

The Art of Readable Code

Surface-Level Improvements(Part One)





Agenda

- Packing information into name
- Names that can't be Misconstrued
- Aesthetics
- Knowing what to commend
- Making comments Precise and compact



The Goal of this book

The goal of this book is help you make your code better, highly readable and easy to understand.



Packing information into name

A lot of the names we see in programs are vague, like `tmp`. Even words that may seem reasonable, such as `size` or `get`, don't pack much information. This chapter shows you how to pick names that do.

- Choosing specific words
- Avoiding generic names (or knowing when to use them)
- Using concrete names instead of abstract names
- Attaching extra information to a name, by using a suffix or prefix
- Deciding how long a name should be
- Using name formatting to pack extra information



Choose Specific Words

For example, the word “get” is very unspecific, as in this example:

```
def GetPage(url):  
    ...
```

The word “get” doesn’t really say much. Does this method get a page from a local cache, from a database, or from the Internet? If it’s from the Internet, a more specific name might be `FetchPage()` or `DownloadPage()`.



Finding more colorful worlds

Word	Alternatives
send	deliver, dispatch, announce, distribute, route
find	search, extract, locate, recover
start	launch, create, begin, open
make	create, set up, build, generate, compose, add, new

Naming that can't be Misconstrued

KEY IDEA of this chapter is Actively scrutinize your names by asking yourself, "What other meanings could someone interpret from this name?"



Example: Filter()

Suppose you're writing code to manipulate a set of database results:

```
results = Database.all_objects.filter("year <= 2011")
```

What does results now contain?

- Objects whose year is ≤ 2011 ?
- Objects whose year is *not* ≤ 2011 ?

The problem is that filter is an ambiguous word. It's unclear whether it means "to pick out" or "to get rid of." It's best to avoid the name filter because it's so easily misconstrued.

If you want "to pick out," a better name is select(). If you want "to get rid of," a better name is exclude().



Aesthetics

In this chapter, we're concerned only with simple “aesthetic” improvements you can make to your code. These types of changes are easy to make and often improve readability quite a bit.

Imagine if you had to use this class:

```
class StatsKeeper {
public:
// A class for keeping track of a series of doubles
    void Add(double d); // and methods for quick statistics about them
private:    int count;          /* how many so far
*/ public:
            double Average();

private:    double minimum;
list<double>
    past_items
            ;double maximum;
};
```

```
// A class for keeping track of a series of doubles
// and methods for quick statistics about them.
class StatsKeeper {
    public:
        void Add(double d);
        double Average();

    private:
        list<double> past_items;
        int count; // how many so far

        double minimum;
        double maximum;
};
```




Rearrange Line Breaks to Be Consistent and Compact

Suppose you were writing Java code to evaluate how your program behaves under various network connection speeds. You have a `TcpConnectionSimulator` that takes four parameters in the constructor:

1. The speed of the connection (Kbps)
2. The average latency (ms)
3. The “jitter” of the latency (ms)
4. The packet loss (percentage)

Your code needed three different TcpConnectionSimulator instances:

```
public class PerformanceTester {  
    public static final TcpConnectionSimulator wifi = new TcpConnectionSimulator(  
        500, /* Kbps */  
        80, /* millisecs latency */  
        200, /* jitter */  
        1 /* packet loss % */);  
  
    public static final TcpConnectionSimulator t3_fiber =  
        new TcpConnectionSimulator(  
            45000, /* Kbps */  
            10, /* millisecs latency */  
            0, /* jitter */  
            0 /* packet loss % */);  
  
    public static final TcpConnectionSimulator cell = new TcpConnectionSimulator(  
        100, /* Kbps */  
        400, /* millisecs latency */  
        250, /* jitter */  
        5 /* packet loss % */);  
}
```



```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator(
            500,    /* Kbps */
            80,    /* millisecs latency */
            200,    /* jitter */
            1      /* packet loss % */);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(
            45000, /* Kbps */
            10,    /* millisecs latency */
            0,     /* jitter */
            0      /* packet loss % */);

    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator(
            100,    /* Kbps */
            400,    /* millisecs latency */
            250,    /* jitter */
            5      /* packet loss % */);
}
```

```
public class PerformanceTester {  
    // TcpConnectionSimulator(throughput, latency, jitter, packet_loss)  
    //                        [Kbps]    [ms]    [ms]    [percent]  
  
    public static final TcpConnectionSimulator wifi =  
        new TcpConnectionSimulator(500,    80,    200,    1);  
  
    public static final TcpConnectionSimulator t3_fiber =  
        new TcpConnectionSimulator(45000,  10,    0,    0);  
  
    public static final TcpConnectionSimulator cell =  
        new TcpConnectionSimulator(100,    400,    250,    5);  
}
```

Knowing What to Comment

INSTRUCTION MANUALS



NEEDED!!



