Requisiti, Analisi, Progettazione, Implementazione, Test, Rilascio
  - Gestione di progetto, configurazione, infrastruttura di sviluppo



- Ciascun flusso di lavoro produce degli artefatti
  - Requisiti
    - documento dei requisiti (Sw Requirements Specification)
    - modellazione dei requisiti funzionali (use-case diagrams)
    - mappatura tra casi di uso e requisiti della SRS
    - requisiti non funzionali
  - Analisi
    - classi di analisi e loro relazioni (modello concettuale)
    - realizzazione dei casi di uso sulle classi di analisi (interaction diags., robustness diag.)
  - Progettazione
    - relazione tra documenti di analisi e progetto: round-trip, tracciabilità
    - team verticale: smoothing della separazione tra analisi e progetto
    - identificazione dei sottosistemi
    - classi del modello di specifica, interfacce
    - realizzazione dei casi di uso sulle classi di progetto (interaction diags.)
    - modello di deployment
  - Implementazione
    - Modello di implementazione (Design patterns)
  - Testing e Deployment

I diversi flussi hanno diverso peso nelle diverse fasi

- nelle prime iterazioni prevalgono i requisiti, nelle ultime il test



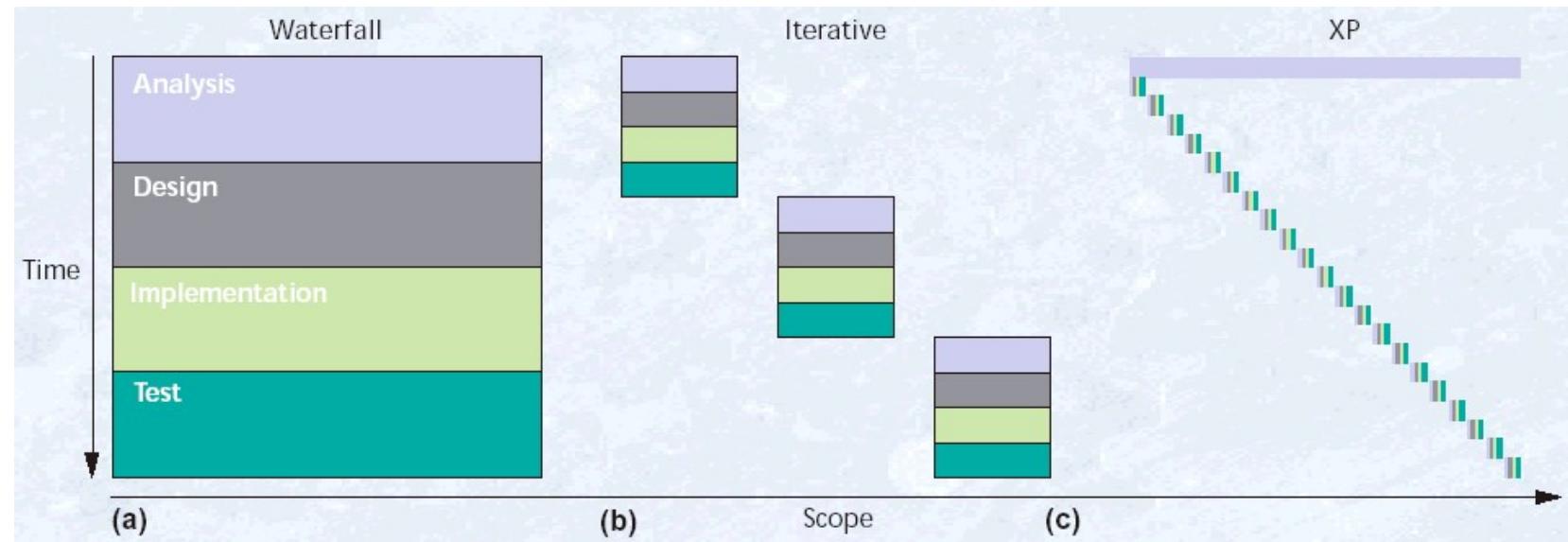
- Istantiato in ciascun progetto
  - pianificazione specifica di artefatti intermedi, ruoli, strumenti di sviluppo, disciplina di archiviazione, attività di gestione, ...
  - in modo più o meno ceremonioso
  - recupera il concetto del tailoring del VModel
- UP metaprocesso base di Rational-UP
  - RUP è comunque un meta-processo, di più semplice istanziazione

- XP comes as a further step from discipline towards agility
- Waterfall (1970)
  - Vmodel (1991)
  - Spiral (1988)
- Unified Process - UP (1999)
  - Mills (1980)
- eXtreme Programming - XP (1999)

- Processo agile, adatto al cambiamento
  - Molto più adatto ai bisogni del team di sviluppo che non al management
- Principi
  - Rapid feedback
  - Assume simplicity
  - Incremental change
  - Embrace change
  - Quality work
- Valori
  - Communication
  - Simplicity
  - Feedback
  - Courage

