The following are the symptoms of bad software designs, as defined in Ch. 7 of [Martin, 2003]:
- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing (i.e. forced into hacks)
- Needless Complexity: Overdesign
- Needless Repetition: Mouse abuse
- Needless Repetition: (Just kidding)
- Opacity: Disorganized expression

Also known as design smells.
Refactoring

How do we modify our designs and our code to prevent rot. Refactoring...

Refactoring “…the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure” [Fowler, 1999]

- You can refactor code:
  - e.g. Read ch. 5 of [Martin, 2003]
- You can refactor your design:
  - We will see many examples

The Single-Responsibility Principle (SRP)

A class should have only one responsibility.

OR

A class should have only one reason to change.

A class with several responsibilities creates unnecessary couplings between those responsibilities.

e.g. Rectangle Class

The Geometry Application is concerned with the mathematics of geometric shapes

The Graphical Application may also involve some geometry, but it also needs to draw geometric shapes

The Rectangle class has two responsibilities:
- Provide a mathematical model of a rectangle
- Render a rectangle

Problems created:
- Inclusion: The GUI must be included in the Geometry Application (C++: linked into executable, Java: GUI.class file included in JAR file)
- A change required for one application may affect the other (e.g. adding a colour attribute)
Solution:
- Separate the two responsibilities (math rep. + drawing) into two separate classes.

```java
interface Modem {
    void dial(String pno);
    void hangup();
    void send(char c);
    char recv();
}
```

Multiple responsibilities? You could say there are two:
- Connection management (dial, hangup)
- Data transfer (send, recv)

If connection management and data transfer are considered separate responsibilities then we can provide the following solution:

```java
<interface> Data Channel + send(char) + recv(): char
<interface> Connection + dial(pno : String) + hangup()
```

Consider our solution again. Modem Implementation has two responsibilities! Isn’t this bad? Yes but...
- This may be unavoidable due to h/w or OS constraints
- Even if Modem Implementation changes, other classes in the system should remain unaffected.
The Need for Design Principles
Refactoring
Design Principles
References
The Open-Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

OR
To change behaviour, add new code rather than changing existing code.

How? Abstraction.

e.g. Client Server

With regards to the Client, the following design does not conform to the OCP.

![Figure 9-1](image1.png)

If we want the Client to use a different Server, we must change the Client. However, the following design resolves this problem:

![Figure 9-2](image2.png)

The DrawShape function violates the OCP:

```cpp
class Shape {
    enum ShapeType {SQUARE, CIRCLE} itsType;
    Shape(ShapeType t) : itsType(t) {};

    class Circle : public Shape {
        Circle() : Shape(CIRCLE) {};
        void Draw();
    };

    class Square : public Shape {
        Square() : Shape(SQUARE) {};
        void Draw();
    };

    void DrawShape(const Shape& s) {
        if (s.itsType == Shape::SQUARE)
            static_cast<const Square&>(s).Draw();
        else if (s.itsType == Shape::CIRCLE)
            static_cast<const Circle&>(s).Draw();
    }
}
```

New derivatives of Shape require changes to DrawShape.

The use of virtual methods solves this problem:

```cpp
class Shape {
    public:
        virtual void Draw() const = 0;
};

class Square : public Shape {
    public:
        virtual void Draw() const;
        //...
};

class Circle : public Shape {
    public:
        virtual void Draw() const;
        //...
};

void DrawShape(const Shape& s) {
    s.Draw();
}
```
References
