Design Patterns in C++

Object Oriented Design Principles

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

March 13, 2011

An object provides services

- You should think of an object as a service provider
- the goal of the programmer is to produce (or find in existing libraries) a set of objects that provide the right services that you need to solve the problem
    - To do this, you need to decompose the problem space into a set of objects
    - it does not matter if you do not know yet how to implement them
    - what it is important is to a) identify what are the "important" objects that are present in your problem and b) identify which services these objects can provide
    - Then, for every object, you should think if it is possible to decompose it into a set of simpler objects
    - You should stop when you find that the objects are *small enough* that can be easily implements and are self-consistent and self-contained

High cohesion

- Thinking of an object as a service provider has an additional benefit: it helps to improve the cohesiveness of the object.
- High cohesion is a fundamental quality of software design
  - It means that the various aspects of a software component (such as an object, although this could also apply to a method or a library of objects) “fit together” well
- One problem people have when designing objects is cramming too much functionality into one object
  - Treating objects as service providers is a great simplifying tool, and it’s very useful not only during the design process, but also when someone else is trying to understand your code or reuse an object
  - if they can see the value of the object based on what service it provides, it makes it much easier to fit it into the design.

Hiding the implementation

- Even when you are writing a program all by yourself, it is useful to break the playing field into class creators and client programmers
  - the class creators implements the internals of a class, so that it can provide services
  - the client programmers use the class to realize some other behavior (e.g. another class)
  - almost all programmers are at the same time class creators and client programmers
- The class creators must not expose the implementation details to the client programmers
  - The goal is to export only the details that are strictly useful to provide the services

Why hiding?

- The concept of implementation hiding cannot be overemphasized
- Why it is so important?
  - The first reason for access control is to keep client programmers' hands off portions they shouldn't touch
  - This is actually a service to users because they can easily see what's important to them and what they can ignore
  - The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer
  - For example, you might implement a particular class in a simple fashion to ease development, and then later discover that you need to rewrite it in order to make it run faster
  - If the interface and implementation are clearly separated and protected, you can accomplish this easily

Hiding implementation in C++

- In C++, not all implementation details are "hidden"
  - Class declaration in the include file contains private members
  - Include files are distributed along with lib files
  - Those are visible, not *hidden*! Customers can view the private part
- *Hidden* does not mean *secret*
  - It only means that they are not part of the interface
  - Thus, modifications to the private part do not imply modification to the client code, because the interface does not change
  - Code modification/adaptation is expensive, and a potential source of bugs

The pimpl idiom

- Changing the private part implies re-compiling all client files
  - Not so expensive, but annoying
- Also, sometimes we want to make all implementation *secret*
- To do this, we can use the “pimpl idiom”

```
// include file
class MyClassImpl;

class MyClass {
  MyClassImpl *pimpl;
public:
  //interface
};

// cpp source file
// definition of private part
class MyClassImpl {...}

MyClass::MyClass() {
  pimpl = new MyClassImpl();
  ...
}
```

pimpl performance

- All private data is in class `MyClassImpl`, which is not declared to the client
- the drawback is one more level of indirection: all private data must be accessed through a pointer, or redirected to an internal class
- this causes a slight increment in overhead
- another performance loss could be the call to new and `delete` operators every time an object is constructed

SOLID

- SOLID denotes the five principles of *good* object oriented programming
  - Single responsibility
  - Open-closed
  - Liskov substitution
  - Interface segregation
  - Dependency inversion
- it is a mnemonic acronym introduced by Robert C. Martin in the early 2000s which stands for five basic patterns of object-oriented programming and design.
- The principles when applied together make it much more likely that a programmer will create a system that is easy to maintain and extend over time.

Single Responsibility Principle

A class should have only one reason to change.