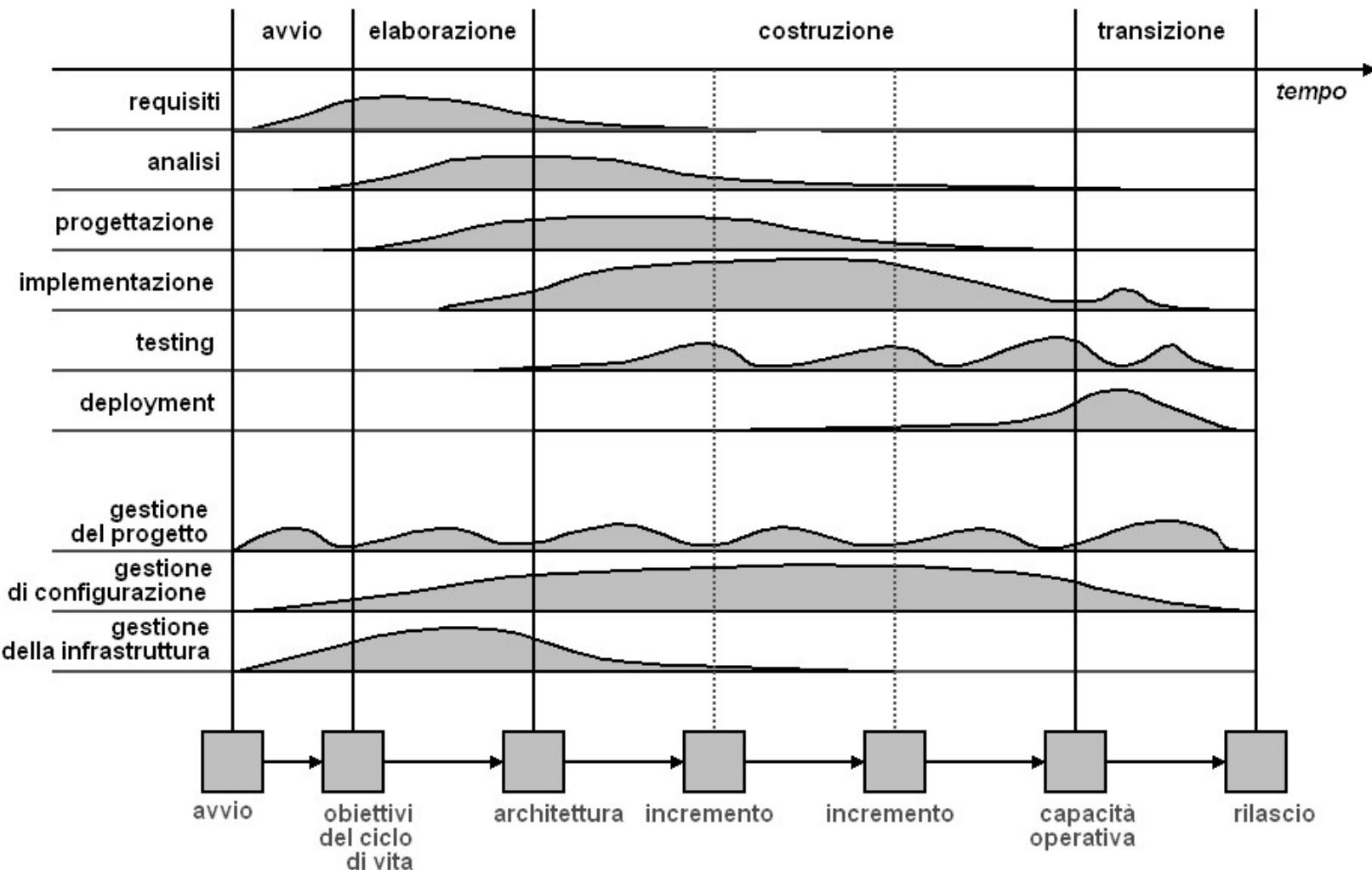
- Evitare di prevedere
  - overdesign, funzionalità (ad oggi) inutili, chiusure strategiche, *up-front*
- Evitare la separazione dei ruoli
  - ciascuno sviluppatore combina Coding, Testing, Listening, Design
  - riduzione della specializzazione dei compiti
  - comunicazione e feedback rapido dall'utente, dagli altri sviluppatori, dai tests
- Evitare la produzione di semilavorati non strettamente necessari
  - molto usati gli use case diagrams, poco tutto il resto (tra cui i class diags)
  - centralità del codice e dei tests
  - sostituito dalla vicinanza e la condivisione del lavoro

- integrazione di analisi, progettazione, programmazione, test
  - dal programmatore" allo "sviluppatore"



- Processo controllato attraverso pratiche più che documenti

■ ...remark: the picture is somehow unfair



- Small releases
  - Sistema in produzione entro breve (un mese)
  - Releases operative ravvicinate (2-4 settimane)
- Continuous integration
  - Il codice viene continuamente integrato in una versione comune (ore)
  - Supporto di strumenti (Git, Jira, ...)
- Simple design
  - Tutti gli sviluppatori comprendono il design
  - Il design non è mai up-front
  - (XP vs design patterns)
- Refactoring
  - Modificare la struttura senza modificare il comportamento
  - Valore del coraggio
  - Supporto di strumenti

- On-site customer
  - Gli utenti partecipano operando in sito nel corso di tutto lo sviluppo
- Planning game
  - Gli utenti definiscono i requisiti attraverso storie
  - Gli sviluppatori stimano il costo delle storie (può variare col tempo)
  - Gli utenti scelgono le storie realizzate nel prossimo incremento secondo un compromesso tra costo realizzato e tempo del rilascio
  - Gli sviluppatori scompongono in tasks le storie scelte
  - Gli sviluppatori scelgono i tasks
  - (velocity)

- <http://www.solutionsiq.com/agile-glossary/spike/>
  - A story or task aimed at answering a question or gathering information, rather than implementing product features, user stories, or requirements. Sometimes a user story is generated that cannot be estimated until the development team does some actual work to resolve a technical question or a design problem. The solution is to create a “spike,” which is a story whose purpose is to provide the answer or solution. Like any other story or task, the spike is then given an estimate and included in the [sprint backlog](#).
- <http://stackoverflow.com/questions/253789/agile-vs-spiral-model-for-sdlc>
  - I believe that Agile is nothing but another implementation of Spiral Model. I am a big supporter of Spiral (The spiral model is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts) since its beginnings and have seen that lot of projects implement Spiral without knowing that they are operating in a Spiral world. Since the day Agile started gaining popularity the concept of spiral started getting overlooked a little bit. I am sure that for complex projects spiral is still the best alternative but I would like to get a better understanding of the similarities and differences between Agile and Spiral techniques. Can anyone explain their differences/similarities?
    - *TBD: Collegare al planning game, anche come richiamo al concetto della spirale*

