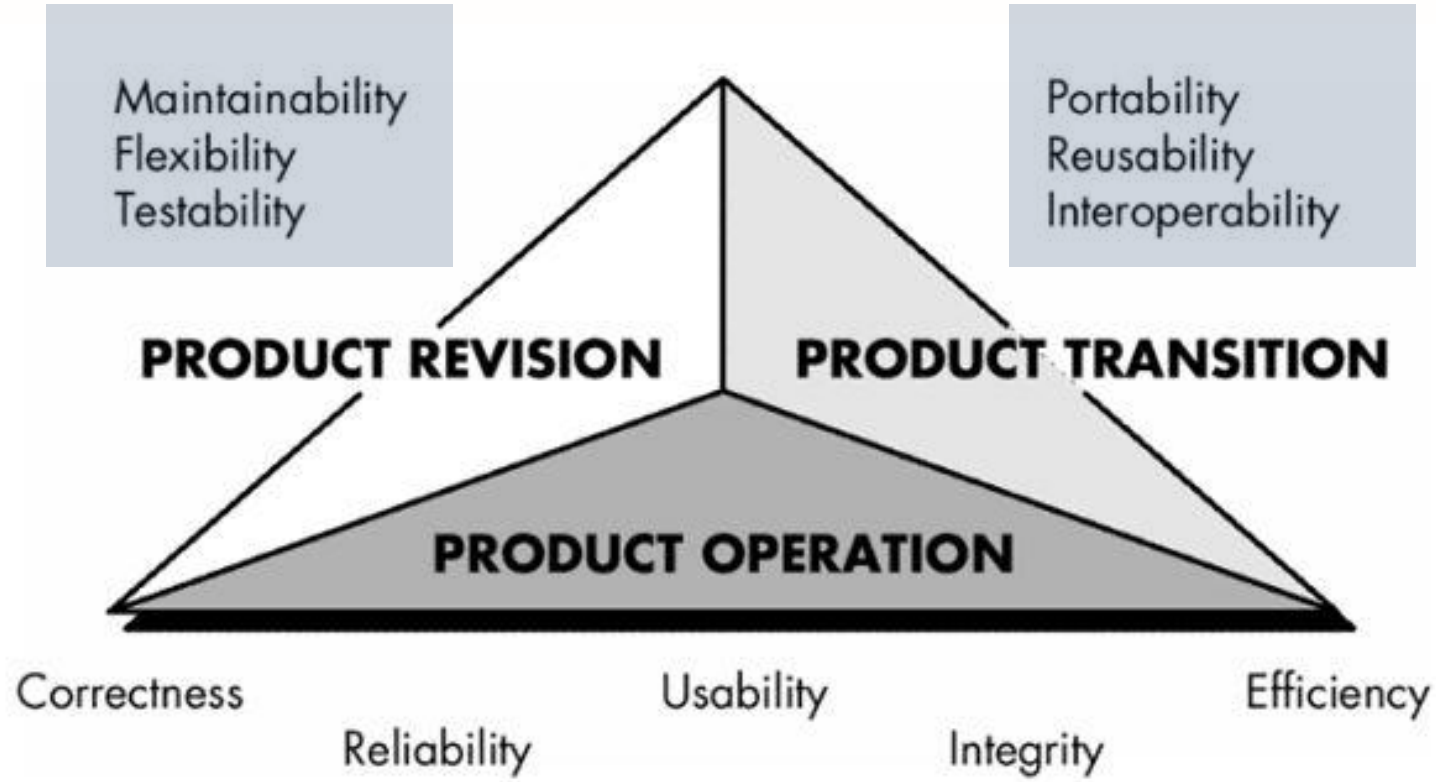
PEMROGRAMAN LANJUT

SOLID: OCP dan LSP

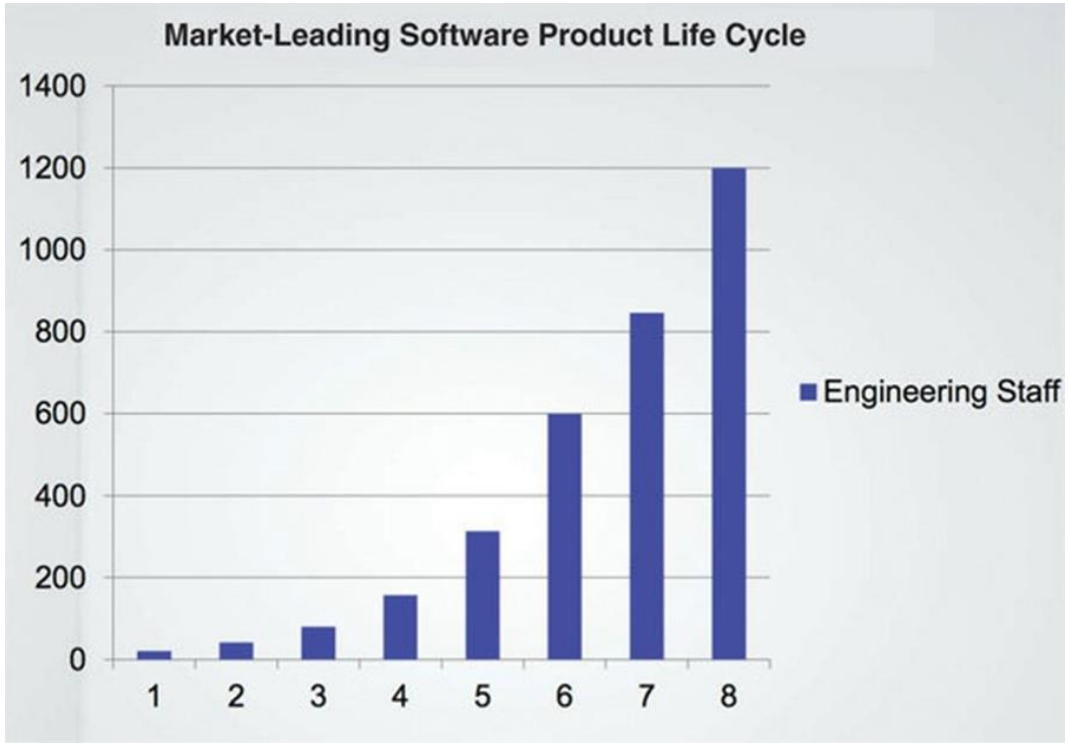
Oleh
Tri Hadiah Muliawati
