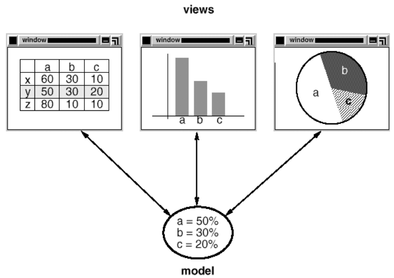
DESIGN... PATTERNS?

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over
- Christopher Alexander on architecture patterns

The Model/View/Controller (MVC)



GOF PATTERN CATALOG

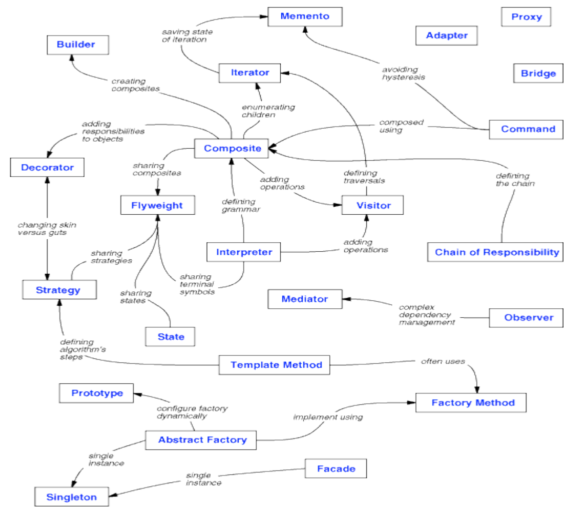
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory(87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility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

FOUR PARTS

- The **pattern name** Finding good names has been one of the hardest parts of developing our catalog.
- The **problem** describes when to apply the pattern. It explains the problem and its context.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
- The **consequences** are the results and trade-offs of applying the pattern., they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

Design for change

- **Creating an object by specifying a class explicitly**
  `Abstract factory, Factory Method, Prototype`
- **Dependence on specific operations**
  `Chain of Responsibility, Command`
- **Dependence on hardware and software platforms**
  `Abstract factory, Bridge`
- **Dependence on object representations or implementations**
  `Abstract factory, Bridge, Memento, Proxy`
- **Algorithmic dependencies**
  `Builder, Iterator, Strategy, Template Method, Visitor`
- **Tight Coupling**
  `Abstract factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer`
- **Extending functionality by subclassing**
  `Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy`
- **Inability to alter classes conveniently**
  `Adapter, Decorator, Visitor`



Software Engineering A.M.Calvagna, 2024 Università di Catania

Principi di progettazione riusabile

Design Principle 0

Principio della singola responsabilità

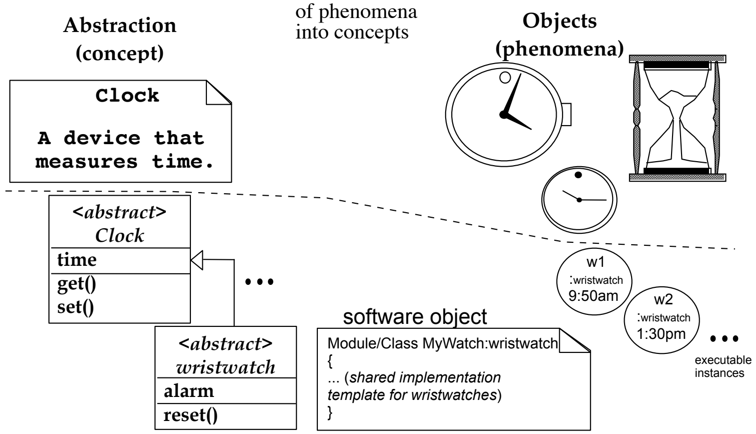
Design methods to perform a single specific task, related to its defining class/ADT

Design Classes/ADT to clearly represent one single concept

Software Engineering A.M.Calvagna, 2024 Università di Catania

Clock example

- Abstraction = Classification of phenomena into concepts



Software Engineering A.M.Calvagna, 2024 Università di Catania

Design Principle 1

Program to an interface, not an implementation

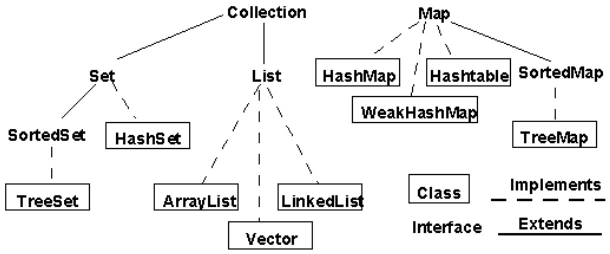
Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

Declare variables to be instances of the abstract class not instances of particular classes

Programming to an Interface: Java Collections

```
Collection students = new XXX;
students.add( aStudent);
```

- students can be any collection type
- We can change our mind on what type to use



Design Principle 2

Favor object composition over class inheritance

Composition

- Allows behaviour changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance vs Composition

Inheritance

```
class A {
   Foo x
   public int complexOperation() { … }
}

class B extends A {
   public void bar() { … }
}
```

Composition

```
class B {
   A myA;
   public int complexOperation() {
      return myA.complexOperation()
   }

   public void bar() { … }
}
```

