Object Oriented Design in practice

Instructor:

Agenda

After the course, student will:

- Understand about software design principles from examples.
- Understand and apply simple design pattern in the common situations.

How to avoid a bad design ?

3 important characteristics of a bad design:

- **Rigidity**: It's hard to change because every change affects too many other parts of the system.
- **Fragility**: When you make a change, unexpected parts of the system break.
- **Immobility**: It is hard to reuse in another application because it cannot be disentangled from the current application.

Rigidity (nghiêm ngặt): It's hard to change…

Fragility (dễ vỡ): When you make a change…

Immobility (không nhúc nhích): It's hard to reuse…

Design Principles

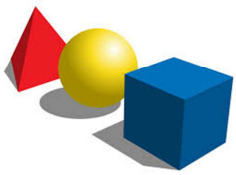
1. Open Close Principle (OCP)
2. Dependency Inversion Principle (DIP)
3. Interface Segregation Principle (ISP)
4. Single Responsibility Principle (SRP)
5. Liskov's Substitution Principle (LSP)
6. Summary

Design Principles #1:Open Close Principle

- **Open for extension**
- But **closed for modifications**.

Design Principles #1: Graphic editor

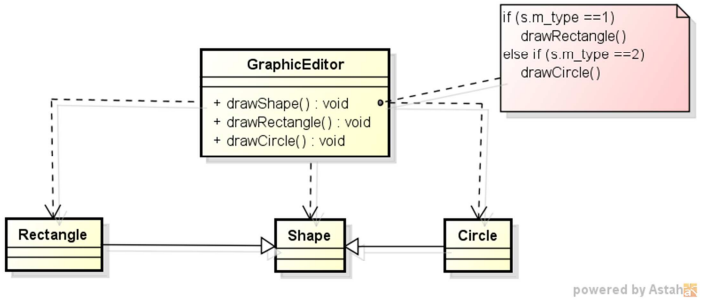
- Graphic editor is a program that has been developed to handle drawing other shapes (Circle, Rectangle, ...).



Ví dụ này thực thi một graphic editor mà handle việc vẽ các hình khác nhau (Rectangle, Circle).

Design Principles #1: A temporal solution

- Have GraphicEditor class to handle drawing
- Have 2 classes representative to Rectangle and Circle objects.
- Rectangle and Circle share so many properties, we create a new abstract base class (Shape).



Thao tác vẽ hình nằm ở hàm drawShape() với tham số là kiểu Shape.

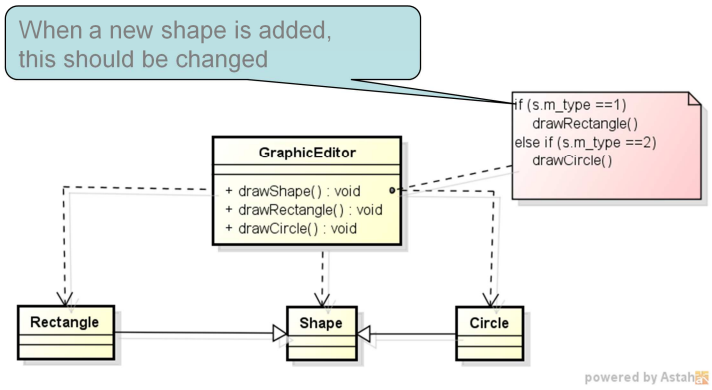
Nếu tham số này thuộc loại Rectangle thì sẽ gọi hàm drawRectangle() để vẽ hình Rectangle.

Và tương tự nếu tham số là thuộc về Circle thì xử lý sẽ gọi hàm drawCircle() để vẽ hình Circle.

Điều kiện để phán đoán tham số được xử lý bằng câu lệnh if trong hàm drawShape().

Design Principles #1: Violation points for extension

- What are violation points for extension ?



- Lecture nên hỏi SV đâu là điểm vi phạm nguyên tắc Open Close trong ví dụ này:

Design Principles #1: Violation points for extension

- Need re-create Unit Testing for GraphicEditor class, including drawShape() method.
- Need deeply understand logic in “if” statement.
- Affect the existing functionality.

Trong trường hợp một hình mới () được add thêm thì cần phải modify lại xử lý trong câu lệnh if(trên hình vẽ) và khi đó:

1) Cần phải thực hiện lại unit testing cho class GraphicEditor, bao gồm hàm drawShape().