Politeknik Elektronika Negeri Surabaya
2021

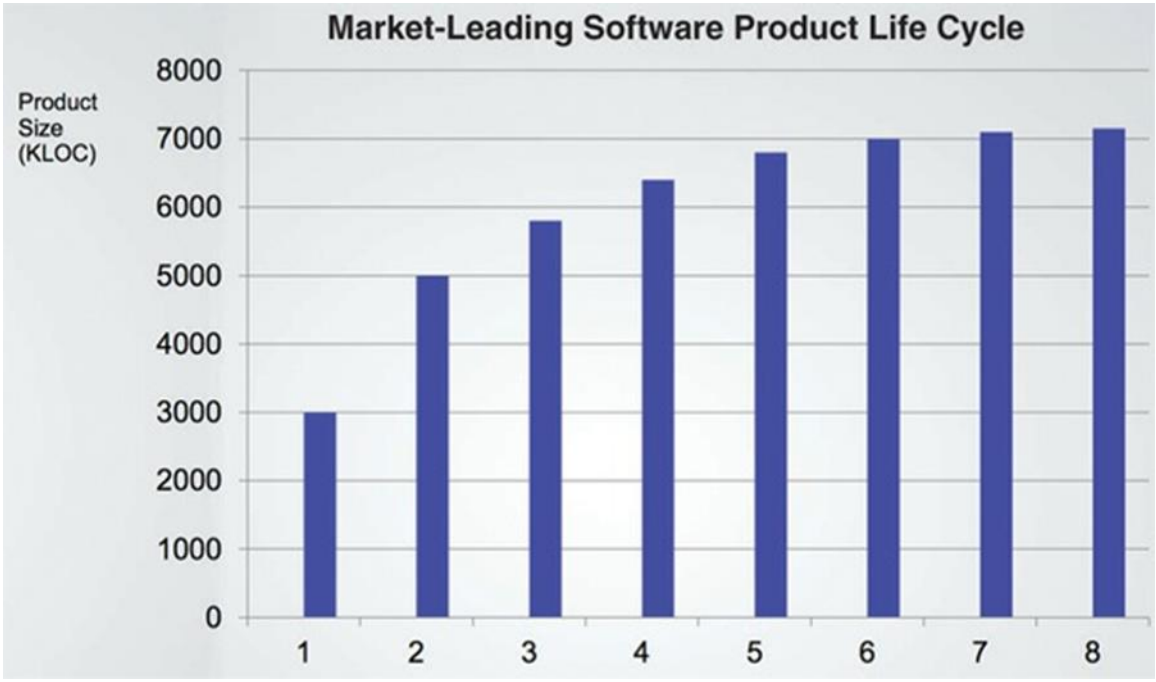
Politeknik Elektronika Negeri Surabaya
Departemen Teknik Informatika dan Komputer

