Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Design Patterns

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

What are design patterns?

In programming there are problems that recur over and over again. To many of these problems there are good solutions which can be described in general terms. Using them one can avoids design errors and reduce code replication.
Some of those solutions have become rather standard and indicated by design patterns. Design Patterns are normally described independently form the specific programming language that will implement them. They are, however, strongly based on the *object programming paradigm*.

Examples of pattern are the Factory or the Decorator. Knowing patterns helps you to identify well designed solution for a large variety of common programming problems.

Main references on design patterns

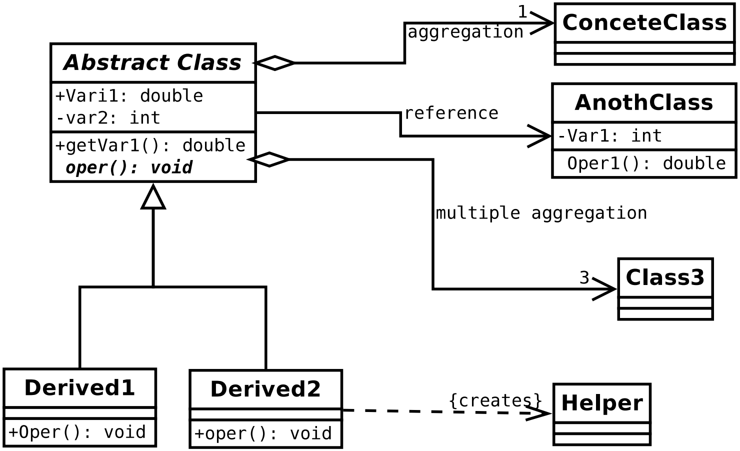
- The book *Design Patterns, elements of reusable object-oriented software* by E. Gamma, R. Helm, R. Johnson, J. Vlissides. Also known as *The Gamma Book* or *The Gang of Four (Gof) Book*, it is still one of the main reference on design patterns.
- The book *Modern C++ Design* by A. Alexandrescu, shows how the coupling of Design Pattern and Generic Programming techniques can bring C++ at its full power. It illustrates generic templated solution for some design patterns.
- The book C++ Programming with Design Patterns Revealed by Muldner introduces the C++ language in terms of design patterns.
- Wikipedia. The pages on design patterns are well written.

Software for design pattern

- The Loki library by A. Alexandrescu, available on sourceforge, implements the design patterns of the *Modern C++ Design* book (and much, MUCH more).
- The Boost project provides libraries with the generic implementation of some patterns.
- The C/C++ implementation of some of the patterns illustrated in the GoF book is available on the net (but it's only illustrative).

UML diagrams

Many design patterns can be described using an UML diagrams. Here the main definitions.



Design pattern subdivision

The Gof book subdivides patterns into three main categories

- Creational Patterns. They concern the process of creating objects. Example: Abstract Factory, Object Factory.
- Behavioural Patterns. They specify how classes interact and distribute responsibilities. Example: Iterator.
- Structural Patterns. They deal on the way classes and objects are composed to form more complex structures. Example: Decorator.

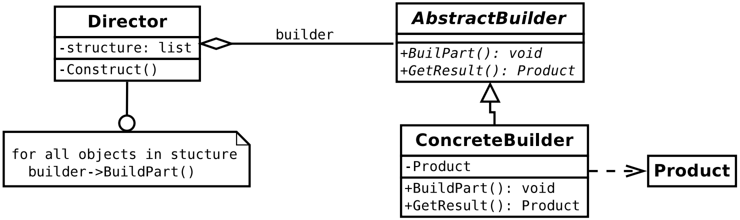
Factories

The main creational pattens are the *Factories* (which can be subdivided into Object Factory and Clone Factory, the Builder, and the Abstract Factory.

They are somehow similar, since the objective of all of them is to create objects, typically returning a pointer to a base class. They differ in the scope:

- Builder.: Separate the construction of a complex object from its representation so that the same construction process may take different representations.
- Factory Methods. Defines and interface to create an object, but delegates to other classes or methods which subclass to instantiate. A Factory Methods lets a class defer the instantiation of subclasses.
- Abstract Factory. Provides an interface for creating families of related and dependent objects without specifying their concrete class.

