Design Smells

"Uncle Bob" Martin, the architect of the SOLID principles, identifies several "design smells" that are symptomatic of "rotting software."

Outline for Week 14

I. Design smells
II. Single Responsibility
III. Liskov Substitution
IV. Interface Segregation
V. Dependency Inversion

Rigidity

The system is hard to change because every change forces changes to other parts of the system.

You start to make what seems to be a simple change, but as you get into it, you find that it impacts more code than you expected. And when you fix the code in the other places, it affects still more modules. What principle or guideline from past weeks does this violate?

Fragility

A single change tends to "break" the program in many places.

Often those places have little apparent relation to the place where the code is changed. Patching those modules may make the problem worse later on.

Immobility

Parts of the code could be useful in other systems, but it is easier to rewrite them than to extricate them and reuse them.

Viscosity

When changes need to be made, the design is hard to preserve.

It is easier to hack a change into the code than to make it in a way that follows the principles of the design. *Example:* Instead of subclassing a base class again, use case statements to add new functionality.

Needless complexity

The design includes elements that aren't currently useful. Maybe the designer *expects* them to be useful later on …

Needless repetition

What's another name for this problem?

Opacity

A module is difficult to understand, not written in a clear and direct manner.

Code tends to become more opaque as it ages, because no one is intimately familiar with it any longer.

SOLID Principles

The SOLID principles are an acronym for five design principles that are not patterns, but just rules to be followed when designing programs. We have already seen two of them:

- the Open-Closed principle, and
- the Liskov Substitution Principle.

The Single-Responsibility Principle

[SaaS §11.3] The Single-Responsibility Principle is

> A class should have only one reason to change.

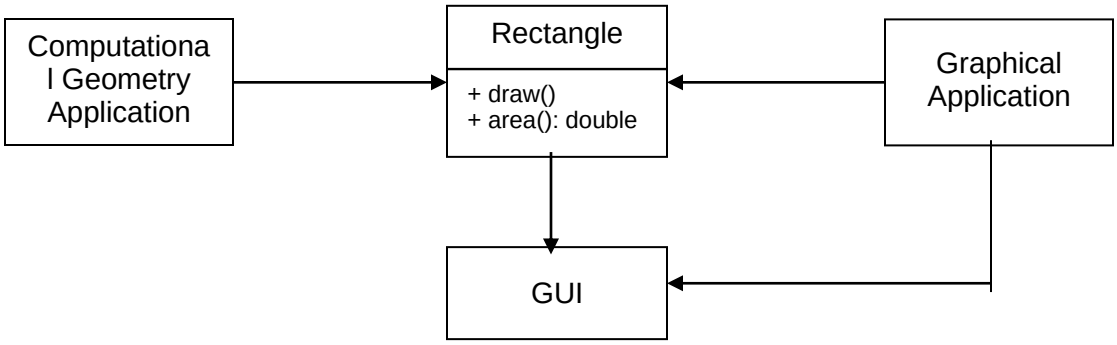
In a way, we have already seen this principle. Good cohesion (Lec. 20) dictates that, "Every class should be responsible for doing one thing only and doing it well."

But what does "doing one thing only" mean? Martin says it means that a class should have only one reason to change.

We saw one of Martin's favorite examples in discussing the Visitor pattern at the end of last week's class. Do you remember what that was?

Why would doing that in a single class violate the SRP?

Another of Martin's examples is a Rectangle class that has two responsibilities: calculate area and draw itself.



Two applications use Rectangles. Only one needs to draw the rectangle.

In a static language, the GUI class would have to be included in both applications. Changes to the **`draw()`** method would force the computational-geometry application to be recompiled, even though it doesn't use the method.

A dynamic language doesn't have these problems, but still
in order to change the view, you have to change model class.

Now, it is possible to go overboard with this principle. Too many classes are bad, too. The ideal number of methods for a class ≠ 1.

Martin clarifies what he means by a single reason for change: "*Gather together the things that change for the same reasons. Separate those things that change for different reasons.*"

Here's an example of a **`CityMap`** class.

"Bad" Example

Main class: **`CityMap`**

In this example, the **`CityMap`** class represents a map consisting of a list of cities with various attributes. Although this represents a single logical object, the **`CityMap`** class takes on several very separate

pieces of functionality which should, according to this principle, be divided into several classes. Those functionalities include managing the list of cities (add and remove), drawing the map on the screen, and calculating the total population.

"Good" Example

The good example simply splits the **`CityMap`** class into two classes, **`Map`** and **`CityList`**. **`CityList`** maintains the **`ArrayList`** of cities and also allows calculating the total population. The Map class focuses solely on drawing the map on a screen. This fixes the issues with the "bad" example, as each class now focuses solely on operations related to one set of data.

First, say which components of the "bad" example should go into each class in the "good" example.

Then, fill in the blanks in the "good" example.