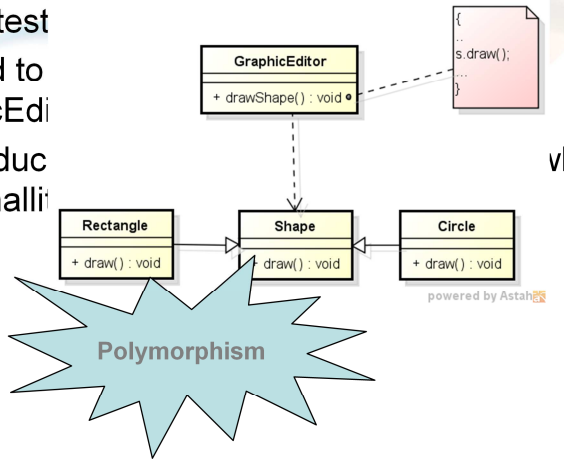
2) Developer cần phải hiểu rõ logic của xử lý trong câu lệnh if.

3) Có khả năng ảnh hưởng đến xử lý vẽ của hình Rectangle và Circle.

Do đó việc mở rộng của thiết kế này là khá khó khăn (**close** *for extension*).

Design Principles #1: Improvement

- no unit test
- no need to GraphicEdi
- it's a reduc vhen new functionalli



Design Principles #1: Conclusion

- OCP is only a principle.
- Introduces new level of abstraction increasing the complexity of the code.
- OCP should be applied in that area which are most likely to be changed.
- Design patterns that help us to extend code without changing it such as : **Decorator, Factory Method, Observer patterns, …**

Design Principles #2: Dependency Inversion Principle

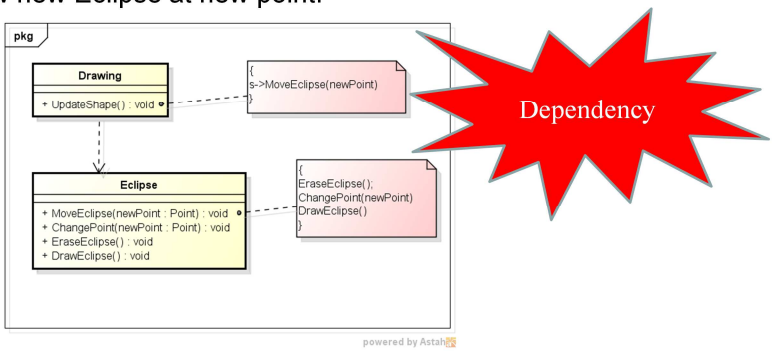
- High-level modules **should not** depend on low-level modules. Both **should depend on abstraction**
- Abstractions **should not** depend on details. Details **should** depend on abstractions.

Low level classes the classes implement basic and primary operations(disk access, network protocols,...)

High level classes the classes encapsulate complex logic(business flows, ...).

Design Principles #2: Problems

- How to update a new Eclipse from old Eclipse when it moved to other position ?
  1. Erase old Eclipse at the current point.
  2. Change new point for Eclipse.
  3. Draw new Eclipse at new point.



Ví dụ này mô tả việc cập nhật hình khi di chuyển một hình vẽ (Eclipse),

ta sẽ thực hiện các bước :

1) Xóa hình eclipse tại vị trí tọa độ hiện tại

2) Cập nhật tọa độ mới cho hình

3) Vẽ hình eclipse ở vị trí mới.

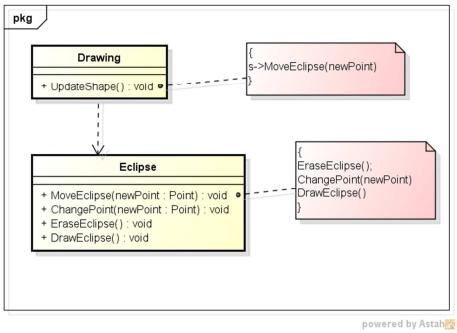
Drawing class đang thể hiện như vai trò của high level class

Trong khi Eclipse thể hiện vai trò low level class.

Bên cạnh đấy, trong bản thân class Eclipse: MoveEclipse() đóng vai trò high-level module, còn EraseEclipse(), DrawEclipse() và ChangePoint() đóng vai trò low level module.