The builder



Director is made of parts which is created by the builder.

Example of builder: a maze

```
 class MazeBuilder {
public:
    virtual void BuildMaze();
    virtual void BuildRoom(int room);
    virtual void BuildDoor(int roomFrom, int roomTo);
    virtual Maze* GetMaze();
protected:
    MazeBuilder();
private:
    Maze M_maze;
};

Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();
    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);
    return builder.GetMaze();
}
```

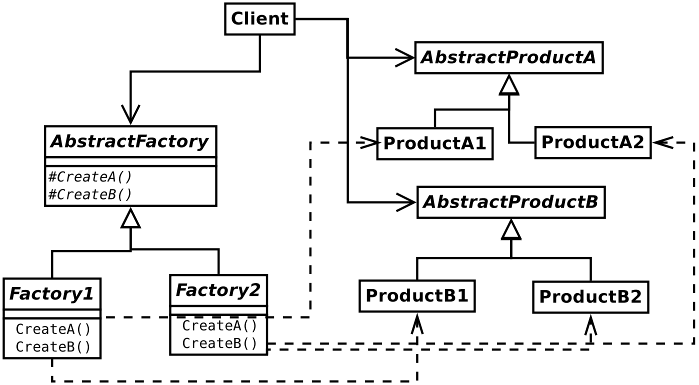
Implementation issues

The intention is to abstract steps of construction of complex objects (the `Maze` in the example) so that different implementations of these steps can construct different representations of objects.

Often, the builder pattern is used to build products in accordance with the composite pattern.

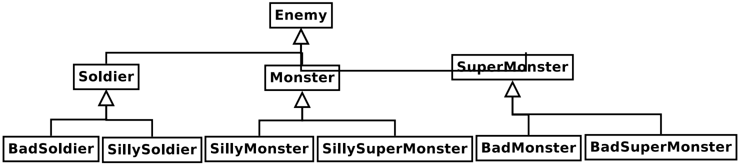
Abstract Factory

The Abstract Factory provides an interface for creating *families* of related or dependent polymorphic objects without specifying their concrete class.
We use an example from Alexandrescu's book. But first let's look at the generic layout



An example

We are developing a shoot'em all game were the Enemy may be Soldiers, Monsters or SuperMonsters. We are planning to have two levels, one for the beginners and one for the advanced users. So we have two implementations of each Enemy.



The Enemy Factory

The construction of the "Enemies" is delegated to a concrete implementation of an abstract class containing the prototype methods.

```
 #include "enemies.hpp"
class AbstractEnemyFactory{
public:
 virtual Enemy * MakeSoldier()=0;
 virtual Enemy * MakeMonster();
 virtual Enemy * MakeSuperMonster();
};
...
class AdvancedLevelEnemyFactory : public AbstractEnemyFactory
{
public:
 Enemy * MakeSoldier(){return new BadSoldier;}
  ...
};
class BeginnerLevelEnemyFactory : public AbstractEnemyFactory
{
public:
 Enemy * MakeSoldier(){return new SillySoldier;}
  ...};
```

The client application instantiates the correct implementation according to the chosen level.

```
std::vector<Enemy* > theEnemies;
int level;
std::cout<<"Give me level: 0 Beginner, 1 Advanced"<<std::endl;
std::cin>>level;
AbstractEnemyFactory * enemyFactory;
if (level==0)
        enemyFactory= new EasyLevelEnemyFactory;
else
        enemyFactory= new AdvancedLevelEnemyFactory;

theEnemies.push_back(enemyFactory->MakeMonster());
theEnemies.push_back(enemyFactory->MakeSuperMonster());
theEnemies.push_back(enemyFactory->MakeSoldier());

for (unsigned int i=0; i<theEnemies.size();++i)
       theEnemies.at(i)->speak();
```

The actual example

The example may be found in the directory
DesignPatterns/AbstractFactory
of the examples. This version uses unique_ptr<> to avoid the risk of memory leaks.

