# C++ design patterns

P. Ronchese
Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

## Design patterns

A "design pattern" is
a common solution for a common problem

They became popular due to a book by
Erich Gamma, Richard Helm, Ralph Johnson
and John Vlissides, known as "the gang of four",
who identified and described 23 classic patterns.

- A design pattern is not an algorithm.
- A design pattern is a repeatable solution or approach to a design problem.
- Design patterns can be classified in 3 main categories.

## Pattern classification

> Design patterns are classified depending on the design problem they address

- Creational patterns.
- Structural patterns.
- Behavioral patterns.

# Creational patterns

**Creational patterns deal with object creation**

- Singleton
- (Abstract) Factory
- Builder
- Prototype

## Singleton

A "Singleton" is a `class` that
can be instantiated only once and
whose instance is globally accessible.

- It's (ab)used for those objects containing very general information:
  - running directives and/or conditions,
  - data common properties,
  - common operations.
- This can be achieved by:
  - declaring the constructor `private` ,
  - providing a `static` function creating only one instance of the `class` and returning the pointer to it.
- The object is actually created at its first use.
- It can be implemented in a `template` .

## Singleton implementation

```
class ObjS {
 public:
  static ObjS* instance();
  ...
 private:
  ObjS();
  ~ObjS();
  ObjS(const ObjS& a);
  ObjS& operator=(const ObjS& a);
};
```

```
ObjS::ObjS() {...}
ObjS::~ObjS() {...}
```

Special care must be taken to avoid:

- creating and using the object before proper initialization,
- using the object after its destruction.

## Singleton creation and destruction

The singleton can be created as a `static` object:

- it can be used in other `static` objects constructors,
- it cannot be used in other `static` objects destructors.

```
ObjS* ObjS::instance() {
  static ObjS obj;
  return &obj;
}
```

The singleton can be created dynamically keeping a `static` pointer to it:

- it can be used in other `static` objects destructors,
- the singleton own destructor is never run.

```
ObjS* ObjS::instance() {
  static ObjS* obj=new ObjS;
  return obj;
}
```

## Base and derived objects

An analogous pattern can be used with base object, with the function `instance()` returning a reference-to-pointer initially set at null, and the constructor setting that pointer to `this` :

- the client code can use only an interface,
- the actual object type can be chosen elsewhere.

```
class Base {
 public:
  static Base*& instance();
  ...
 protected:
  Base();
  ...
};
```

# Pointer saving

```
Base::Base() {
  Base*& i=instance();
  if(i==0)i=this;
}
```

```
Base*& Base::instance() {
  static Base* obj=0;
  return obj;
}
```

- The pointer `obj` is initially set at null.
- When an object is created and `obj` is null, its pointer is stored in `obj` .
- `obj` points to the first object that has been created.
- Any following call to `instance()` will return a pointer to the first object that has been created.

## Derived object creation

```
class Derived: public Base {
 public:
  Derived() {...}
  ...
};
static Derived d;
```

- When a `Derived` object is created, a `Base` object is also created.
- If a `Derived` object is created before any other object derived from `Base` , its pointer is saved.
- Any following call to `instance()` will return a pointer to a `Derived` object.
- A declaration and definition of one global `Derived` object make any call to `instance()` to return a pointer to it.

# Factory

A "Factory" is a `class` that creates
an instance of another `class`
from a family of derived `class`es

- A "Factory" `class` has a (usually `static`) function returning a pointer to a base `class` .
- The actual object type depend on the arguments and/or other informations the function can obtain.
- The client does not depend on the derived `class`es.
- The client need not knowing all the informations needed to create the objects.

```
class ShapeFactory {
 public:
  static Shape* create(...);
  ...
};
```

## Abstract factory

An “Abstract Factory” is an interface to a concrete Factory, so that the objects actually created depend on the actual factory object

- The function to create objects is declared `virtual` .
- It is (re)implemented in all concrete factories.
- Example:
  - A generic `CarFactory` has `small` , `medium` and `large` functions to create `Car`s.
  - `Volkswagen` is a `CarFactory`, and implements its create functions to return `Polo` , `Golf` or `Passat` .
  - `Ford` is a `CarFactory`, and implements its create functions to return `Fiesta` , `Focus` or `Mondeo` .

