SOLID PRINCIPLES

PRINCIPLES OF CLASS DESIGN

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

SINGLE RESPONSIBILITY PRINCIPLE

A class should have one, and only one, reason to change.

Additional responsibilities lead to inappropriate coupling

Inappropriate coupling makes the design fragile

OPEN-CLOSED PRINCIPLE
BERTRAND MEYER

SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.

PRIMARY ATTRIBUTES

1. They are “Open For Extension”

This means that the behavior of the class can be extended. We can make the class behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

2. They are “Closed for Modification”

The source code of such a class does not change.

LISKOV SUBSTITUTION

What is wanted here is something like the following substitution property:

If for each object o1of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

INTERFACE SEGREGATION PRINCIPLE

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.

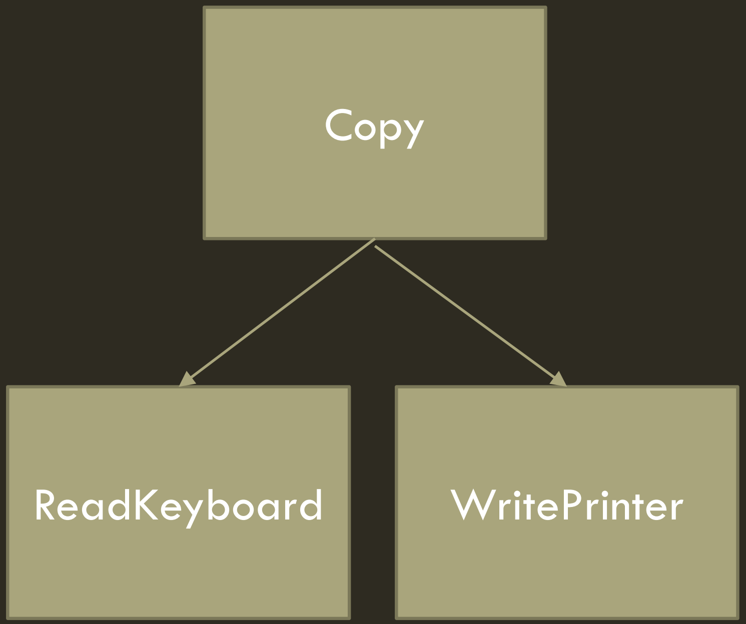
CAUSES OF BAD SOFTWARE DESIGN

It is hard to change because every change affects too many other parts of the system. (Rigidity)

When you make a change, unexpected parts of the system break. (Fragility)

It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility)

AN EXAMPLE



```
public void Copy()
{
   int c;

   while ((c = ReadKeyboard()) != -1)
        WritePrinter(c);
}
```

“ENHANCED COPY PROGRAM”

```
enum OutputDevice {printer, disk};

void Copy(OutputDevice dev)
{

    int c;

    while ((c = ReadKeyboard()) != -1)

    {

    if (dev == printer)

        WritePrinter(c);

    else

        WriteDisk(c);

    }

}
```

DEPENDENCY INVERSION PRINCIPLE

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

REFACTORING

The other half of TDD

WHAT IS REFACTORING?

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. – Martin Fowler

Refactoring is a long-term , cost-efficient, and responsible approach to software ownership

REFACTORING EXERCISE

The Sieve