Object Oriented Software Design - I

Object Oriented Design Principles

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

Scuola Superiore Sant'Anna – Pisa

October 28, 2011

Outline

1. Principles of OO programming and design
2. Single Responsibility
3. Open/Close principle
4. Liskov's Substitution Principle
5. Interfaces
6. Dependency Inversion
7. Bibliography

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.

Outline

1. Principles of OO programming and design
2. Single Responsibility
3. Open/Close principle
4. Liskov's Substitution Principle
5. Interfaces
6. Dependency Inversion
7. Bibliography

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

```
interface IEmail {
    void setSender(String sender);
    void setReceiver(String receiver);
    void setContent(String content);
};

class Email implements IEmail {
    public void setSender(String sender)     { /* set sender;*/ }
    public void setReceiver(String receiver) { /* set receiver;*/ }
    public 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
interface IEmail {
    void setSender(string sender);
    void setReceiver(string receiver);
    void setContent(IContent content);
};

interface IContent {
    String getAsString(); // used for serialization
};

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

Outline

1. Principles of OO programming and design
2. Single Responsibility
3. Open/Close principle
4. Liskov's Substitution Principle
5. Interfaces
6. Dependency Inversion
7. Bibliography

Designing for change

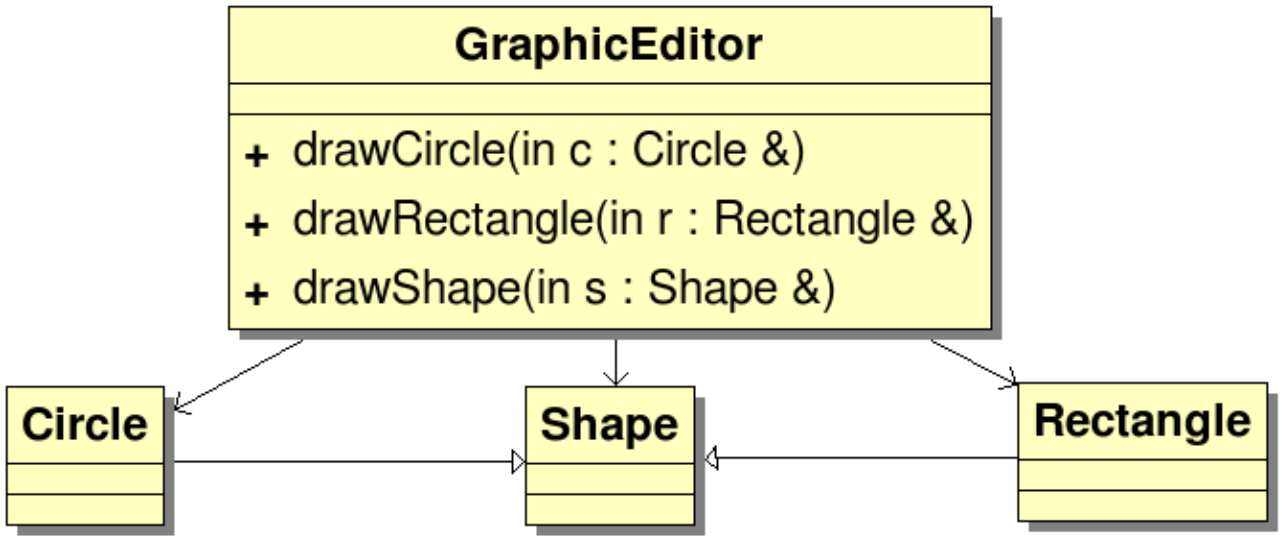
- Every software is subject to change
  - A good design makes changes less trouble
- Problems related to change:
  - The immediate cause of the degradation of the design is when requirements change in ways that the initial design did not anticipate
  - Often these changes need to be made quickly, and 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 [2]
- 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)



The code

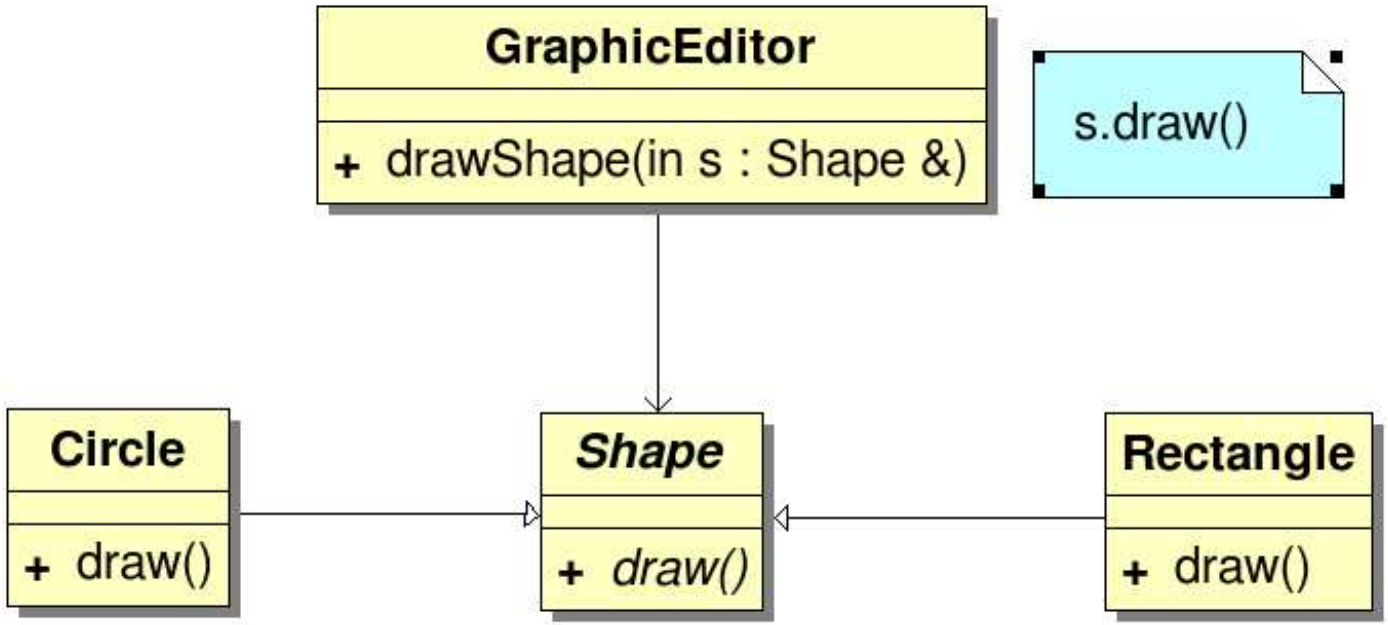
```
class GraphicEditor {
    public void drawShape(Shape s) {
        if (s.type==1) drawRectangle(s);
        else if (s.type==2) drawCircle(s);
    }
    public void drawCircle(Circle r) {....}
    public void drawRectangle(Rectangle r) {....}
 }