- In this context a responsibility is considered to be one reason to change.
- This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes.
  - Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it.
  - When we need to make a change in a class having more responsibilities, the change might affect the other functionality of the classes.
- Single Responsibility Principle was introduced Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

Example

- An object that represents an email message

```
class IEmail {
public:
    virtual void setSender(string sender) = 0;
    virtual void setReceiver(string receiver) = 0;
    virtual void setContent(string content) = 0;
};

class Email : public IEmail {
public:
    void setSender(string sender) {// set sender; }
    void setReceiver(string receiver) {// set receiver; }
    void setContent(string content) {// set content; }
};
```

Reasons top change

- our `IEmail` interface and `Email` class have 2 responsibilities (reasons to change).
  - One would be the use of the class in some email protocols such as pop3 or imap. If other protocols must be supported the objects should be serialized in another manner and code should be added to support new protocols.
  - Another one would be for the Content field. Even if content is a string maybe we want in the future to support HTML or other formats.
- We can create a new interface and class called `IContent` and `Content` to split the responsibilities.
- Having only one responsibility for each class give us a more flexible design:
  - adding a new protocol causes changes only in the Email class.
  - adding a new type of content supported causes changes only in Content class

Separate responsibilities

```
// single responsibility principle - good example
class IEmail {
public:
    virtual void setSender(string sender) = 0;
    virtual void setReceiver(string receiver) = 0;
    virtual void setContent(IContent content) = 0;
};

class IContent {
public:
    virtual string getAsString() = 0; // used for serialization
};

class Email : public IEmail {
public:
    void setSender(string sender) {// set sender; }
    void setReceiver(string receiver) {// set receiver; }
    void setContent(IContent content) {// set content; }
};
```

Designing for change

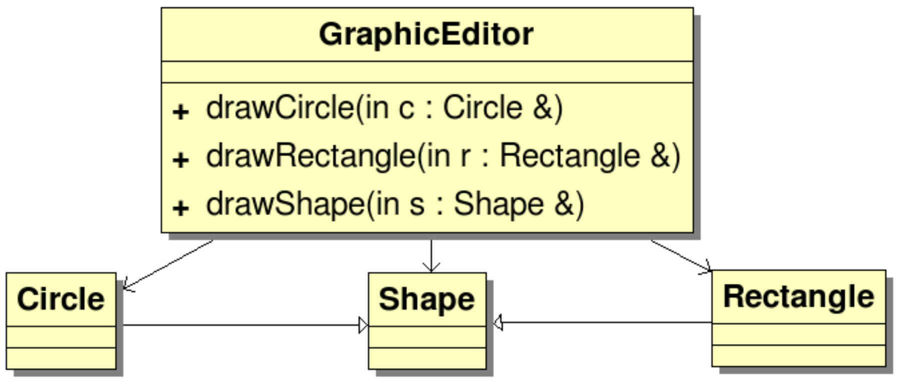
- Every software is subject to change
  - A good design makes changes less trouble
- Problems related to change:
  - The immediate cause of the degradation of the design is when requirements change in ways that the initial design did not anticipate
  - Often these changes need to be made quickly, and may be made by engineers who are not familiar with the original design philosophy
  - So, though the change to the design works, it somehow violates the original design. Bit by bit, as the changes continue to pour in, these violations accumulate until malignancy sets in
- The requirements document is the most volatile document in the project
  - If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault

The open/close principle

A class should be open for extension, but closed for modification

- Bertrand Meyer [4]
- In an ideal world, you should never need to change existing code or classes
  - Except for bug-fixing and maintenance
- all new functionality should be added by adding new subclasses and overriding methods, or by reusing existing code through **delegation**

A bad example (UML)


GraphicEditor
+ drawCircle(in c : Circle &)
+ drawRectangle(in r : Rectangle &)
+ drawShape(in s : Shape &)
Circle
Shape
Rectangle