The Liskov Substitution Principle

[SaaS §11.5] We've already seen the Liskov Substitution Principle (Lec. 16), but not the harm of violating it.

A short statement of it is,

> Subtypes must be substitutable for their base types.

If they are not, you have to be careful in writing code that uses the base type. One example from StackExchange:

Suppose you have a class **`Task`** and a subclass **`ProjectTask`**. **`Task`** has a **`close()`** method that doesn't work for **`ProjectTask`**.

Here is some code that uses **`close()`**.

```
public void processTaskAndClose(Task taskToProcess)
{
    taskToProcess.execute();
    taskToProcess.dateProcessed = DateTime.now;
    taskToProcess.close();
}
```

You can't be sure this code will work if a **`ProjectTask`** is passed to **`processTaskAndClose`**. So you need to put some kind of **if** statement or **`case`** statement around the call to **`close()`**.

Here's an exercise involving the LSP.

"Bad" Example

In this example, a **`Computer`** object keeps track of its amount of RAM and its OS version. It also has methods for upgrading the RAM and updating the OS. Two classes, a **`DesktopComputer`** and a **`Phone`**, extend this class and implement its methods.

A **`ComputerUpgrader`** object claims to be able to upgrade any **`Computer`** (that is, add more RAM and update the OS), but it really can't add more RAM to a phone, so it must check to make sure the **`Computer`** object it has been given isn't a **`Phone`**.

This violates the LSP, as a **`Phone`** cannot fully be substituted for a **`Computer`**.

"Good" Example

The most straightforward method of solving the above problems is to add a new interface **`HardwareUpgradable`**, which is only implemented by **`Computer`**s which can have their hardware upgraded (**`DesktopComputer`** can, Phone cannot).

Next, by splitting the upgrade method in **`ComputerUpgrader`** into **`upgradeRAM`** (which accepts **`HardwareUpgradable`**) and **`upgradeOS`** (which accepts any computer), the issue can be resolved. No type-checking is necessary.

Fill in the blanks to finish this example.

The Interface-Segregation Principle

The Single-Responsibility Principle tells us that classes that are too big are no good. The Interface-Segregation Principle says the same thing about interfaces. It is,

> Clients should not be forced to depend on methods that they do not use.

If you know Java, you are probably familiar with the **`MouseListener`** and **`MouseMotionListener`** interfaces. Both of them handle **`MouseEvents`**. Why are two listeners needed for **`MouseEvents`**, when all other kinds of events have only one listener interface?

This video describes the issue of read streams vs. read-and-write streams.

OK, you might say, this makes sense for Java, but how about Ruby? Ruby doesn't even have interfaces!

Indeed, dynamically typed languages don't need interfaces. Why?

Ducktype.

The issue, then, is how to give a Ruby class access to the methods it needs from another class, rather than giving it access to *all* the methods, which it would get if it inherited from the class.

The **forwardable** mixin has a **def_delegator** method that allows one Ruby class to use some, but not all, of the methods of the class it delegates to. This video shows how **forwardable** can be used to create a **Moderator** class that can edit posts, but not create or delete them.

Here is an exercise involving the ISP.

"Bad" Example

In this example, a single interface, **`Game`**" is created, for two classes, **`SingleplayerGame`** and **`MultiplayerGame`**.

This is on the surface a logical structure. However, in this case, the methods **`getServerList`** and **`pauseGame`**, published in the **`Game`** interface, are not used by both clients (As a **`MultiplayerGame`** cannot be paused, and a **`SingleplayerGame`** does not have servers).

Because of this mismatch, the **`SingleplayerGame`** is forced to throw an **`UnsupportedOperationException`** when **`getServerList`** is called on it, and **`MultiplayerGame`** is forced to throw an **`UnsupportedOperationException`** when **`pauseGame`** is called on it.

This demonstrates a violation of the Interface Segregation principle, as a single, logical but ill-fitting interface is used by several clients, despite clear incompatibilities.

"Good" Example

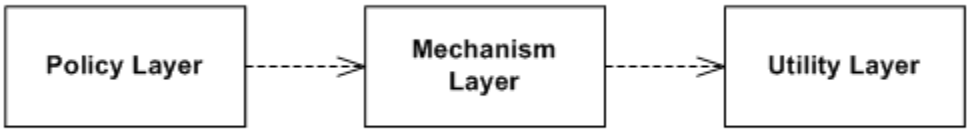
This example is derived from the “bad" example. In this case, the single interface **`Game`** was split into three interfaces with a single method each: **`BasicGame`**, **`OnlineGame`**, and **`PausableGame`**.

With this split, **`MultiplayerGame`** can implement **`BasicGame`** and **`OnlineGame`**, but it does not need to implement **`PausableGame`** (as it is not pausable), and **`SingleplayerGame`** can implement **`Basicgame`** and **`PausableGame`** (as it can be paused but is not online). This corrects the need to throw **`UnsupportedOperationExceptions`**, and follows the ISP by dividing one general purpose interface into several smaller interfaces.