Review



Mc Call Software Quality Metric





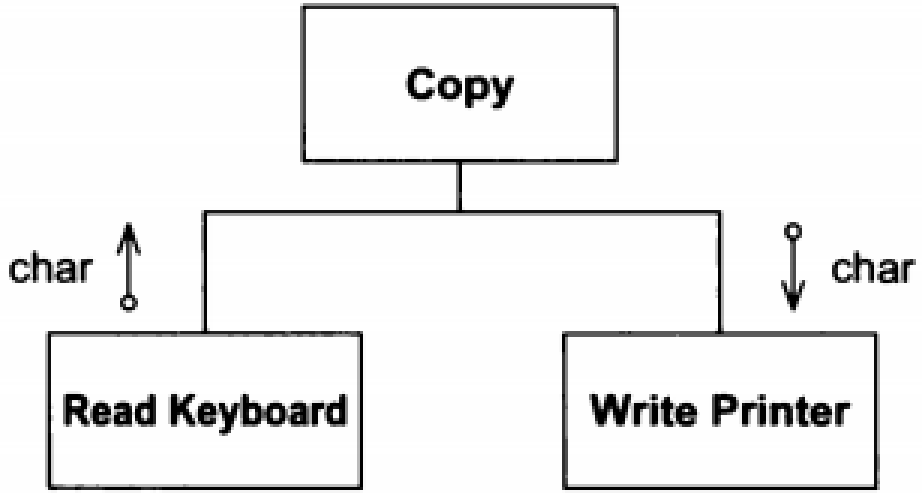
Cost/LOC is increasing per iteration

Symptoms of Poor Design – Designs Smells

1. Rigidity – hard to change
   The system is hard to change because every change forces many other changes to other parts of the system
2. Fragility – easy to break
   Changes cause the system to break in places that have no conceptual relationship to the part that was changed
3. Immobility – hard to reuse
   It is hard to disentangle the system into components that can be reused in other systems
4. Viscosity – hard to do the right thing
   Doing the things right is harder than doing the things wrong
5. Needless Complexity – overdesign
   The design contains infrastructure that adds no direct benefit
6. Needless Repetition – mouse abuse
   The design contains repeating structures that could be unified under a single abstraction
7. Opacity – disorganized expression
   It is hard to read and understand. It does not express its intent well

Case Study

- Aim:
  Program to copy characters from keyboard to printer



Initial Code

```
void Copy()
{
  int c;
  while ((c=RdKbd()) != EOF)
    WrtPrt(c);
}
```

Case Study

- 1st change of requirements:
  Copy program should be able to read from printer and paper tape reader

```
void Copy()
{
  int c;
  while ((c=RdKbd()) != EOF)
    WrtPrt(c);
}
```

```
bool ptFlag = false;
// remember to reset this flag
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(c);
}
```

RdPt() ➔ read from paper tape reader
RdKbd() ➔ read from keyboard
ptFlag ➔ flag to check whether input is paper tape reader

Case Study

WrtPrt() ➔ output is sent to printer
WrtPunch() ➔ output is sent to paper tape punch
punchFlag ➔ flag to check whether output is sent to paper tape punch

- 2st change of requirements:
  Copy program should be able to output to paper tape punch

```
bool ptFlag = false;
// remember to reset this flag
void Copy()

{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(c);
}
```

```
bool ptFlag = false;
bool punchFlag = false;
// remember to reset these flags
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    punchFlag ? WrtPunch(c) : WrtPrt(c);
}
```

The structure of program is beginning to topple.
Any more changes to the input device will certainly force developer to completely restructure `while`-loop conditional.

Requirements always change ☺

We (developers) live in the world of changing requirements, and our job is to make sure that our software can survive those changes.