The code

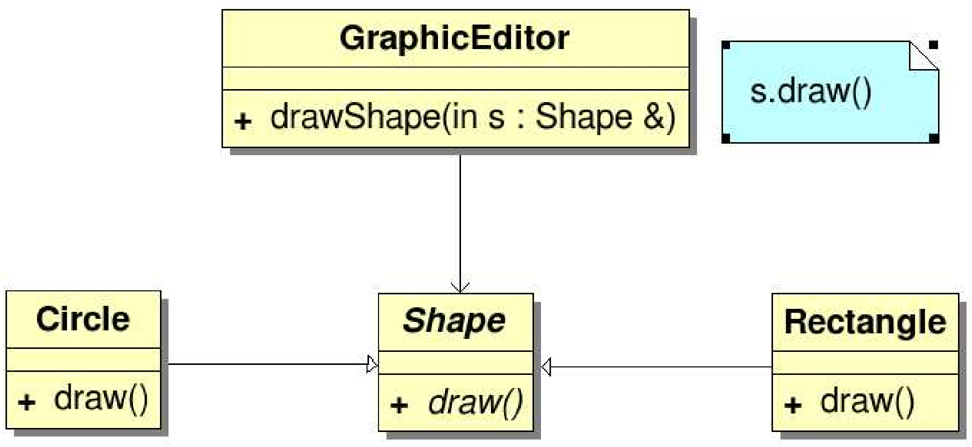
```
class GraphicEditor {
public:
    void drawShape(Shape &s) {
        if (s.type==1) drawRectangle(s);
        else if (s.type==2) drawCircle(s);
    }
    void drawCircle(Circle &r) {....}
    void drawRectangle(Rectangle &r) {....}
 };

 class Shape {
 public:
    int type;
 };

 class Rectangle : public Shape {
    Rectangle() { type=1; }
 };

 class Circle : public Shape {
    Circle() { type=2; }
 };
```

Solution (UML)



Design for change

- The Open-Closed principle
- Key issue: *prepare for change*
- Causes for re-design
  - Dependence on hardware or software platform
  - Dependence on representation or implementation
  - Algorithmic dependence
  - Tight coupling
  - Overuse of inheritance
  - Inability to alter classes easily

Liskov Substitution Principle

Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it.

Barbara Liskov, "Data Abstraction and Hierarchy," [3]

- The importance of this principle becomes obvious when you consider the consequences of violating it.
- If there is a function which does not conform to the LSP, then that function uses a pointer or reference to a base class, but must know about all the derivatives of that base class.

Example of violations of LSP

- One of the most glaring violations of this principle is the use of C++ Run-Time Type Information (RTTI) to select a function based upon the type of an object.

```
void DrawShape(const Shape& s)
{
  Square *q;
  Circle *c;

  if (q = dynamic_cast<Square *>(s))
    DrawSquare(q);
  else if (c = dynamic_cast<Circle *>(s))
    DrawCircle(c);
}
```

- Clearly the DrawShape function is badly formed. It must know about every possible derivative of the Shape class, and it must be changed whenever new derivatives of Shape are created. Indeed, many view the structure of this function as anathema to Object Oriented Design.

Other examples of violation

- There are other, far more subtle, ways of violating the LSP

```
class Rectangle
{
  public:
    void   SetWidth(double w) {itsWidth=w;}
    void   SetHeight(double h) {itsHeight=w;}
    double GetHeight() const    {return itsHeight;}
    double GetWidth() const     {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};
```

- Now suppose we want to introduce a Square
  - A square is a particular case of a rectangle, so it seems natural to derive class Square from class rectangle
  - Do you see problems with this reasoning?

Problems?

- Square will inherit the SetWidth and SetHeight functions.
- These functions are utterly inappropriate for a Square!
  - since the width and height of a square are identical.
- This should be a significant clue that there is a problem with the design.
- Suppose we write the code so that when we set the height the with changes as well, and viceversa.
- We have to do the Rectangle members virtual, otherwise it does not work!

Fixing it – the code

```
class Rectangle
{
  public:
    virtual void SetWidth(double w)  {itsWidth=w;}
    virtual void SetHeight(double h) {itsHeight=h;}
    double       GetHeight() const   {return itsHeight;}
    double       GetWidth() const    {return itsWidth;}
  private:
    double itsHeight;
    double itsWidth;
};
```

```
class Square : public Rectangle
{
  public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};
```

```
void Square::SetWidth(double w)
{
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

The real problem

- We changed the interface, not only the behavior!