Design Principle 2

Use Parametrised types

- Parameterized types give a third way to compose behavior in an object-oriented system
- It gives the flexibility of dynamic (run-time) binding, just as like composition does

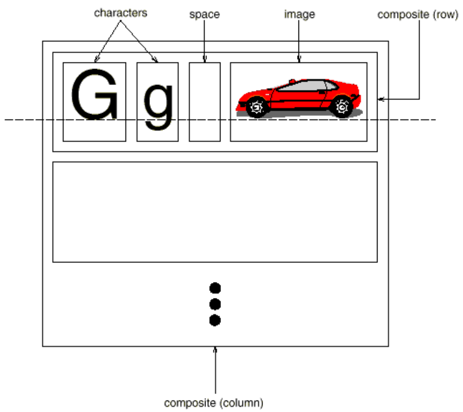
Parameterized Types

- Generics in Ada, Eiffel, Java
- Templates in C++
- Allows you to make a type as a parameter to a method or class

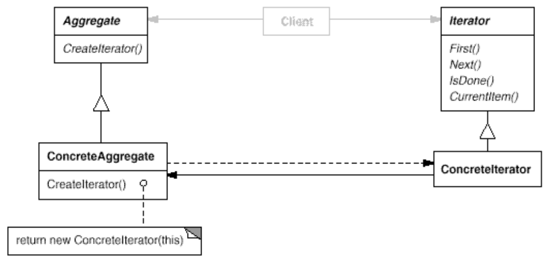
```
template <class TypeX>
TypeX min(  TypeX a, TypeX b )
   {
   return a < b ? a  :  b;
   }
```


Patterns case study: Document editor

- design problem: how to design a heterogeneous data structure so to allow for easy navigation of its elements?



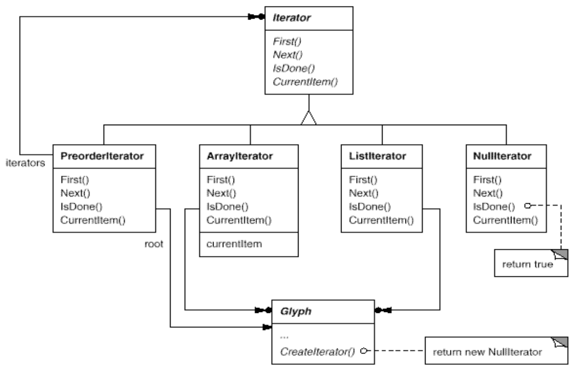
Solution: use the Iterator pattern



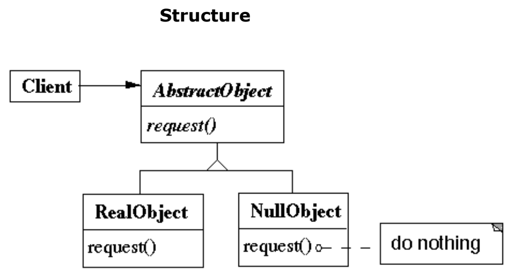
•to access an aggregate object's contents without exposing its internal representation.
•to support multiple traversals of aggregate objects.
•to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

•See Java *collections*

Iterator Pattern applyed



Null Object



NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing

Esercitazione

- implementare un ADT albero binario di ricerca.
- Inserimento di un nuovo nodo.
- Ricerca di un elemento

- scrivere un programma client che riempie la struttura dati con numeri da tastiera.

- dove/come usare iterator e null object?

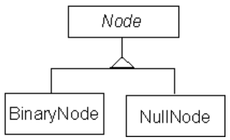
Binary Search Tree Example

Without Null Object

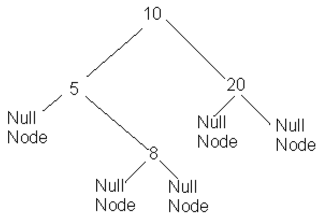
```
public class BinaryNode {
   Node left = new NullNode();
   Node right = new NullNode();
   int key;

   public boolean includes( int value ) {
      if (key == value)
         return true;
      else if ((value < key) & left == null) )
         return false;
      else if (value < key)
         return left.includes( value );
      else if (right == null)
         return false;
      else
         return right.includes(value);
   }
etc.
}
```

Binary Search Tree Example

Class Structure



Object Structure



Searching for a Key

```
public class BinaryNode extends Node {
   Node left = new NullNode();
   Node right = new NullNode();
   int key;

   public boolean includes( int value ) {
      if (key == value)
         return true;
      else if (value < key )
         return left.includes( value );
      else
         return right.includes(value);
   }
etc.
}

public class NullNode extends Node {
   public boolean includes( int value ) {
      return false;
   }
etc.
}
```


Refactoring…

- Introduce Null Object
  - You have repeated checks for a null value?
  - Replace the null value with a null object

```
If (customer==null) plan = new BasicBillingPlan();
    else plan = customer.getPlan();
```

Becomes:

```
plan = customer.getPlan();
```

```
Class Nullcustomer {
  getPlan() {  return new BasicBillingPlan(); }
}
```


Applicabilità

Use the Null Object pattern when:

- Some collaborator instances should do nothing
- You want clients to ignore the difference between a collaborator that does something and one that does nothing

  Client does not have to explicitly check for null or some other special value

- You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way