-Robert C. Martin-

Case Study

```
bool ptFlag = false;
// remember to reset this flag
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(c);
}
```

```
class Reader
{
  public:
    virtual int read() = 0;
};

class KeyboardReader : public Reader
{
  public:
    virtual int read() {return RdKbd();}
};
```

```
KeyboardReader GdefaultReader;

void Copy(Reader& reader = GdefaultReader)
{
  int c;
  while ((c=reader.read()) != EOF)
    WrtPrt(c);
}
```

Instead of simply changing the code to accommodate the first change of requirements (**Copy program should be able to read from printer and paper tape reader**), developers can improve the design as well.

Therefore, the design can be more resilient towards similar changes in the future (additional type of reader used).

Principles of OO Design

- Single Responsibility Principle
- Open Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Principles of OO Design

- Those principles are not applied directly in up-front design. Rather, they are applied from iteration to iteration in an attempt to keep the code, and the design it embodies, clean.
- They don’t apply principles when there are no smells. It is a mistake to unconditionally conform to a principle just because it’s a principle.
- Principles are not perfume to be liberally scattered all over the system. Over conformance to the principles leads to the design smell of needless complexity.

Single Responsibility Principle (SRP)

- A module should have only one reason to change. It doesn't mean that every module should do just one thing.
- If a module has more than on responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the ability of the module to meet the others.

Case Study: Email

```
// single responsibility principle - bad example

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

- IEmail interface and Email class have 2 responsibilities (reasons to change):

1. 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.
2. Even if content is a string maybe we want in the future to support HTML or other formats.

Case Study: Email

```
// single responsibility principle - bad example

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

- If we keep only one class, each change for a responsibility might affect the other one:

1. Adding a new protocol will create the need to add code for parsing and serializing the content for each type of field.
2. Adding a new content type (like html) make us to add code for each protocol implemented.