Design Principles #2: Violation points of dependency

- **Drawing class** *depends on* **Eclipse class**
- **MoveEclipse()**
  depends on
  EraseEclipse()/ChangePoint()/DrawEclipse()

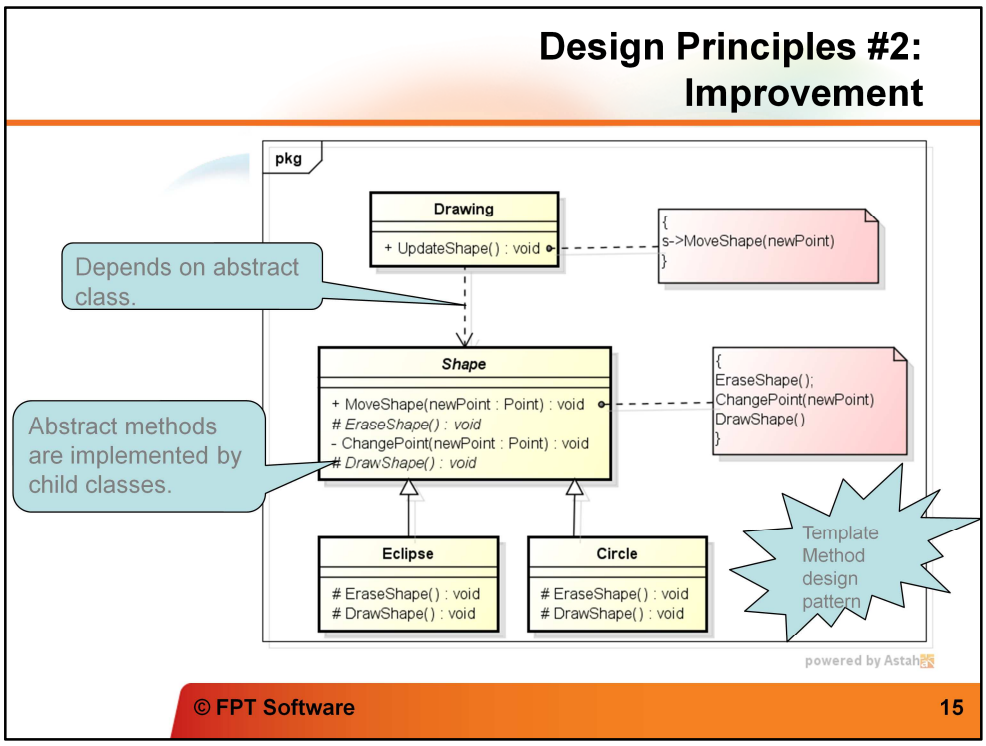


Drawing class (high-level class) đang phụ thuộc vào Eclipse class (low-level class).

MoveEclipse() phụ thuộc trực tiếp vào 3 hàm EraseEclipse(), DrawEclipse() và ChangePoint().

Vì thế đã vi phạm vào nguyên tắc Dependency Inversion.

Yêu cầu sinh viên trả lời -> chuyện gì sẽ sảy ra khi ta cần thực hiện cập nhật hình khi di chuyển một hình … Circle ?



Drawing class (high-level class) không phụ thuộc vào Eclipse class (low-level class) mà phụ thuộc vào abstract class (Shape).

MoveShape() trong class Shape chỉ phụ thuộc trực tiếp vào hàm ChangePoint() (xử lý common cho các loại Eclipse, Circle). 2 hàm EraseEclipse(), DrawEclipse() là abstract method và được implement cụ thể trong Eclipse hay Circle class.

Design Principles #2: Conclusion

- This principle means that the high level classes are not working directly with low level classes.
- Should use some creational design patterns can be used, such as **Factory Method**, **Abstract Factory**, **Prototype** to instantiate new low level objects inside the high level classes.
- **The Template Design Pattern** is an example where the DIP principle is applied.
- If we have a class functionality that is more likely to remain unchanged in the future there is not need to apply this principle.

In this case instantiation of new low level objects inside the high level classes(if necessary) can not be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype.