# Builder

A "Builder" is a `class` that creates complex objects step by step

- Used to encapsulate the operations needed to create a complex object.
- A Builder has functions to specify how the objects is to be built, and a function to actually create the result.
- A Builder usually specify only an interface, while a Concrete Builder actually performs the operations.
- The client can create different objects by using similar operations.

## Prototype

A "Prototype" is a `class` that creates new objects by cloning an initial object

- The object to clone can be obtained from a manager.
- As with Factory, the client need not knowing all the concrete object types.
- The Prototype manager can allow the registration of new objects at runtime.
- New objects can be registered by loading new dynamic libraries at runtime by using the `dlopen` function.

```
#include <dlfcn.h>
  ...
  void* p=dlopen("libName.so",RTLD_LAZY);
  ...
```

## Structural patterns

Structural patterns deal with object relations

- Proxy
- Flyweight
- ...and others

## Proxy

A "Proxy" is an object to be used in place of another one

- The use of a proxy can be convenient when:
  - an object cannot be duplicated,
  - an object is too expensive to create or duplicate,
  - the object creation can/must be postponed until its really needed,
  - any operation can be done in the access to an object to make the access less expensive.
- A proxy provides a modified (usually simpler) interface to an, usually complex, object.
- A proxy contains a pointer or reference to the object it represents.

## Flyweight

A “Flyweight” is a shared object that can be used in multiple contexts

- Several objects can exist, with:
  - large part of them being identical,
  - only a small part different.
- A save in memory (and sometimes computing time) can be obtained sharing the common parts.
- Fine granularity in object design.

## Behavioral patterns

Behavioral patterns deal with object doing operations

- Observer
- Visitor
- Iterator
- ...and others

## Observer

An "Observer" is an object doing an operation each time the state of some other object changes

- An observer is "updated" when the objects it depends on changes.
- Many objects may depend on the same object:
  - a standard way of subscribing to listening for events is needed,
  - a standard way of notifying dependent objects is needed,
  - an automatic subscribing can be implemented.
- Dispatcher/Observer `class` pairs can be implemented in `template`s.

## Observer interface and implementation

```
class Object;
class Observer {
 public:
  Observer();
  virtual ~Observer();
  virtual void update(const Object& x)=0;
};
```

```
...
Observer::Observer() {
  Dispatcher::subscribe(this);
}
Observer::~Observer() {
  Dispatcher::unsubscribe(this);
}
```

The observer registers/unregisters itself in the constructor/destructor

## Dispatcher interface

```
class Observer;
class Object;
class Dispatcher {
 public:
  static void   subscribe(Observer* obs);
  static void unsubscribe(Observer* obs);
  static void notify(const Object& x);
 private:
  Dispatcher()  { };
  ~Dispatcher() { };
  static std::set<Observer*>* obsL();
};
```

## Dispatcher implementation

```
static std::set<Observer*>* obsL() {
  static std::set<Observer*>* ptr=
     new std::set<Observer*>;
  return ptr;
}
```

An observer can be created and registered at any time:

- the container must be created when needed first,
- a single instance can be provided by a `static` function.

```
void Dispatcher::subscribe(Observer* obs) {
  obsL()->insert(obs);
}
```

```
void Dispatcher::unsubscribe(Observer* obs) {
  obsL()->erase(obs);
}
```

## Notification

```
void Dispatcher::notify(const Object& x) {
  std::set<Observer*>::iterator
                      it=obsL()->begin();
  std::set<Observer*>::iterator
                      ie=obsL()->end();
  while(it!=ie) (*it++)->update(x);
  return;
}
```

- The dispatcher notifies all the registered observers by calling their `update` function.
- Any concrete observer must override the `update` function.