Case Study: Email

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

interface Content {
        public 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; }
}
```

- Having only one responsibility for each class give us a more flexible design:

1. adding a new protocol causes changes only in the Email class.
2. adding a new type of content supported causes changes only in Content class.

End of Review

Principles of OO Design

- Single Responsibility Principle
- Open Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

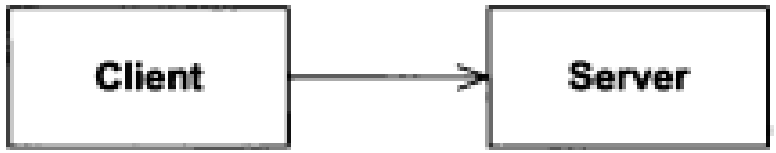
Open Close Principle

Open Close Principle

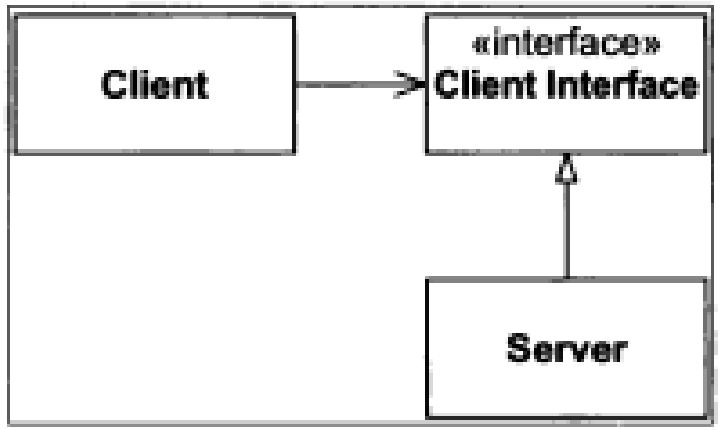
- Originated by Bertrand Meyer, 1988
- Software entities should be open for extension, but closed for modification