NOT NEEDED



Key Idea

Don't comment on facts that can be derived QUICKLY from the code itself.

```
// The class definition for Account  
class Account {  
    public:  
        // Constructor  
        Account();  
  
        // Set the profit member to a new value  
        void SetProfit(double profit);  
  
        // Return the profit from this Account  
        double GetProfit();  
};
```

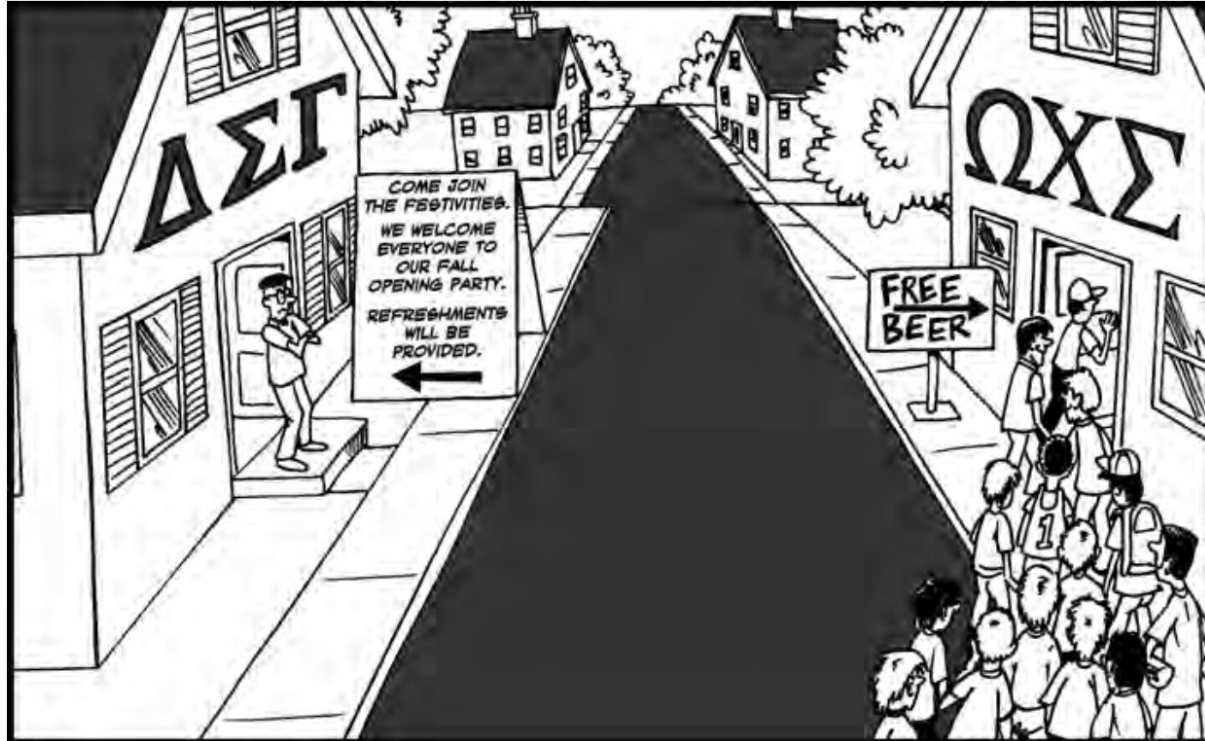


The word “quickly” is an important distinction, though. Consider the comment for this Python code:

remove everything after the second

```
name = '*'.join(line.split('*')[:2])
```

Making comments Precise and compact



Keep Comments Compact

Here's an example of a comment for a C++ type definition:

```
// The int is the CategoryType.  
// The first float in the inner pair is the 'score',  
// the second is the 'weight'.  
typedef hash_map<int, pair<float, float> > ScoreMap;
```

But why use three lines to explain it, when you can illustrate it in just one line?

```
// CategoryType -> (score, weight)  
typedef hash_map<int, pair<float, float> > ScoreMap;
```

Some comments need three lines of space, but this is not one of them.