The Abstract Factory may be integrated with other design patterns, such as the Prototype pattern, to make it more flexible. A C++ implementation of the prototype pattern can be made via the Virtual Copy Constructor (Clonable classes). In our example we could have added a virtual clone() method to our hierarchy of "Enemies". Then we could have set a single Factory

```
class EnemyFactory{
 public:
  // sets the type we want: bad or silly
  setSoldierType(auto_ptr<Soldier>);
  Enemy * MakeSoldier{ return M_soldier->clone();}
  ...
 private:
  unique_ptr<Soldier> M_soldier;
```

A generic implementation of the Abstract Factory is present in the Loki library. It it rather sophisticated and makes heavy use of template metaprogramming.

The Factory Method

The Factory method defines an interface for creating an object out of a polymorphic hierarchy. There are many variants

- The parametrised factory, also called Object Factory. The type of the object is determined through a user defined identifiers. We have already seen a few examples in the course:

```
BaseClass * a = factory.CreateObject("MyType");
```

- The Clone Factory. It creates a clone of an object through the pointer to a base class. It is useful when the classes have not been designed to be Clonable.

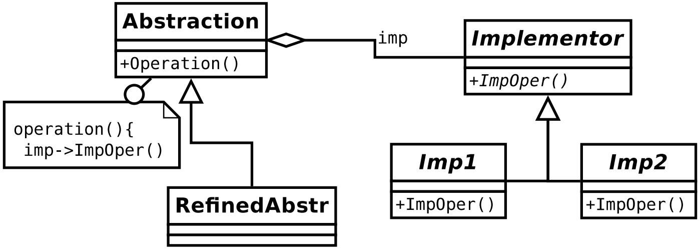
```
void fun (BaseClass & a){
// I need a new copy of a,
// but  I don't know the actual type of the object
BaseClass * b = Clonefactory.CreateObject(a);
```

A clone factory may in fact be implemented in terms of an object Factory. The implementation, however, is not trivial at all!

Object and Clone factories are normally implemented a Singleton objects. The Singleton itself is an example of Design Pattern!. Templates for generic Singletons, Object and Clone Factories are contained in the Loki library. An implementation of the Clone Factory is contained also in the *Functional* library of the *Boost* project.

Bridge

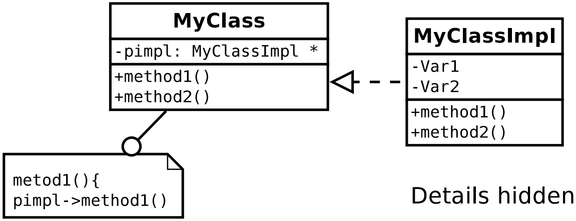
A Bridge is a structural pattern that aims to decouple an abstraction from its implementation, so that the two may vary independently. We have indeed already used this pattern.



Bridges and Pimpl

The Bridge pattern is also used to completely hide the implementation of a class. The so called Pimpl idiom (Pointer to implementation) is indeed a special application of the Bridge pattern (used in C++ programming). In this idiom the implementation of a class is completely hidden from the user by using a pointer to a separate class.
Indeed in standard C++ programming the implementation cannot be completely hidden, since the definition of a class contains the definition of its private members. At the price of an additional indirection one may deny the user this type of information.

The Pimpl idiom



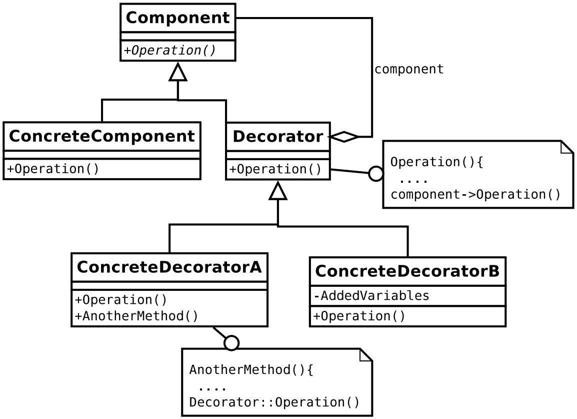
User interface

Details hidden from the user

Code stored in a library

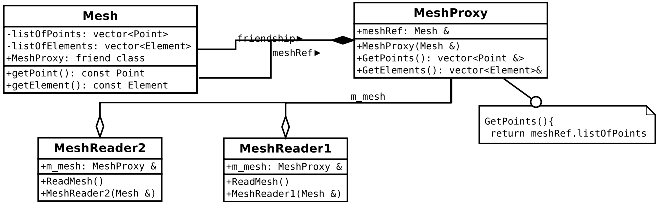
Decorator

A Decorator attaches additional responsibilities to an object (either statically or dynamically). It is an alternative to subclassing for extending the functionality of a class. In C++ they are implemented through a combination of inheritance and aggregation (we have already seen an example in the course).