- A good software architecture would reduce the amount of changed code to the barest minimum. Ideally, zero.

Open Close Principle

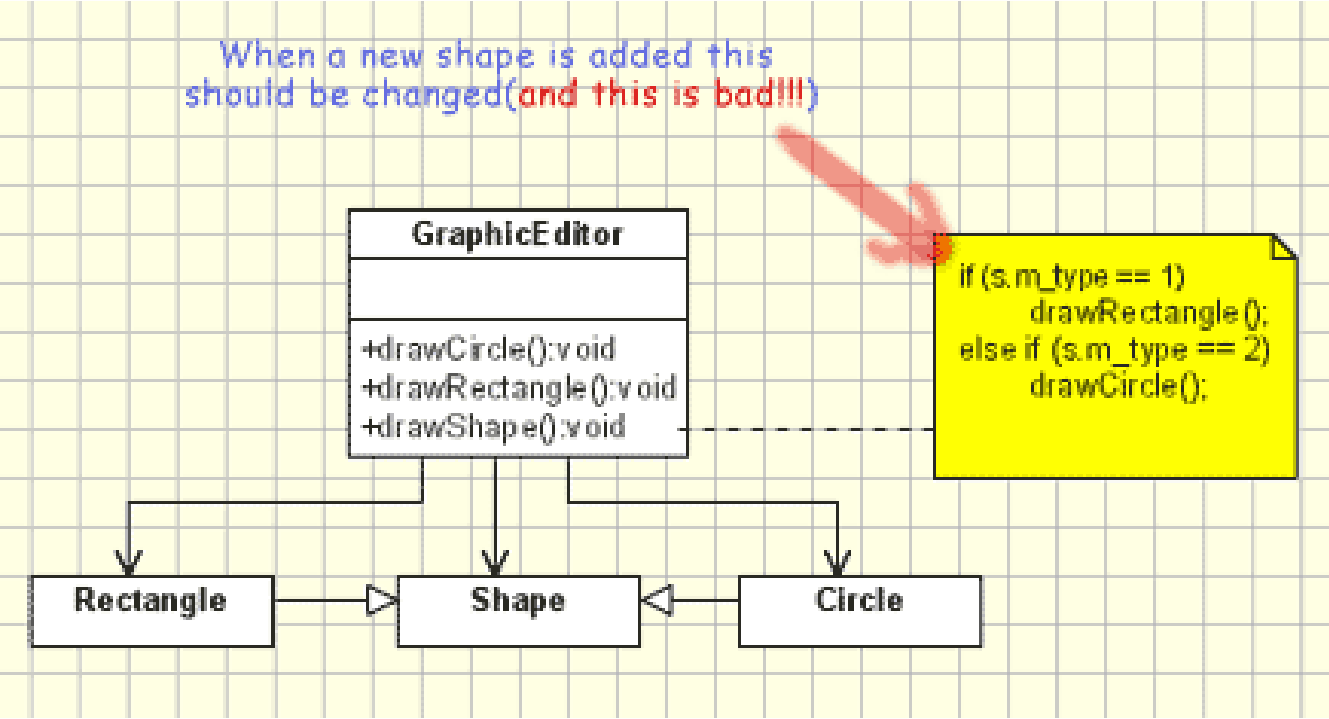
- It is implemented using abstraction



By using abstraction, **Client class is not tightly coupled with Server class**. Thus, if there is a new requirement to add another type of Server, developer can create new class as long as it implements Client Interface.



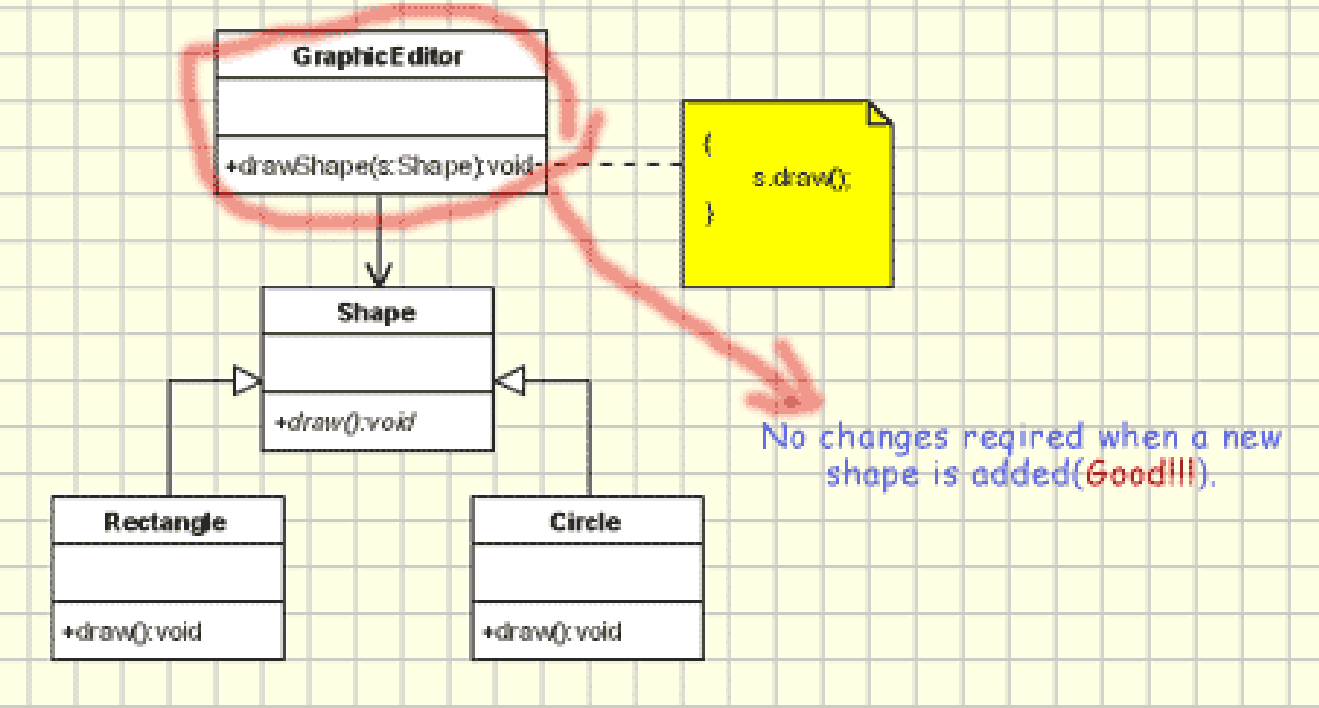
Open Close Principle



If there is new requirement to add another type of shape (i.e.: Triangle, Trapezium, Pentagon, Diamond, etc.), developer should modify drawShape() in GraphicEditor. **It is a violation of OCP**.