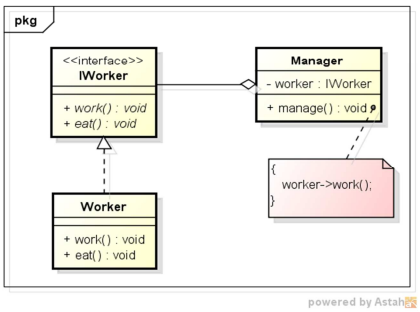
Design Principles #3:Interface Segregation Principle

- **Many** client specific interfaces are **better** than **one general purpose** interface

Nếu có một class đối tượng được sử dụng bởi nhiều Client, thay vì dồn hết các chức năng vào trong class đối tượng đó thì chúng ta sẽ phân loại các chức năng ứng với nhu cầu của từng Client.

Design Principles #3: Example

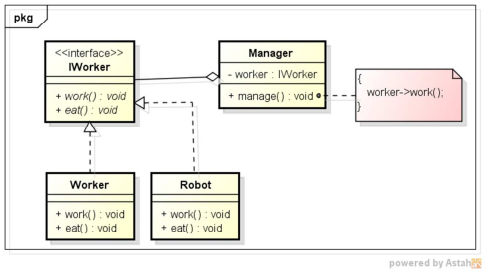
- Workers work and they need a daily launch break to eat.
- Worker class implements IWorker interface with work() and eat() method.
- Manager class which represent the person which manages the workers and only use work() method from IWorker interface.

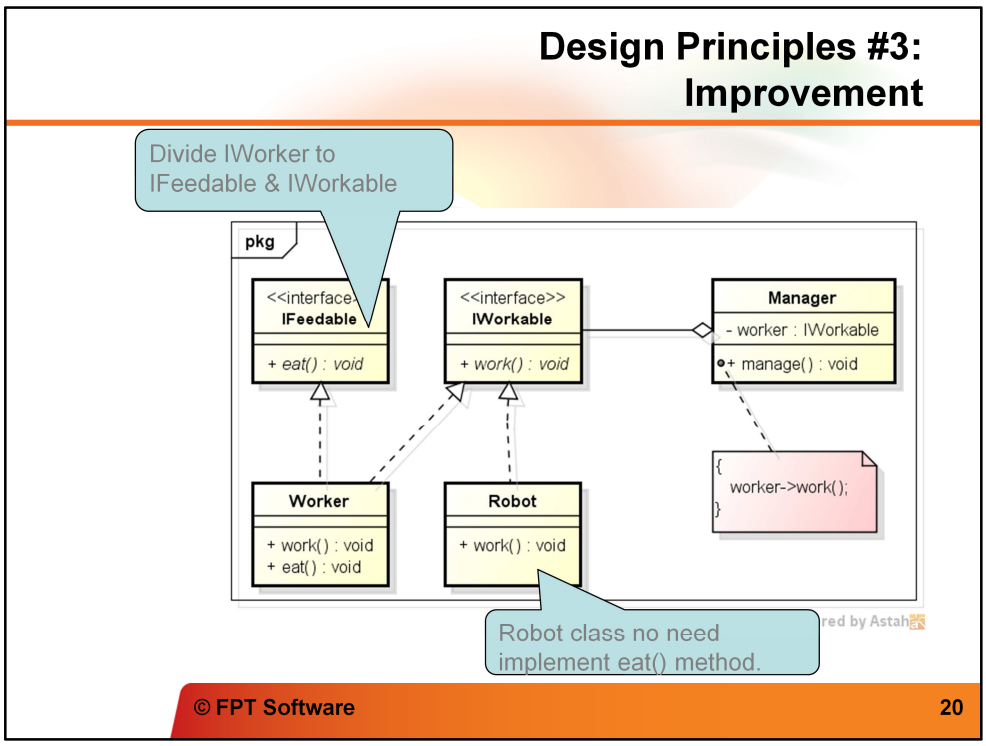


Với phương pháp *"chia để trị"* như hình bên phải , thiết kế đảm bảo rằng những thay đổi để đáp ứng nhu cầu của bất kỳ Client nào cũng không thể ảnh hưởng đến các Client còn lại.

Design Principles #3: Detect problems

- Add new Robot class and it need to implement the IWorker interface because robots works.
- But the new Robot class don't have to implement eat() method because they don't eat.
- If we keep the present design, the new Robot class is forced to implement the eat method.