```
void g(Rectangle& r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

- The code above was written by a programmer that did not know about squares
- what happens if you pass it a pointer to a Square object?
  - the programmer made the (at that time correct) assumption that modifying the height does not change the width of a rectangle.

Good design is not obvious

- The previous design violates the LSP
- A Square is not the same as a Rectangle for some pieces of code
  - from the behavioural point of view, they are not equivalent (one cannot be used in place of the other)
  - The *behaviour* is what is important in software!
  - See the paper

Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they don't use.

- This means that when we write our interfaces we should take care to add only methods that should be there.
- If we add methods that should not be there the classes implementing the interface will have to implement those methods as well.
    - For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?
- avoid polluted or fat interfaces

Bad example

```
class IWorker {
public:
    virtual void work() = 0;
    virtual void eat() = 0;
};
class Worker : public IWorker {
public:
    void work() { ... }
    void eat() { ... }
};
class SuperWorker : public IWorker {
public:
    void work() { ... }
    void eat() { ... }
};
class Manager {
    IWorker *worker;
    public void setWorker(IWorker *w) { worker=w; }
    public void manage() { worker->work(); }
};
```

Pollution

- The `IWorker` interface is polluted, as `Manager` does not use function `eat()`
- Suppose a robot is bought that can `work()` but does not need to eat at lunch break
- it could implement interface `IWorker`, but returning an exception of error code for function `eat()`
- This is bad design, because the interface is doing too much
- The solution is to separate the interfaces for `work()` and `eat()`

Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

- In an application we have low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and rely on the low level classes.
- A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes.
- But this is not a flexible design. What happens if we need to replace a low level class?

Dependency problem

- Let's take the classical example of a copy module which read characters from keyboard and write them to the printer device.
- The high level class containing the logic is the `Copy` class. The low level classes are `KeyboardReader` and `PrinterWriter`.
- In a bad design the high level class uses directly the low level classes. In this case if we want to change the design to direct the output to a new `FileWriter` class we have to change the `Copy` class.
- Since the high level modules contains the complex logic they should not depend on the low level modules
- a new abstraction layer should be created to decouple the two levels

The solution

- According to the Dependency Inversion Principle, the way of designing a class structure is to start from high level modules to the low level modules:
- High Level Classes $\Rightarrow$ Abstraction Layer $\Rightarrow$ Low Level Classes
  1. Write interfaces to low level modules (abstract layer)
  2. Make sure the high level classes use only references to the abstract interfaces
  3. Use some creational pattern to make the connection (i.e. insert the reference to the right low level class into the high level class)

Motivation

- Good Object Oriented programming is not easy
  - Emphasis on design
- Errors may be expensive
  - Especially design errors!
- Need a lot of experience to improve the ability in OO design and programming
- Reuse experts' design
- Patterns $=$ documented experience

The source

- The design patterns idea was first proposed to the software community by the "Gang of four" [2]
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Design patterns: elements of reusable object-oriented software
- They were inspired by a book on architecture design by Christopher Alexander [1]

  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, without ever doing it the same way twice.

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design
    - One of the few examples in which software development has been inspired by other areas of engineering
- The expected benefits of applying well-know design structures
    - Finding the right code structure (which classes, their relationship)
    - Coded infrastructures
    - A Common design jargon (factory, delegation, composite, etc.)
    - Consistent format

Patterns and principles

- Patterns can be seen as extensive applications of the OO principles mentioned above
- For every patter we will try to highlight the benefits in terms of hiding, reuse, decoupling, substitution, etc.

Pattern Categories

- **Creational:** Replace explicit creation problems, prevent platform dependencies
- **Structural:** Handle unchangeable classes, lower coupling and offer alternatives to inheritance
- **Behavioral:** Hide implementation, hides algorithms, allows easy and dynamic configuration of objects

Bibliography

Cristopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdhal-King, and Shlomo Angel.
A pattern language.
Oxford University Press, 1997.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design patterns: elements of reusable object-oriented software.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Barbara Liskov.
Data abstraction and hierarchy.
SIGPLAN Notice, 23(5), 1988.

Bertrand Meyer.
Object-Oriented Software Construction.
Prentice Hall, 1988.