Avoid Code Duplication (a.k.a. DRY - Don't Repeat Yourself)

Single Responsibility Principle

Related Concepts: Cohesion, Separation of Concerns

Every class should have a single responsibility. All of its services should be narrowly aligned with that responsibility.

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

EXAMPLE: Consider a class that compiles and prints a report. Such a class can be changed for two reasons. First, the content of the report can change. Second, the format of the report can change. These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes. It would be a bad design to couple two things that change for different reasons at different times.

Aspects of a system that change independently should be implemented in different classes.

Separation of Concerns: Create distance between dissimilar concepts in your code. This allows you to change one without affecting the other.

MORE EXAMPLES:

CsvWriter class: Combines both the destination and format of the output. These should be seprated into different classes (e.g., by passing in an output stream instead of hard-coding the output to System.out).

Java Date class (most of it is deprecated and replaced by the GregorianCalendar class). The Date class represents two notions: 1) moments in time, and 2) the Gregorian calendar (i.e., representing moments in time as month, day, year). Those different responsibilities were later separated into the Date and Gregorian Calendar classes.

Design Error: Divergent Change: A class that suffers many kinds of changes

Isolated Change Principle

Ideally, the implementation of a responsibility will be isolated in a single class, rather than spanning many different classes. (Or, if not a single class, a single package of classes.)

Ideally, there is a one-to-one mapping between responsibilities and classes.

This way, if something about a responsibility changes, there will be only one place in the code that needs to be changed.

This is not always achievable, but it is something to strive for.

Design Error: Shotgun Surgery: One change that alters many classes

Orthogonality

Related Concepts: Loose Coupling, Minimize Dependencies, Avoid Ripple Effects

The Single Responsibility Principle and Isolated Change Principle relate to another important design principle called Orthogonality.

What Is Orthogonality?

"Orthogonality" is a term borrowed from geometry. Two lines are orthogonal if they meet at right angles, such as the axes on a graph. In vector terms, the two lines are independent. Move along one of the lines, and your position projected onto the other doesn't change.

In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface.

Before we look at the benefits of orthogonal systems, let's first look at a system that isn't orthogonal.

A Nonorthogonal System

You're on a helicopter tour of the Grand Canyon when the pilot, who made the obvious mistake of eating fish for lunch, suddenly groans and faints. Fortunately, he left you hovering 100 feet above the ground. You rationalize that the collective pitch lever[2] controls overall lift, so lowering it slightly will start a gentle descent to the ground. However, when you try it, you discover that life isn't that simple. The helicopter's nose drops, and you start to spiral down to the left. Suddenly you discover that you're flying a system where every control input has secondary effects. Lower the left-hand lever and you need to add compensating backward movement to the right-hand stick and push the right pedal. But then each of these changes affects all of the other controls again. Suddenly you're juggling an unbelievably complex system, where every change impacts all the other inputs. Your workload is phenomenal: your hands and feet are constantly moving, trying to balance all the interacting forces.

Helicopter controls are decidedly not orthogonal.

[2] Helicopters have four basic controls. The cyclic is the stick you hold in your right hand. Move it, and the helicopter moves in the corresponding direction. Your left hand holds the collective pitch lever. Pull up on this and you increase the pitch on all the blades, generating lift. At the end of the pitch lever is the throttle. Finally you have two foot pedals, which vary the amount of tail rotor thrust and so help turn the helicopter.

Benefits of Orthogonality

As the helicopter example illustrates, nonorthogonal systems are inherently more complex to change and control. When components of any system are highly interdependent, there is no such thing as a local fix.

Tip 13

Eliminate Effects Between Unrelated Things

We want to design components that are self-contained: independent, and with a single, well-defined purpose (what Yourdon and Constantine call cohesion

[YC86]). When components are isolated from one another, you know that you can change one without having to worry about the rest. As long as you don't change that component's external interfaces, you can be comfortable that you won't cause problems that ripple through the entire system.

The Pragmatic Programmer(c) From Journeyman to Master

Authors: Hunt A., Thomas D.

Minimizing Dependencies

For example, suppose you are writing a class that generates a graph of scientific recorder data. You have data recorders spread around the world; each recorder object contains a location object giving its position and time zone. You want to let your users select a recorder and plot its data, labeled with the correct time zone. You might write

```
public void plotDate(Date aDate, Selection aSelection) {
     TimeZone tz =
          aSelection.getRecorder().getLocation().getTimeZone();
     ...
}
```

But now the plotting routine is unnecessarily coupled to three classes: Selection, Recorder, and Location. This style of coding dramatically increases the number of classes on which our class depends. Why is this a bad thing? It increases the risk that an unrelated change somewhere else in the system will affect your code. For instance, if Fred makes a change to Location such that it no longer directly contains a TimeZone, you have to change your code as well.

Rather than digging through a hierarchy yourself, just ask for what you need directly:

```
public void plotDate(Date aDate, TimeZone aTz) {
     ...
}
plotDate(someDate, someSelection.getTimeZone());
```

We added a method to Selection to get the time zone on our behalf: the plotting routine doesn't care whether the time zone comes from the Recorder directly, from some contained object within Recorder, or whether Selection makes up a different time zone entirely. The selection routine, in turn, should probably just ask the recorder for its time zone, leaving it up to the recorder to get it from its contained Location object.

Traversing relationships between objects directly can quickly lead to a combinatorial explosion[1] of dependency relationships.

The Pragmatic Programmer(c) From Journeyman to Master

Authors: Hunt A., Thomas D.

The Law of Demeter

The Law of Demeter for functions [LH89] attempts to minimize coupling between modules in any given program. It tries to prevent you from reaching into an object to gain access to a third object's methods. The law is summarized in Figure 5.1.

Figure 5.1. Law of Demeter for functions

The Law of Demeter for functions states that any method of an object should call only methods belonging to:

```
class Demeter {
private:
    C  c;
    int func();
public:
    // …
    void example(B& b);
};

void Demeter::example(B& b) {

    // itself
    int f = func();

    // any parameters passed to the method
    b.invert();

    // any objects it created
    A* a = new A();
    a->setActive();

    // any directly held component objects
    c.print();
}
```

By writing "shy" code that honors the Law of Demeter as much as possible, we can achieve our objective:

Tip 36

Minimize Coupling Between Modules

The Pragmatic Programmer(c) From Journeyman to Master

Authors: Hunt A., Thomas D.