- **Metafora**
  - Visione condivisa e concisa degli obiettivi del progetto
- **Collective ownership**
  - Il codice integrato è di tutti e tutti lo possono modificare
- **Pair programming**
  - Un computer, una tastiera, un mouse
  - Coppie formate di task in task
  - Fattore di continuità e omogeneizzazione del team

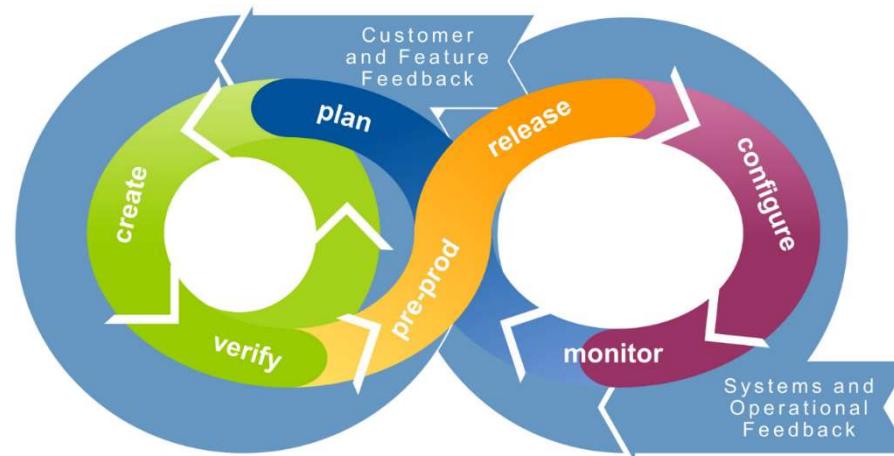
- Tests
    - I programmatore scrivono gli unit tests dei tasks prima del codice
    - I committenti scrivono i functional tests
    - Il codice viene integrato quando passano gli unit tests
    - La release è chiusa quando passano i test funzionali
    - Supporto di strumenti (Junit, Mockito, Selenium, ...)
  - Coding standards
    - Aumenta la capacità autodocumentale del codice
  - 40 hours week
  - Open space
- ... they are just rules

## ■ Problemi

- Rischio nella stima dei costi (velocity)
- Richiede cooperazione dei committenti
- E' difficile il contratto
- Peopleware
- Competenze, età ed evoluzione dei ruoli
- Limitate dimensioni del team
  - 4 Programmatori, 1 team leader, 1 project responsible
  - Riportati casi fino a 10

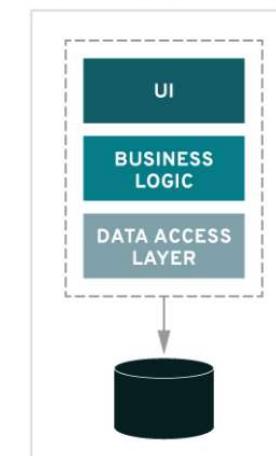
## ■ Condizioni di migliore funzionamento

- progetti di breve durata (Internet age)
- progetti con requisiti instabili (approccio esplorativo, progetti di ricerca)
- (connection to MicroServices and DevOps)

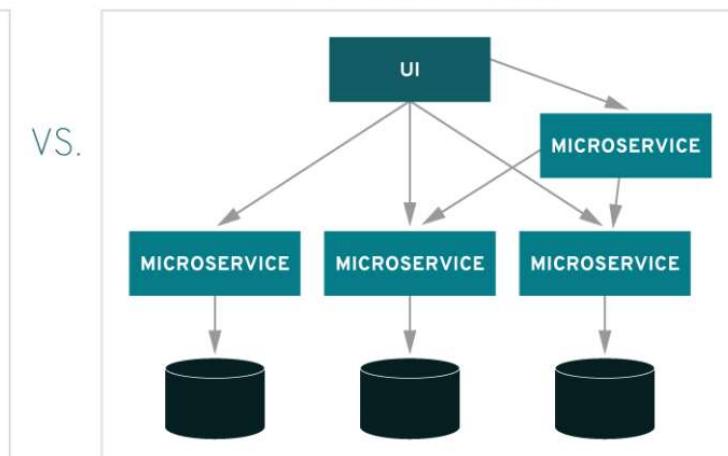


Cisco, its logo and its affiliates. All rights reserved.

MONOLITHIC



MICROSERVICES



- Diverso rapporto tra costo della previsione e dell'imprevisto
- Teoria classica (Bohem 1976)
  - Il costo di una modifica cresce esponenzialmente con l'avanzamento



- Claim di XP (Beck 2000)