```
// Open-Close Principle - Bad example
 class GraphicEditor {

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

 class Shape {
        int m_type;
 }

 class Rectangle extends Shape {
        Rectangle() {
                super.m_type=1;
        }
 }

 class Circle extends Shape {
        Circle() {
                super.m_type=2;
        }
 }
```

Open Close Principle




```
// Open-Close Principle - Good example
 class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
 }

 class Shape {
        abstract void draw();
 }

 class Rectangle extends Shape  {
        public void draw() {
                // draw the rectangle
        }
 }
```

If there is new requirement to add another type of shape (i.e.: Triangle, Trapezium, Pentagon, Diamond, etc.), developer can easily create a new class which is an extension of Shape class. **It is an implementation of OCP**.

Open Close Principle

- Making a flexible design involves additional time and effort spent for it and it introduce new level of abstraction increasing the complexity of the code. So this principle should be applied in those area which are most likely to be changed.
- We can adapt “Fool Me Once” to keep loading our software with needless complexity. Thus, we initially write our code expecting it not to change. When change occurs, we implement the abstractions that protect us from future changes of that kind.

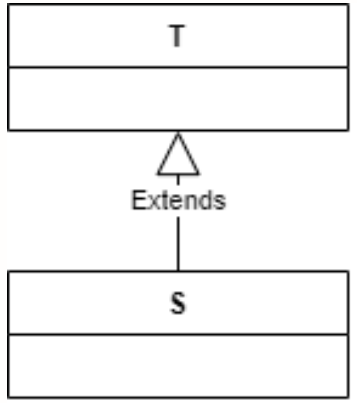
Liskov Substitution Principle

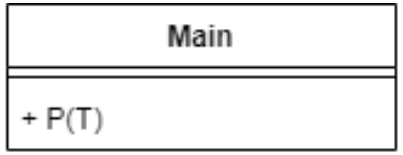
Liskov Substitution Principle

- Originated by Barbara Liskov, 1988
- Subtypes must be substitutable for their base types.
- We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

Liskov Substitution Principle

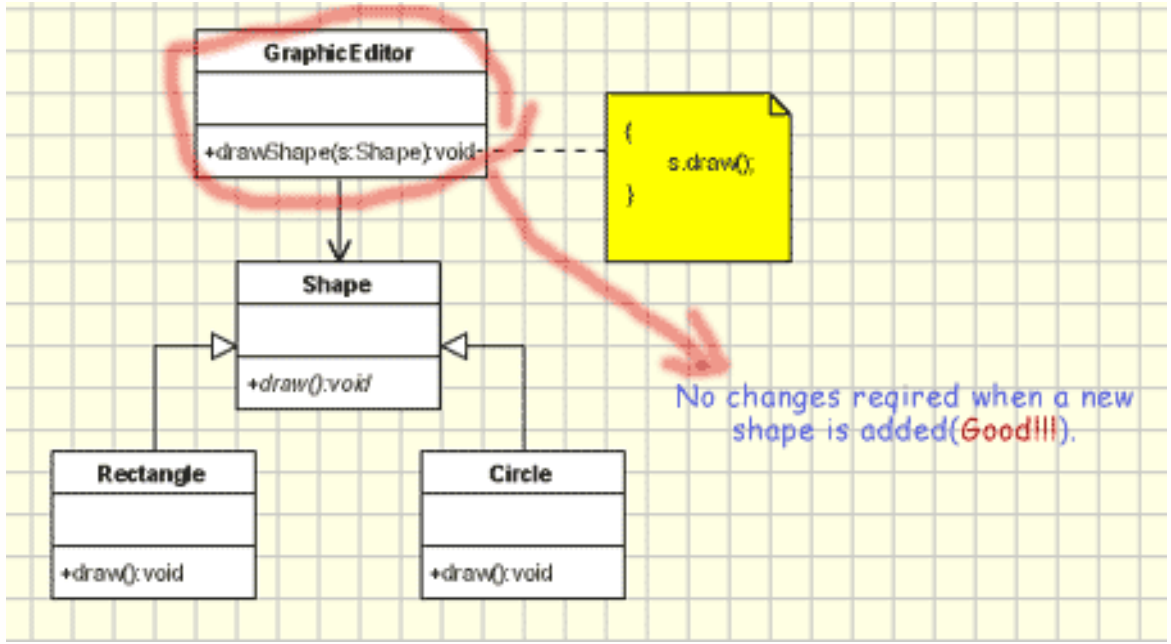
- If for each object `O1` of type `S`, there is an object `O2` of type `T`. Such that for all programs `P` defined in terms of the `T`, the behavior of `P` is unchanged when `O1` is substituted for `O2` when `S` is a subtype of `T`.

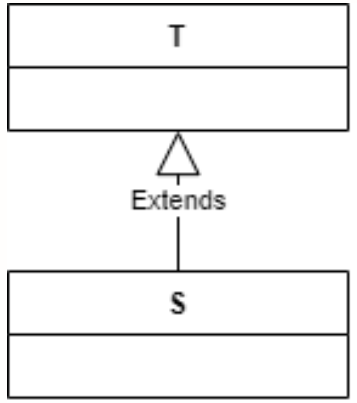


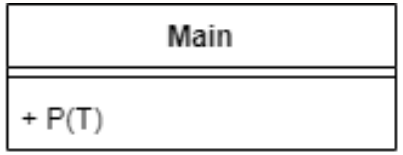



```
Main{
    main(){
        S O1 = new S();
        T O2 = new T();
        P(O1);
        P(O2);
    }
}
```

Liskov Substitution Principle





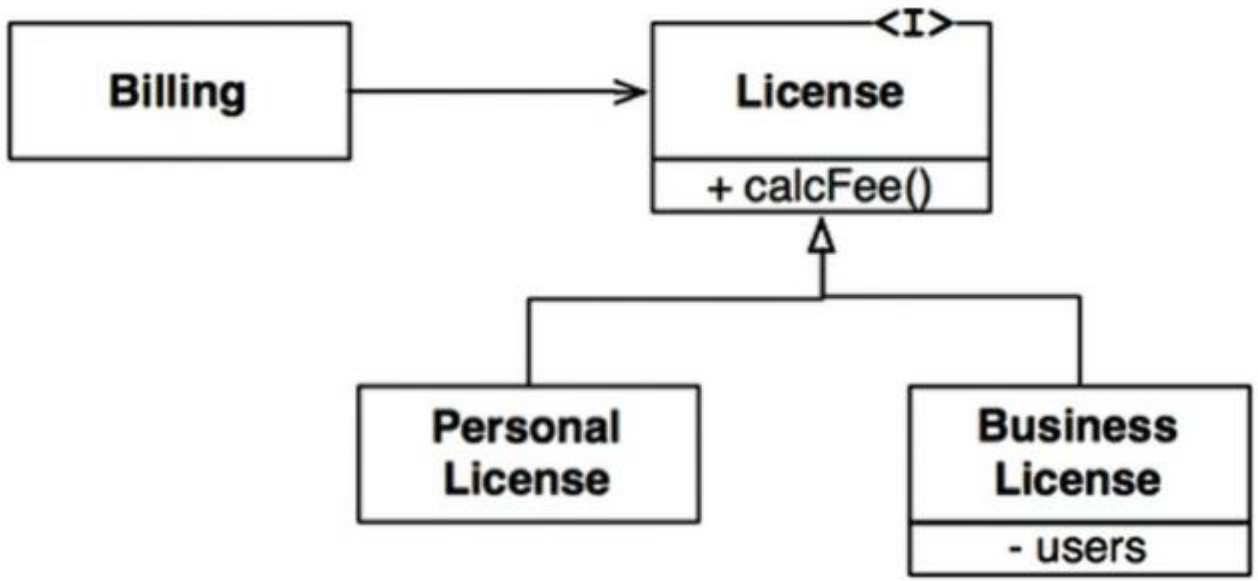