Proxy

A Proxy is a structural patterns that is used as a placeholder to another object to provide controlled access to it. For instance suppose that in order to make thing safer we want to make all methods of our Mesh class constant (so inexperienced users cannot make a mess). Yet we need to read the mesh and we can have many mesh readers... How can they have a full fledged access to my mesh? One may use a Proxy class which exposes the Mesh data.



Iterator

The Iterator provides a way to access elements of an aggregate object sequentially, without exposing its underlying representation. An example: the iterators of the STL containers.

Strategy

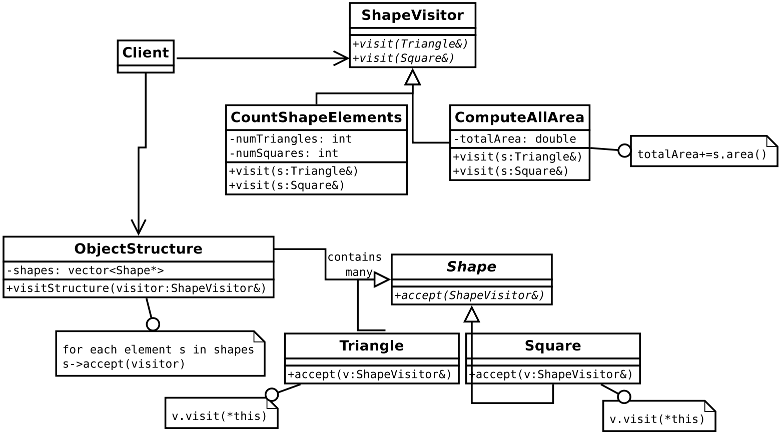
The Strategy pattern is a behavioral pattern where a family of algorithms are encapsulated and used interchangeably to modify the behavior of an operation.
The Strategy pattern and Policies are strictly related. The term Strategy, however, refers to the general design, policies are instead specific C++ programming technique that implement the Strategy pattern.
Moreover, Strategies are normally *bound run-time* while Policies are can also *statically bound* as template parameters.
Yet, static and dynamic binding may coexist: the comparison operator for a std::set<>, for instance, may be given as a policy via a template argument, or as an object passed as an argument of the constructor. We are in both cases implementing a Strategy pattern.
There is a similarity between Strategy and Bridge patterns, as their general implementation is similar. However the Bridge is meant to decouple the entire implementation (or large part of it), the Strategy, instead, modifies the behavior of specific methods.

Visitor

It represents an operation to be performed on the elements of an object structure (for instance a STL vector or a binary tree). Visitors lets you define a new operation without changing the classes of the elements on which it operates.
It implements double-dispatching: the method that will be called depends both on the type of the visitor and on that of the visited object!

An example

Suppose we have a vector of pointers to Shapes, a base class for a hierarchy of geometrical shapes (Triangle, Square and so on). We want to extract some statistics on those shapes (for instance the total area, or the number of objects of each type...). Moreover we want to be able to extend the type of statistics without changing the interface of Shape. How can it be done?
The idea is that the Shapes must have a method, let's call it accept() which has as argument a reference to a base class called Visitor. Each concrete Visitor class, in turns, derives from the base class Visitor and implements several method called visit(), one for each concrete Shape . . . let's look at the details.



Example in DesignPatterns/Visitor

The double-dispatch

The Visitor pattern implements a double-dispatch technique since the action taken by the visitor depends on two class types: the one of the visitor and that of the element we are visiting. C++ does not support multiple dispatching natively, this is why we need to implement it using special techniques.
To make things clear, suppose we tried to implement the same procedure by creating a template function

```
 template <class Iterator, class Visitor>
 void visit(Iterator begin,
            Iterator end, Visitor & v){
  for (Iterator i=begin,i!=end;++i) v.visit(*i);
}
```

This template function operates on a range that we “visit” using an object which implements a `visit()` method.