- tecnologie di testing,  
refactoring, versioning
  - Object orientation

- Internet age

- Il topolino, il dinosauro e la glaciazione

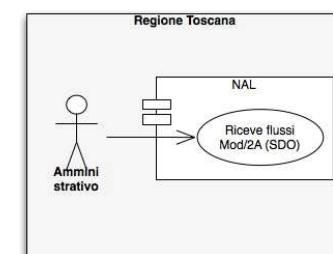
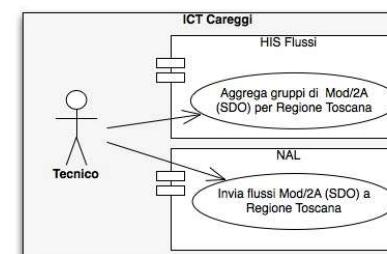
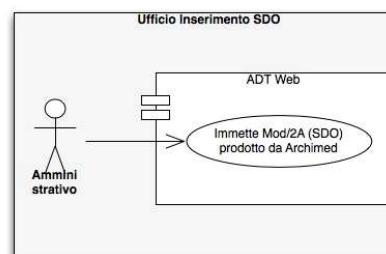
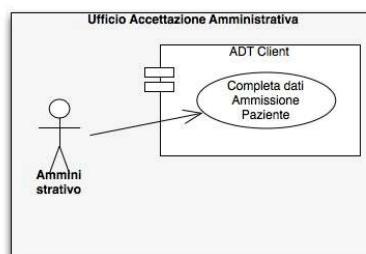
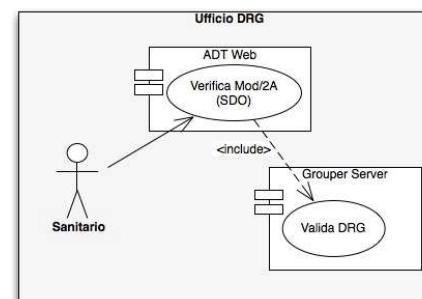
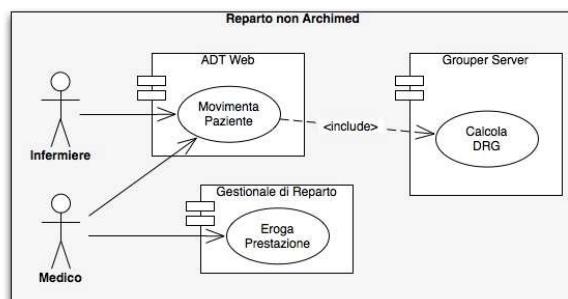
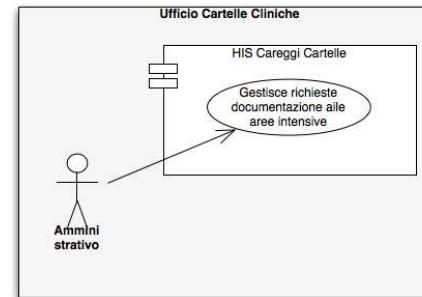
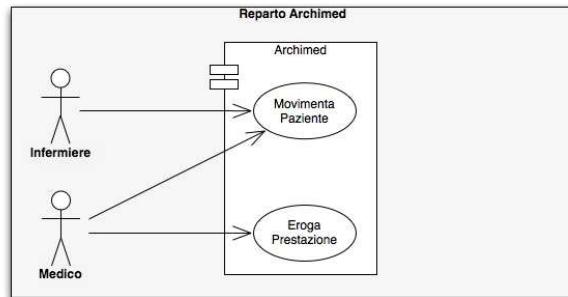
- Waterfall
  - W.W.Royce, "Managing the development of large software systems: Concepts and techniques," Proc.IEEE WestCon, 1970.
  - B.Bohem, "A spiral model of software development and enhancement," IEEE Computer, 1988.
- Incremental
  - H.D.Mills et alii, "The management of software engineering," IBM Systems Journal, Vol.24, No.2, 1980.
- UP
  - I.Jacobson, G.Booch, J.Rumbaugh "Unified Software Development Process," Addison Wesley 1999.
  - J.Arlow, I.Neustadt, "UML e Unified Process," McGraw Hill, 2003.
  - [https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\\_bestpractices\\_TP026B.pdf](https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf)
- XP
  - K.Beck, "Embracing change with eXtreme Programming," IEEE Computer, Vol.32, No.10, 1999.
  - K.Beck, "Extreme programming explained:embracing change," Addison Wesley, 2000.
  - W.Strigel, "Reports from the field: using XP and other experiences,"Editorial Intro for special issue on IEEE software, Nov/Dec 2001.



# Executable architecture: il CART Il problema - partizionamento

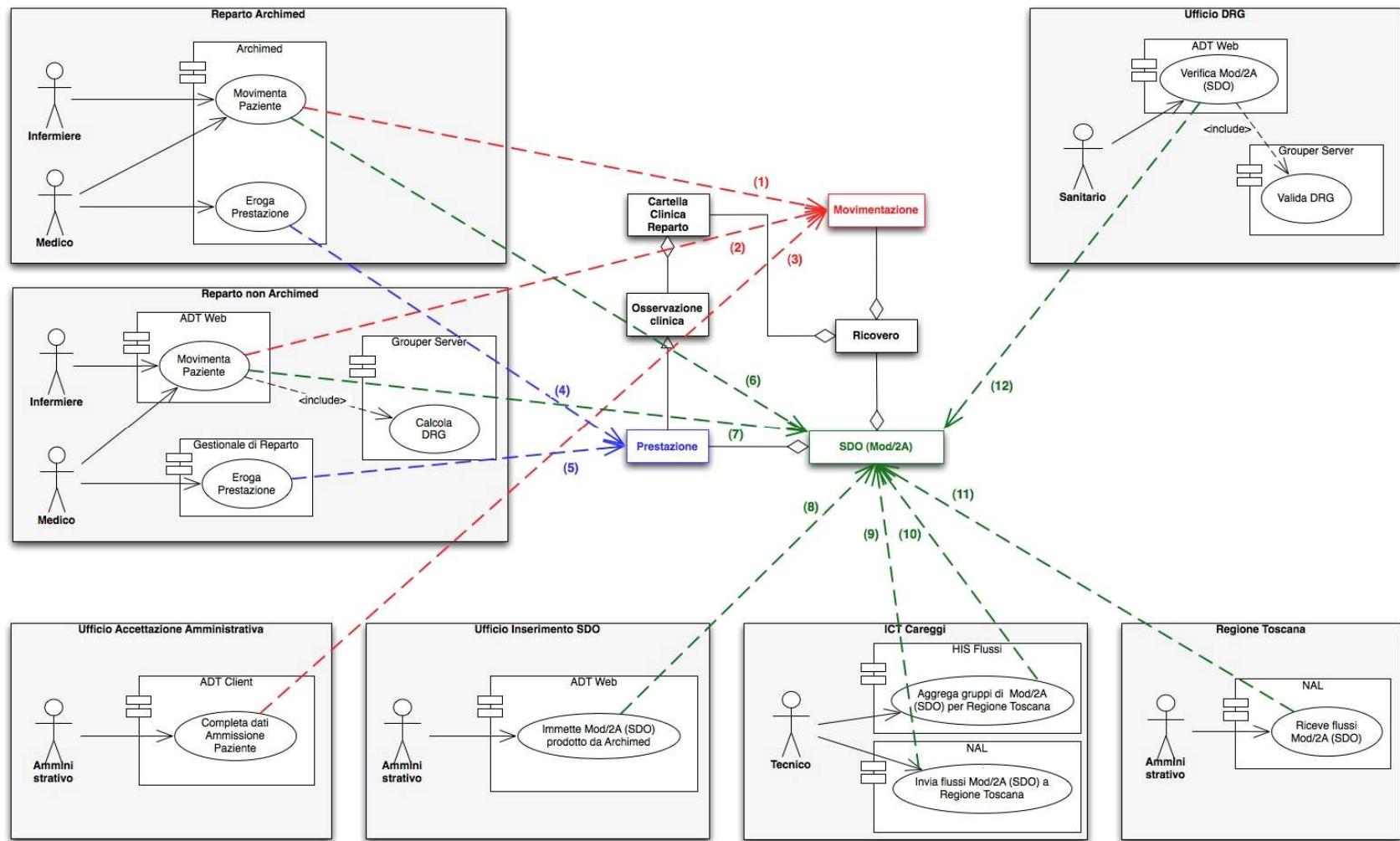
In un sistema complesso è necessario partizionare le responsabilità:

- separare ruoli, anche in relazione a diversi livelli e località di competenza;  
abilitare iniziativa autonoma e molteplicità di fornitori; vincoli di legacy;  
... ridurre la complessità



■ ... ma il partizionamento non è mai perfetto, e resta dell'accoppiamento

- esistono concetti nel modello di dominio su cui intervengono più casi d'uso allocati ad applicazioni separate
- Esempio: la Scheda di Dimissione Ospedaliera - a Careggi

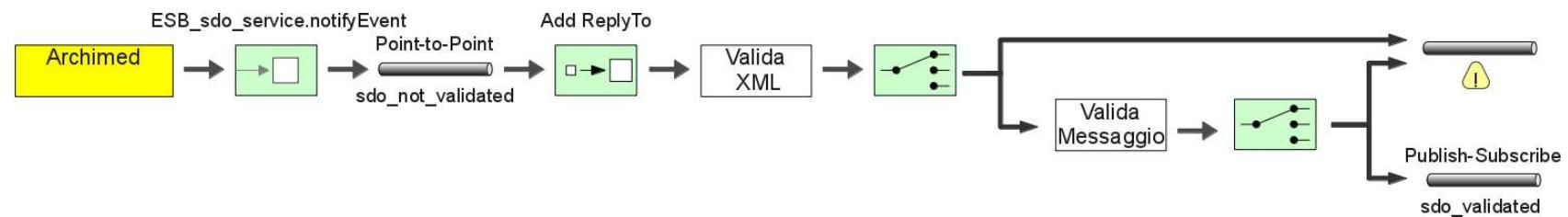


## ■ Enterprise integration patterns

- ... "large-scale integration solutions across many implementation technologies"
- File exchange, Tabelle di frontiera, Chiamata in contesto, ... aggiustamenti
- Remote Procedure Call, Message passing
- Many different ways, some dubious, each applicable if the aim is adaptation

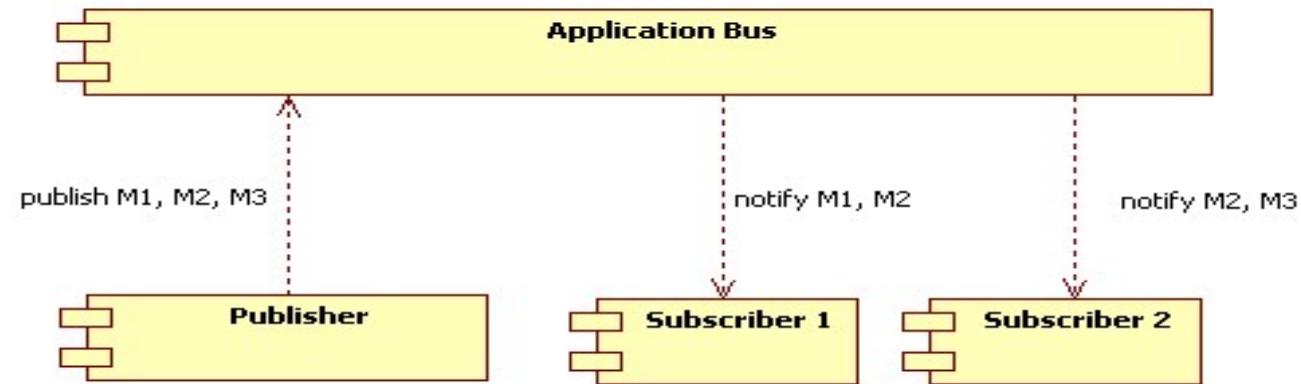
## ■ Occorre un'architettura di integrazione

- (la metafora del piano regolatore)
- modelli di riferimento, supporto infrastrutturale alla loro realizzazione
- serve a disaccoppiare lo sviluppo e la manutenzione, ... anche i contratti



## Il CART è un'architettura di integrazione

- prevalentemente Event Driven Architecture (EDA)