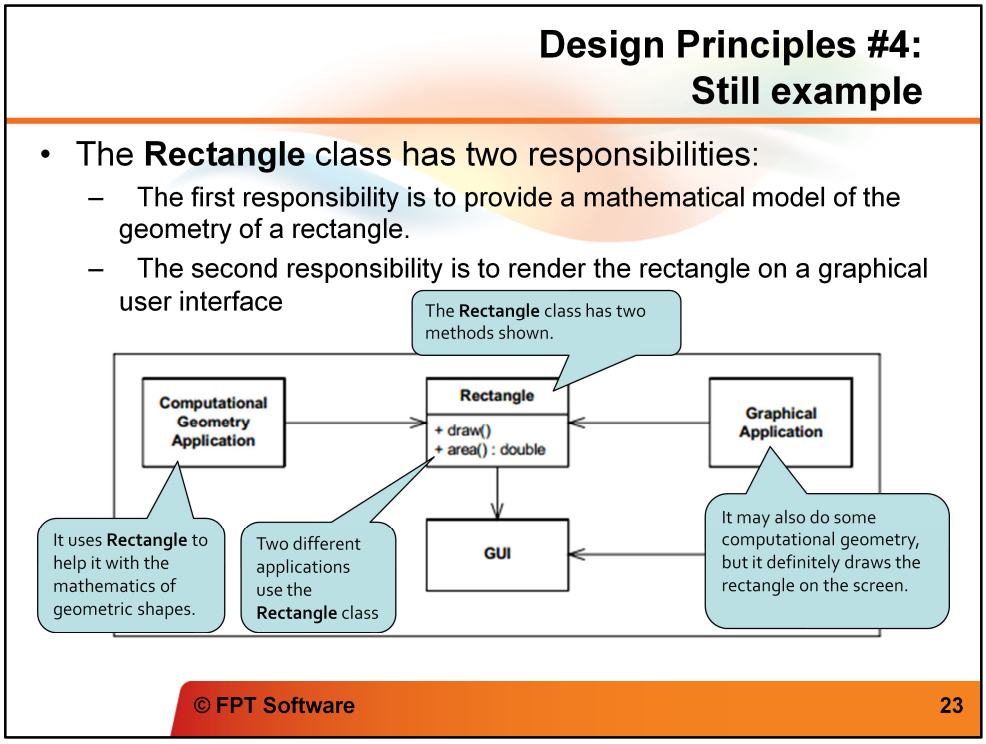
Với phương pháp *"chia để trị"* như hình, thiết kế đảm bảo rằng những thay đổi để đáp ứng nhu cầu của bất kỳ Client nào cũng không thể ảnh hưởng đến các Client còn lại.

Design Principles #3: Conclusion

- If the design is already done fat interfaces can be segregated using the **Adapter pattern**.
- If we are going to apply it more than is necessary it will result a code containing a lot of interfaces with single methods and increase the complexity of code.

Design Principles #4: Single Responsibility Principle

- A class should have only a single responsibility



Hình trên mô tả 1 ví dụ violate the SRP

The **Rectangle** class has two methods shown. One draws the rectangle on the screen, the other computes the area of the rectangle.

Two different applications use the **Rectangle** class:

One application does computational geometry. It uses **Rectangle** to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen.

The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.

The **Rectangle** class has two responsibilities:

The first responsibility is to provide a mathematical model of the geometry of a rectangle.

The second responsibility is to render the rectangle on a graphical user interface.

The violation of SRP causes several nasty problems:

Firstly, we must include the GUI in the computational geometry application.

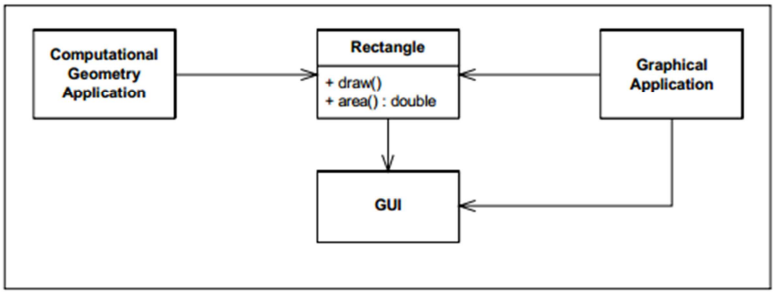
Secondly, if a change to the **GraphicalApplication** causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the **ComputationalGeometryApplication**.