```
Main{
    main(){
        S O1 = new S();
        T O2 = new T();
        P(O1);
        P(O2);
    }
}
```

The behavior of drawShape() is unchanged when object of class Shape is substituted by object of class Rectangle or class Circle

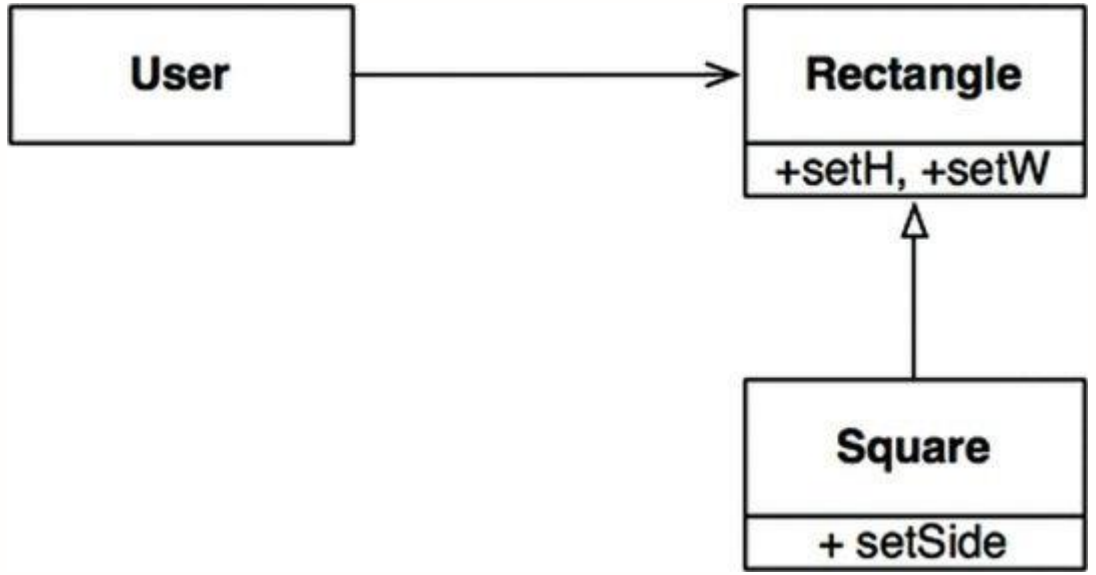
Liskov Substitution Principle

- Billing class has a method named calcFee(), which is called by the Billing application.
- There are two “subtypes” of License: PersonalLicense and BusinessLicense. They use different algorithms to calculate the license fee.
- This design conforms to the LSP because the behavior of the Billing application does not depend, in any way, on which of the two subtypes it uses. Both of the subtypes are substitutable for the License type



Liskov Substitution Principle

- In this example, Square is not a proper subtype of Rectangle because the height and width of the Rectangle are independently mutable; in contrast, the height and width of the Square must change together.



References

- Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. Pearson. Pearson. 2002
- Martin, Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson. 2017.
- https://www.oodesign.com/open-close-principle.html
- https://www.oodesign.com/liskov-s-substitution-principle.html

bridge to the future

http://www.eepis-its.edu