 class Shape {
    public int type;
 };

 class Rectangle extends Shape {
    Rectangle() { type=1; }
 };

 class Circle extends 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

Outline

1. Principles of OO programming and design
2. Single Responsibility
3. Open/Close principle
4. Liskov's Substitution Principle
5. Interfaces
6. Dependency Inversion
7. Bibliography

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,” [1]

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,” [1]

- 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.

```
public void DrawShape(Shape s)
{
  Square q;
  Circle c;

  if (s instanceof Square) {
    q = (Square) s;
    DrawSquare(q);
  }
  else if (s istanceof Circle) {
    c = (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

Other examples of violation

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

```
class Rectangle
{
    private double itsWidth;
    private double itsHeight;

    public void   SetWidth(double w) {itsWidth=w;}
    public void   SetHeight(double h) {itsHeight=w;}
    public double GetHeight() {return itsHeight;}
    public double GetWidth()  {return itsWidth;}
};
```

Other examples of violation

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

```
class Rectangle
{
    private double itsWidth;
    private double itsHeight;

    public void   SetWidth(double w) {itsWidth=w;}
    public void   SetHeight(double h) {itsHeight=w;}
    public double GetHeight() {return itsHeight;}
    public double GetWidth()  {return itsWidth;}
};
```

- 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.

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
{
    private double itsHeight;
    private double itsWidth;

    public void SetWidth(double w)  { itsWidth=w; }
    public void SetHeight(double h) { itsHeight=h; }
    public double GetHeight()       { return itsHeight; }
    public double GetWidth()        { return itsWidth; }
};
```

```
class Square extends Rectangle
{
    public void SetWidth(double w) {
        super.SetWidth(w);
        super.SetHeight(w);
    }
    public void SetHeight(double h) {
        super.SetWidth(h);
        super.SetHeight(h);
    }
};
```

The real problem

- We changed the interface, not only the behavior!

```
void g(Rectangle r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  if (r.GetWidth() * r.GetHeight()) != 20) {
      // ERROR!!
  }
}
```

- 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

Outline

1 Principles of OO programming and design

2 Single Responsibility

3 Open/Close principle

4 Liskov's Substitution Principle

5 Interfaces

6 Dependency Inversion

7 Bibliography

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

```
interface IWorker {
    void work();
    void eat();
};
class Worker implements IWorker {
    public void work() { ... }
    public void eat() { ... }
};
class SuperWorker implements IWorker {
    public void work() { ... }
    public 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()`

Outline

1. Principles of OO programming and design
2. Single Responsibility
3. Open/Close principle
4. Liskov's Substitution Principle
5. Interfaces
6. Dependency Inversion
7. Bibliography

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)

Outline

1. Principles of OO programming and design
2. Single Responsibility
3. Open/Close principle
4. Liskov's Substitution Principle
5. Interfaces
6. Dependency Inversion
7. Bibliography

Bibliography

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

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