A better design is to separate the two responsibilities into two completely different classes as shown [Hình bên dưới].

This design moves the computational portions of **Rectangle** into the

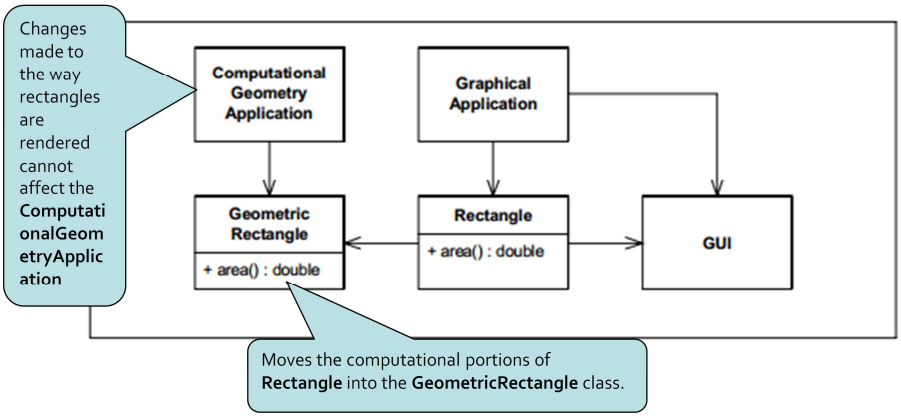
Design Principles #4: Detect problems

- The violation of SRP causes several nasty problems:
  - Must include the GUI in the computational geometry application.
  - If a change to the **GraphicalApplication** causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the **ComputationalGeometryApplication.**



Design Principles #4: Improvement

- Separate the two responsibilities into two completely different classes



A better design is to separate the two responsibilities into two completely different classes as shown [Hình bên tren].

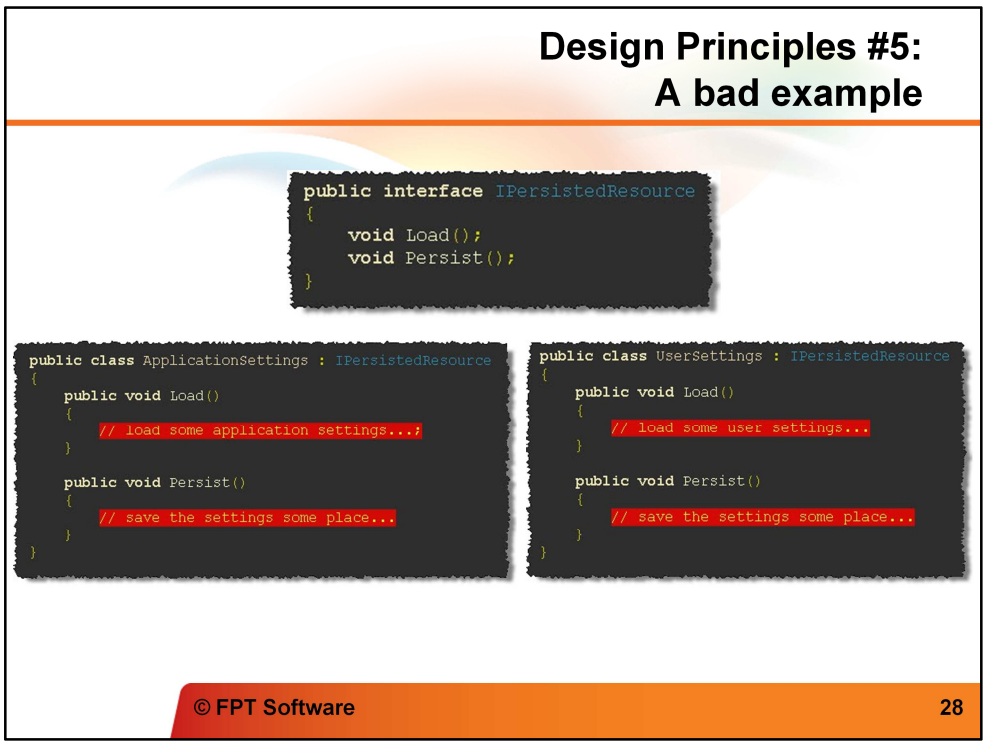
Design Principles #4: Conclusion

- Represent a good way of identifying classes during the design phase of an application
- Remind to think of all the ways a class can evolve.
- A good separation of responsibilities is done only when the full picture of how the application should work is well understood.