Fill in the blanks in the “good” code.

The Dependency-Inversion Principle

[SaaS §11.6] Any object that uses another object to carry out its work is said to *depend* on the other object. A very common layered architecture has higher-level modules depending on lower-level modules, like this.



However, the Dependency-Inversion Principle says this is not good. It says,

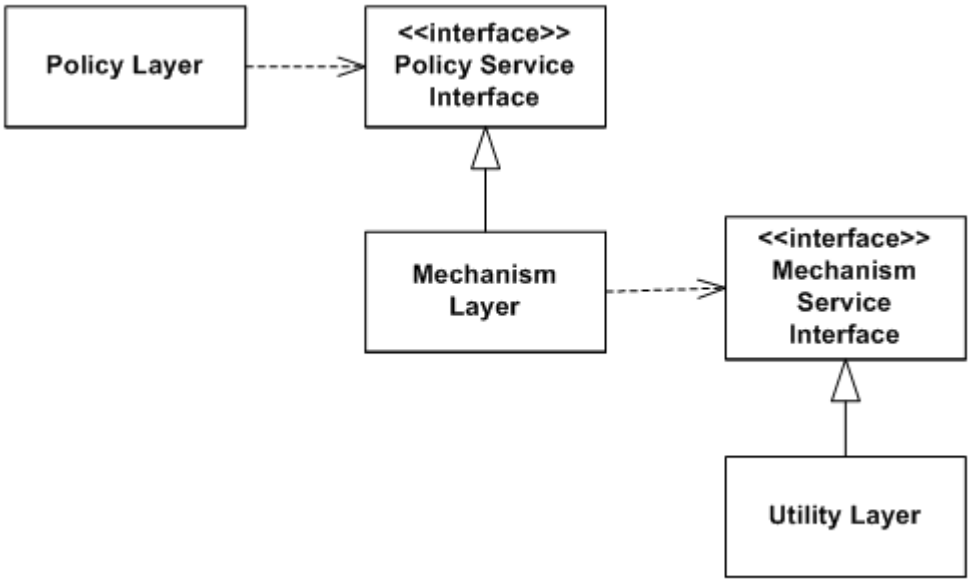
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

The reason that it's bad for high-level modules to depend on low-level modules is that a change to a low-level module can require a change to a high-level module.

This makes it hard to contain the damage when editing low-level modules.

The reason that this is called an "inversion" principle is that it goes against the advice of other software-development methodologies, such as Structured Analysis and Design.

Interposing interfaces between the various levels allows either level to change without affecting the other, assuming that the same interface is still implemented.



The second part of the principle says, essentially, that high-level modules should not get involved in performing low-level functions.

This video explains that

- the Coca Cola CEO should not deliver products to 7/11, and
- the Ruby ActiveRecord class should not say how an application's **users** table is to be structured.

Just as with the Single-Responsibility and Interface-Segregation Principles, it is possible to go overboard with Dependency Inversion.

The system should not be filled with single-method classes and interfaces, nor should there be an interface between every two classes.

Interfaces should be reserved for the boundaries between layers, or collections of classes that are otherwise cohesive.

Here is an exercise involving the Dependency-Inversion Principle.

"Bad" Example

Main Class: **`Bank`**

In this example, a **`Bank`** class is a high-level class with complex functionality (redacted for this example). One piece of that functionality is handling transactions between various accounts (simple, low-level classes).

The **`Bank`**, however, is very tightly coupled with the **`CheckingAccount`** and **`SavingsAccount`** classes. Adding additional account types (such as a **`MoneyMarketAccount`**, **`InvestmentAccount`**, or **`RetirementAccount`**) would exponentially increase the complexity of Bank.

"Good" Example

Main Class: **`Bank`** (high level) / **`Account`**s (low level)

To correct the above issues, a layer of abstraction between the various types of accounts and the high-level **`Bank`** class is added. In this case, a simple **`BasicAccount`** interface is added between the layers.

Although each account may process transactions in different ways (for example, Federal Reserve Regulation D requires that savings accounts limit the number of transactions per month, but this does not apply to Checking accounts), a simple, uniform interface can be provided. This dramatically simplifies the **`Bank`** class, and will allow for new types of accounts to be added easily.

An argument could be made to make **`BasicAccount`** an abstract class. In this limited example, this would actually simplify the codebase, by allowing the repeated code found in various versions of **`deposit()`** and **`withdraw()`** to be moved to a single location. In that setup, classes which require checks/validation before accepting a deposit or withdraw could make those and then delegate to the abstract class.

However, in a more complete system, there may very well be scenarios where this structure would not work (for example, HSA/IRA/Business/Credit accounts may have very different deposit or withdrawal structures). Therefore, the class is left as an interface, as that structure makes the example clearer (it is essential to clarify the interface as existing primarily as a layer between **`Bank`** and the **`Account`** classes).

Fill in the blanks to complete the code.