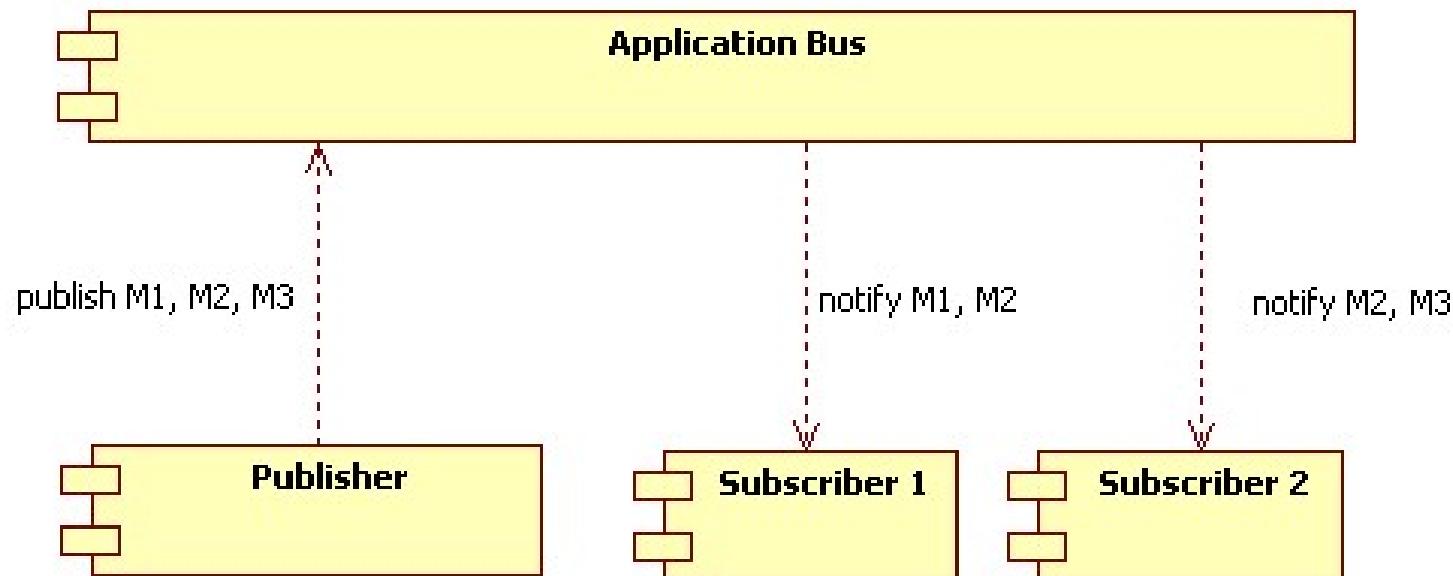
... compara con Service Oriented Architecture (SOA),  
un tema intorno al 2005

- Request&reply vs Publish&Subscribe
- Incentrato su business events più che su business processes
- ... molto diverso il modello di governance

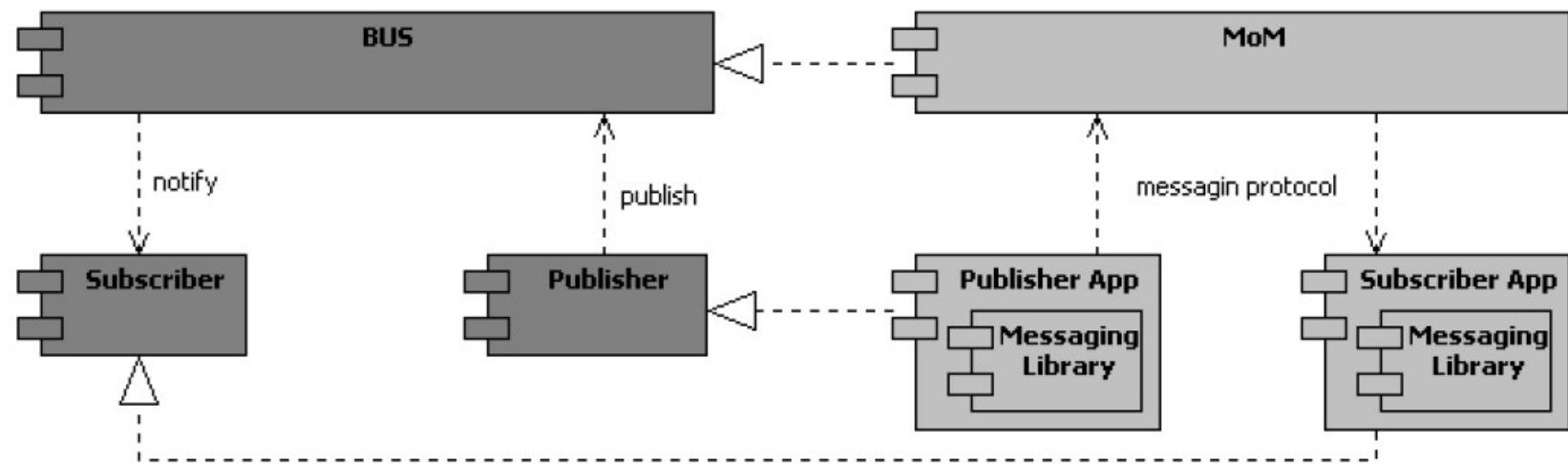
Il CART non è solo un modello

- anche una piattaforma tecnologica resa disponibile come servizio
- ... anche un processo di governance

- Un sistema di cooperazione publish & subscribe
  - Also Known As: broker, event channel
  - Equivalente architetturale del pattern observer
- Participants:
  - Publisher: pubblica un evento su un topic sul broker
  - Subscriber: sottoscrive un topic sul broker, accetta notifiche dal broker
  - Broker: accetta eventi e li inoltra a tutti sottoscrittori del topic



- Il Bus è implementato con un Message Oriented Middleware (MoM)