Design Principles #5: Liskov's Substitution Principle

- Just an extension of the Open Close Principle.
- Derived types must be completely substitutable for their base types.

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.



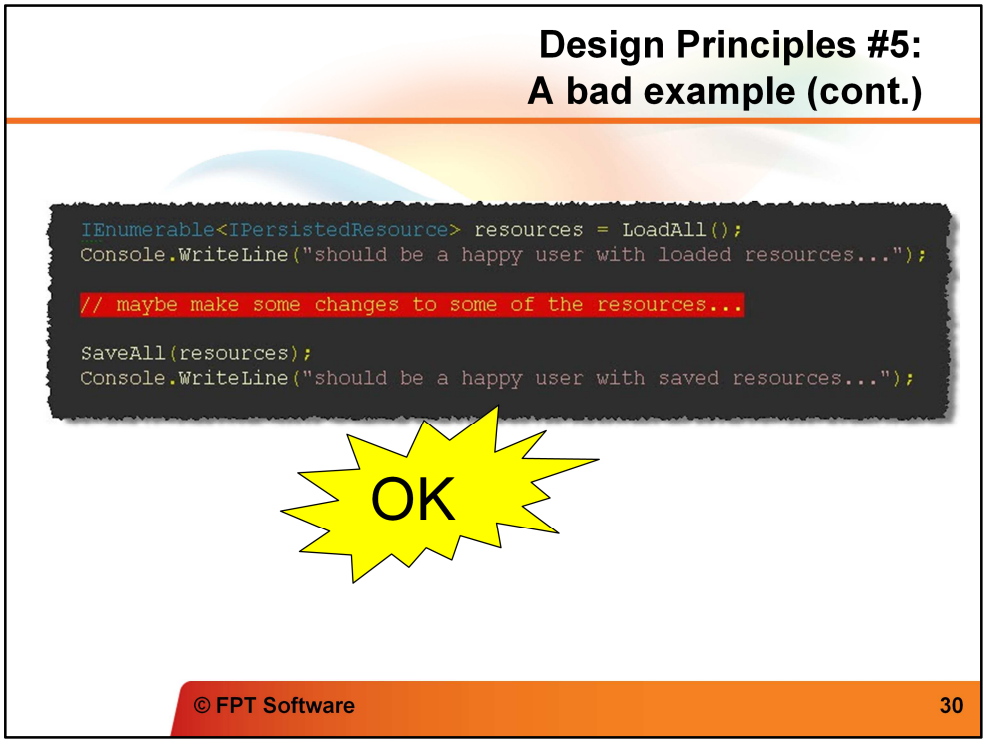
Chúng ta define 1 interface và 2 classes như trên hình vẽ.

Design Principles #5: A bad example (cont.)

```
static IEnumerable<IPersistedResource> LoadAll()
{
    var allResources = new List<IPersistedResource>
                           {
                               new UserSettings(),
                               new ApplicationSettings()
                           };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

```
static void SaveAll(IEnumerable<IPersistedResource> resources)
{
    resources
        .ForEach(r => r.Persist());
}
```


Implement 2 hàm để load và save data.



Lúc này khi chạy chương trình không có vấn đề gì.

Design Principles #5: A bad example (cont.)

New class

```
public class SpecialSettings : IPersistedResource
{
    public void Load()
    {
        // stuff...
    }

    public void Persist()
    {
        throw new NotImplementedException();
    }
}
```

```
static IEnumerable<IPersistedResource> LoadAll()
{
    var allResources = new List<IPersistedResource>
                           {
                               new UserSettings(),
                               new ApplicationSettings(),
                               new SpecialSettings()
                           };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```


A new class is added to the system in order to handle, let's say, some "special settings":

Design Principles #5: A bad example (cont.)

```
static void SaveAll(IEnumerable<IPersistedResource> resources)
{
    resources
        .ForEach(r =>
                    {
                        if (r is SpecialSettings)
                            return;

                        r.Persist();
                    });
}
```

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

Design Principles #5: Improvement

```
static void SaveAll(IEnumerable<IPersistResource> resources)
{
    resources
        .ForEach(r => r.Persist());
}

static IEnumerable<ILoadResource> LoadAll()
{
    var allResources = new List<ILoadResource>
                           {
                               new UserSettings(),
                               new ApplicationSettings(),
                               new SpecialSettings()
                           };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

Design Principles #5: Improvement (cont.)