## Operations “on demand”

An “Observer” can perform its operation:

- for each event (“active observer”), as previously described,
- only if needed (“lazy observer”).

- A “lazy observer” when notified does not “update” itself:
  - it simply sets its state as “not updated”,
  - if requested to “check” its state it actually “updates”, if necessary,
  - it can cache any result and return it without recomputing, unless necessary.
- Such a mechanism allow performing any operation only when needed:
  - an operation is not done if not needed,
  - operations are done at most once,
  - operations are automatically performed in the correct order.

## Lazy Observer interface

```
class Object;
class LazyObserver {
 public:
  LazyObserver();
  virtual ~LazyObserver();
  virtual void lazyUpdate(const Object& x);
 protected:
  virtual void update(const Object& x)=0;
  virtual void check();
 private:
  bool upToDate;
  bool updating;
  const Object* last;
};
```

## Lazy Observer implementation

- The dispatcher actually calls the function `lazyUpdate` .
- The function `update` is called by the function `check` ,

```
...
LazyObserver::lazyUpdate(const Object& x) {
  upToDate=updating=false;
  last=&x;
}
LazyObserver::check() {
  if(updating)return; // check for recursion
  updating=true;
  if(!upToDate)update(*last);
  upToDate=true;
  updating=false;
  return;
}
```

## Active/lazy notification

```
void Dispatcher::notify(const Object& x) {
  std::set<Observer*>::iterator
                          lit=lOL()->begin();
  std::set<Observer*>::iterator
                          lie=lOL()->end();
  while(lit!=lie) (*lit++)->lazyUpdate(x);
  std::set<Observer*>::iterator
                          ait=aOL()->begin();
  std::set<Observer*>::iterator
                          aie=aOL()->end();
  while(ait!=aie) (*ait++)->update(x);
  return;
}
```

- The dispatcher notifies the lazy observers first.
- The active observers can use results from lazy ones.

# Visitor

A "Visitor" is an object doing an operation on another object, where both the object and the operation belong to two families each sharing the same interface

- Example: several operations (area, perimeter...) can be performed on several geometric shapes (triangle, square...).
- The object used for the operation "accepts" the visitor.
- A generic visitor can be sent to a generic acceptor object, without knowing the exact derived type.
- The acceptor sends back a pointer to itself, making the visitor recognizing its type.
- In the whole process, two `virtual` function calls occur: the technique is called "double dispatch".

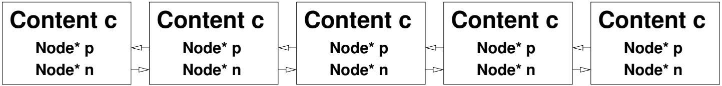
# Iterator

An “Iterator” is an object able to navigate across the objects in a collection (e.g. an array)

- An iterator:
  - can be de-referenced as a pointer,
  - encapsulates the operations needed to navigate through the objects.
- A lot of implementations are available in the STL.

# Linked list

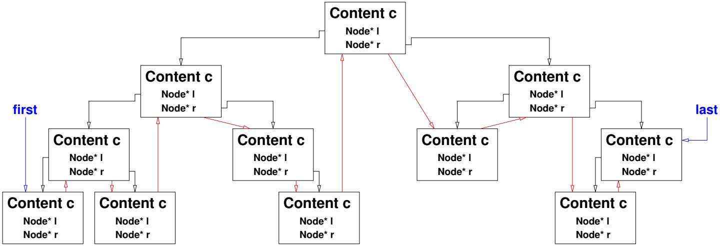
A "linked list" is a collection of objects contained in "nodes" where each node is linked to its neighbours through a pointer




- The navigation through the list requires the handling of the pointers.
- An "iterator" object can deal with that and let the client code to be simpler.

## Recursive tree

A “recursive tree” is a collection of objects contained in “nodes” where each node can be linked to a left and a right node through a pointer




- All contents of “left branch” and “right branch” are less or greater, respectively, than the content of the parent node.
- The navigation can be a bit complicated.