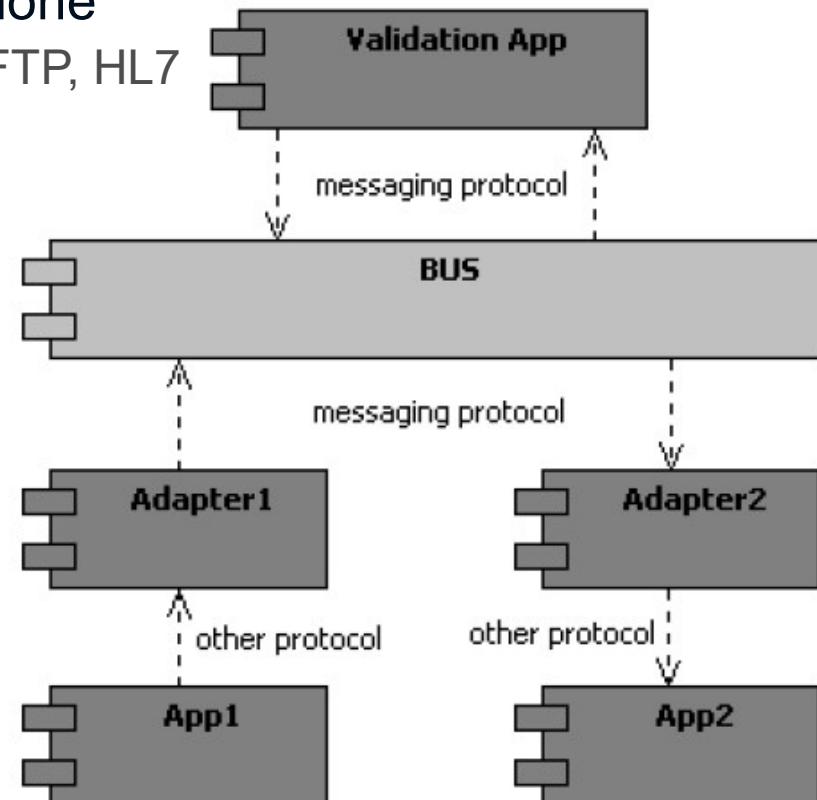
- Le applicazioni possono essere realizzate con tecnologie diverse
  - ciascuna possibilmente supportata da una Messaging Library

- Adapter: converte i protocolli di interazione
  - http, JMS, Soap, WS, JDBC, TCP/IP, FTP, HL7

- Applicazioni infrastrutturali possono estendere il bus
  - E.g. filtraggio, validazione, conversione di formato

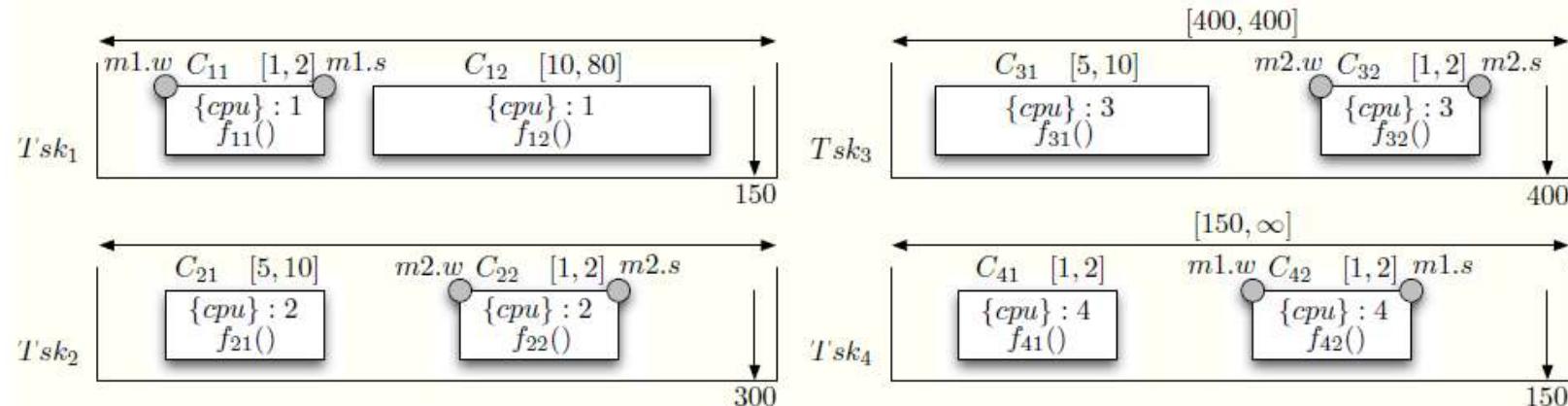
## Enterprise Service Bus

- estende un Message Oriented Middleware con i componenti che abilitano Enterprise Integration Patterns
- Adapters, strumenti di routing



## Architettura di un real time task-set

- Un set di tasks concorrenti
- Ciascun task rilascia jobs, composti da chunks
- priority driven preemptive scheduling
- chunks sincronizzati su risorse (semafori)



## I chunks realizzano funzioni

- Ciascuno implementato da una funzione c (entrypoint)
- ... che può chiamare altre funzioni, ma non modifica l'uso delle risorse

## Executable architecture

- I chunks sono emulati da una funzione busy-sleep
- E' implementata l'architettura ma manca completamente la funzionalità