```
public interface ILoadResource
{
    void Load();
}

public interface IPersistResource
{
    void Persist();
}
```


```
static void SaveAll(IEnumerable<IPersistResource> resources)
{
    resources
        .ForEach(r => r.Persist());
}

static IEnumerable<ILoadResource> LoadAll()
{
    var allResources = new List<ILoadResource>
                           {
                               new UserSettings(),
                               new ApplicationSettings(),
                               new SpecialSettings()
                           };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

Design Principles #5: Conclusion

- LSP is about letting the user handle different objects that implement a super type without checking what the actual type they are.
- This principle provides an alternative to do type-checking and type-conversion.
- This principle is achieved through pull-up refactoring or applying patterns such as **Visitor.**

Summary

- **S**RP - A class should have only one reason to change.
- **O**CP - Open for extension but closed for modifications.
- **L**SP - Derived types must be completely substitutable for their base types.
- **I**SP - Many client specific interfaces are better than one general purpose interface.
- **D**IP - High-level modules should not depend on low-level modules. Both should depend on abstraction.

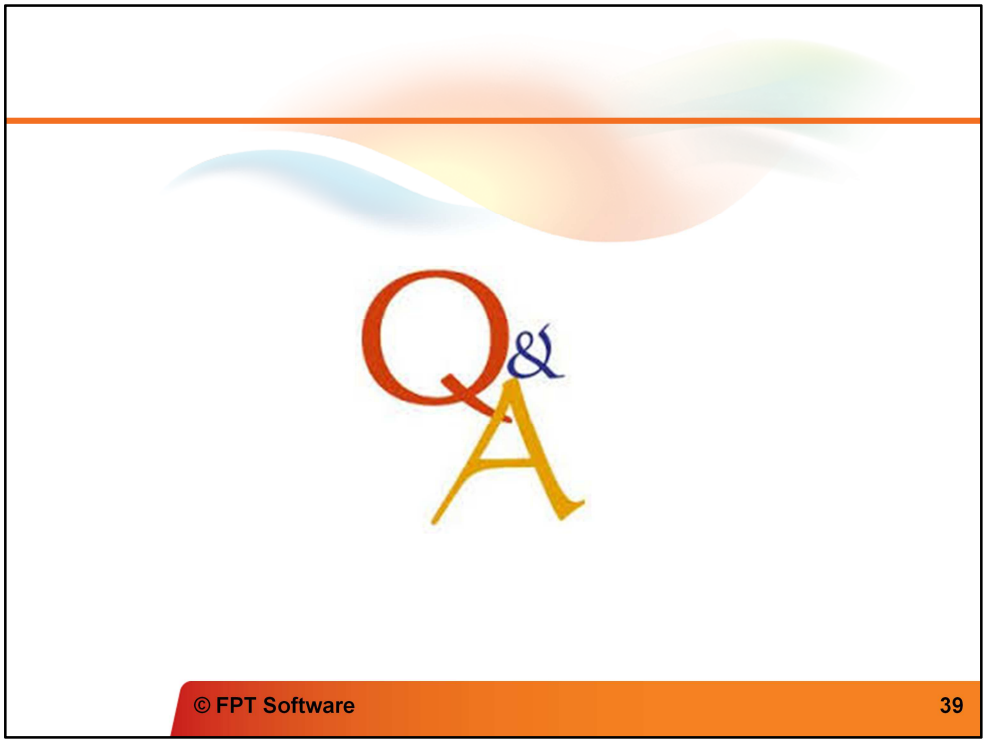
SOLID Principles

Summary (cont.)

- Making a flexible design involves additional time and effort spent for it.
- More complex code, but more flexible.
- Applying should be done based on experience and common sense in identifying the areas where extension of code are more likely to happens in the future.

Reference

- http://www.oodesign.com/design-principles.html
- O'Reilly HeadFirst - Object Oriented Analysis and Design
- O'Reilly HeadFirst - Design Patterns
- Addison-Wesley Professional - Refactoring: Improving the Design of Existing Code


Q&A