We are not double-dispatching!!

This could be fine in may cases (indeed many algorithms of the Standard Library operate this way), but it is not a Visitor Pattern!.
The reason is that visit() now cannot be overloaded over a set of polymorphic objects!
Let's explain it with an example. Suppose that our sequence contains Shape $*$ that are in fact pointing to objects of the Shapes hierarchy. Let's suppose also that our Visitor class has correctly defined visit(Triangle $*$), visit(Square $*$) and so on. *(We are using here pointers as arguments of* visit()*, but the same issue arises if we use references instead.*
Let's compare what happens with the Visitor Pattern proper and with our attempt to replicate it with a template function.

With the Visitor pattern

Each element calls the accept(Visitor& v) method, which in turns would calls v. visit (**this**). Since accept() is a method of a concrete class of the Shape hierarchy the compiler can deduce the actual type of **this** and thus dispatch visit () to the correct function: if **this** is a Square $*$ then visit (Square $*$) will be called, and so on.

With the template function

visit ($*$i) has now signature **void** visit (Shape $*$) and there is no way that the compiler can statically detect to which concrete type Shape $*$ is pointing. So no overloading is performed. Here, the only way to implement the semantic of the Visitor pattern is to use a inside visitor (Shape $*$ p) an if statement with **dynamic_cast**<> (inefficient!).

The templated function implementation

```
void CountShapes::visit(Shape * s){
if (dynamic_cast<Triangle *>(s)!=0)++numTriangles;
else if (dynamic_cast<Square *>(s)!=0)++numSquares;
else if (dynamic_cast<Point *>(s)!=0)++numPoints;
}
```

This solution is less elegant, less scalable, and less efficient than the Visitor Pattern proper.

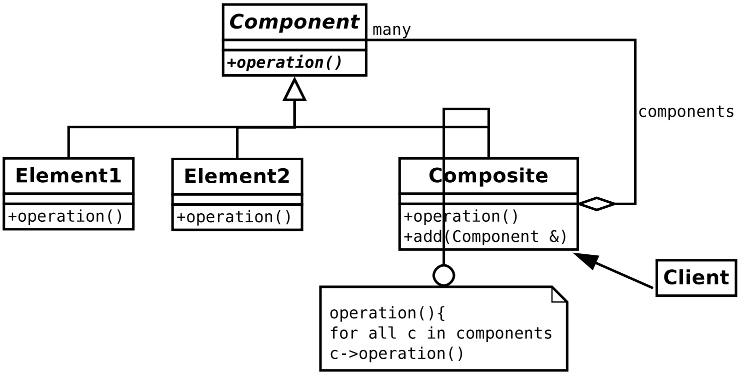
Note: In the Visitor Pattern we could have implemented different visiting methods for each concrete Shape: visitTriangle(), visitSquare(), etc. This is the way the pattern is explained in the GoF book. Since this is a C++ course we have preferred to use overloading: it is more C++ style.

A warning

Visitor is a very powerful, yet controversial pattern. It applies best to a well defined and stable structure. If we want to extend the elements on which it operates (add a new shape, for instance), we need to change all the visitors!
Moreover it is prone to errors: if we derive a EquilateralTriangle from Triangle how can we make sure that the visitor will launch the correct visit()? We will need to use a different name for the function (which is what is often done in many implementations).
Alexandrescu in his book describes this and other pitfalls of the Visitor pattern and proposes a generic (templated) Visitor that addresses them. It is a very nice piece of template metaprogramming. Its description, however, goes well beyond the scope of this notes.

Composite Pattern

Often an application needs to build a tree-like structure of objects that are the components of a complex system, and to do operations recursively on the component of the structure. For instance, a document is composed by pages, themselves composed by paragraphs, graphical objects, text boxes, themselves possibly composed by paragraphs and other graphical object and so on. The Composite Pattern presents a general framework to build such structure, where we derive from a common abstract `Component` class some `Elements` among which one is a `Composite` element which may aggregate other `Components`. We have thus a recursive structure.


Component
+operation()
many
components
Element1
+operation()
Element2
+operation()
Composite
+operation()
+add(Component &)
Client
operation(){
for all c in components
c->operation()

An example

An example in DesignPatterns/Composite