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

### An example of problem statement – una rete di biblioteche

- X è un sistema di gestione integrato delle attività di fruizione, di gestione front-office e back-office di una rete cittadina di biblio+media+teche che conservano volumi, riviste, audio e video in un a varietà di media diversi.
- X incide sul servizio di prestito fornito dal Comune migliorandone l'usabilità e l'efficacia per gli utenti. Gli utenti possono consultare il catalogo attraverso il web da loro postazioni private oppure da una o più postazioni dislocate presso ciascuna sede. Il servizio prevede anche la possibilità di ricevere assistenza nell'accesso al sistema da parte del personale che opera presso le sedi. Ciascun oggetto nel catalogo è associato ad una descrizione specializzata in base alla natura dell'oggetto, al suo medium e alla sua specifica categoria.
- La consultazione del catalogo include funzioni di ricerca per titolo o autore, completamento, ordinamento e filtraggio secondo una varietà di criteri basati su tipologia di oggetto, medium, classificazione tematica, sede di conservazione. Essa permette di verificare la disponibilità dell'oggetto cercato e identificarne la locazione presso una o più biblioteche della rete.
- Gli utenti registrati presso una delle sedi e titolari di una tessera di iscrizione possono compilare la richiesta di prestito, che include la verifica del corretto stato di uso del prestito da parte dell'utente. Essi possono anche proporre l'acquisizione di nuovi oggetti e rilasciare propri commenti e classificazioni folksonomiche esposti in modo anonimo circa gli oggetti che hanno consultato. Tali informazioni possono essere tenute in conto come elemento di filtraggio a scelta dell'utente. Un utente non registrato può richiedere la registrazione via Web ma deve comunque poi averla autenticata attraverso identificazione presso una delle sedi.
- X mira anche ad incidere sull'efficienza ed il costo di erogazione del servizio fornendo interfacce efficaci agli operatori di front-office e abilitando la condivisione delle attività di back-office. In particolare X mira ridurre il costo di dislocazione e manutenzione di HW e SW presso le diverse sedi attraverso una architettura web-based, nella quale l'archivio centrale e il server sono dislocati presso una unica sede con funzione di coordinamento.
- Gli operatori di front-office supervisionano il funzionamento "fisico" delle diverse sedi secondo le modalità consuete di una biblioteca e assistono o sostituiscono gli utenti nell'accesso al sistema informativo. Il sistema offre all'operatore le funzioni di registrazione di un utente con assegnamento di una tessera personale con numerazione univoca, registrazione di un prestito e registrazione di un rientro.
- Gli operatori di back-office sono supportati dal sistema nella analisi statistica dello stato dei prestiti, la verifica dei ritardi e l'inoltro di solleciti, la gestione di nuove acquisizioni e il mantenimento di un elenco di fornitori.
- Il sistema deve inizialmente operare su una rete di 8 biblioteche prevedendo un carico di utenza nell'ordine di 16000 accessi al giorno, avendo però la capacità di scalare fino a 32 nodi e 64000 accessi attraverso il potenziamento del nodo centrale e delle interconnessioni delle sedi. Il tempo di risposta nell'accesso ad una voce del catalogo da parte di un generico utente web deve rimanere contenuto al disotto di 10 secondi, fatto salvo il ritardo sul lato utente. Il tempo di accesso nell'accesso al catalogo effettuato dall'interno di una delle sedi non deve eccedere i 2 secondi.
- Tutte le interfacce devono soddisfare i requisiti di accessibilità al livello di AA.
- Il sistema gestisce in modo sicuro la autenticazione dei diversi ruoli e garantisce il mantenimento di un log delle operazioni con caratteristiche di non-revocabilità.

- La Scuola di Medicina offre scuole di specializzazione per una varietà di professioni sanitarie. La scuola di Medicina ha un Direttore, e ciascuna Scuola di Specializzazione ha un Coordinatore. Il Direttore e i Coordinatori sono professori universitari.
- Ciascuna specializzazione richiede lo svolgimento di un insieme di esperienze assistenziali in diversi ambiti di attività medica.
- L'attività di formazione è sviluppata in collaborazione con un numero di Aziende Ospedaliere o Sanitarie (e.g. Careggi, Meyer, Toscana-Centro-Firenze, Toscana-Centro-Prato, ...). Ciascuna Azienda ha un Direttore Generale ed è articolata in un insieme di strutture organizzative (i.e. un reparto ospedaliero, un ambulatorio, ...), presso ciascuna delle quali operano medici specializzati in diversi ambiti di attività. Ciascuna Struttura Organizzativa ha un primario e può ospitare contemporaneamente un certo numero di studenti in specializzazione per ciascun ambito di attività.
- Lo studente iscritto ad una scuola di specializzazione ha un piano formativo che identifica le attività assistenziali concrete attraverso le quali realizza le esperienze richieste per il percorso di specializzazione. Ciascuna attività concreta è svolta presso una struttura organizzativa di un'azienda, in un periodo di tempo durante il quale lo studente opera sotto la supervisione di un medico della struttura. Il piano formativo individuale è redatto da una unità amministrativa della Scuola di Medicina ed è approvato dal Coordinatore della scuola di specializzazione e dai Primari delle Strutture Organizzative che ospitano le attività.
- Nel corso dello svolgimento di ciascuna attività formativa, lo studente redige un registro delle presenze, indicando per ciascuna la data, le ore di presenza, e l'attività svolta. Al termine del periodo, il medico supervisore può visionare e validare il registro delle presenze dello studente, e redige una relazione di sintesi delle attività svolte da ciascun studente supervisionato. Ciascun medico impegnato in ruoli di supervisione redige anche un registro personale del tempo che ha impegnato in attività di formazione di degli studenti che ha supervisionato. Il registro identifica l'impegno dedicato su una scansione settimanale.

- ♣ The school of medicine delivers a number of specialization schools for a variety of health professions. The School has a director, and each specialization schools has a coordinator. The Director and Coordinators are university professors.
- ♣ Each specialization requires that the student is trained by carrying out a set of practical experiences of care in different areas of medicine.
- ♣ The training activity is developed in collaboration with a number of Hospital and Healthcare Organizations (e.g. Careggi Hospital, Meyer Hospital, Central Tuscany Healthcare Organization in its sites of Florence, Prato, Pistoia, Empoli, ...). Each Organization has a General Manager and is divided into a set of organizational structures (i.e. a hospital ward, a medical clinic, ...), each specialized in different areas of activity. Each organizational structure has a head physician and can simultaneously host a certain number of students in specialization for each area of activity.
- ♣ The student enrolled in a specialization school has an individual training plan that identifies the concrete assistance activities through which he / she realizes the experiences required for the specialization path. Each concrete activity is carried out at an

organizational structure of some organization, in a period of time during which the student works under the supervision of a doctor of the structure. The individual training plan is drawn up by an administrative unit of the school of medicine and is approved by the coordinator of the specialization school and by the general director of the organizational structures that host the activities.

♣ During the course of each training activity, the student draws up an attendance register, reporting date, hours of attendance, and activities carried out. At the end of the period, the supervising doctor can view and validate the student attendance register, and draw up a summary report of the activities carried out by each supervised student. Each physician engaged in supervisory roles also draws up a personal time log that he has engaged in training activities for students he supervised. The register identifies the effort dedicated by each physician on a weekly basis.