8

Function Heuristics

What follows are some heuristics that I have found helpful in keeping my code clean. I don't follow them as hard-and-fast rules, but when all else is equal, these are my biases.

Function Arguments

Every argument you add to a function signature makes that function harder to understand. The easiest argument list to understand is the empty argument list.

In the first edition of this book, this statement confused a number of people. Their complaint was that a function with an empty argument list can do little other than return a constant. So, apparently, some further explanation is needed.

In an object-oriented program, every method of an object has at least one argument. In Python and Go, it is explicitly declared and named `self` by convention. In Java, C#, and C++, it is the implicit `this`. This argument, whether explicit or implicit, represents the enclosing scope.

Do functions in C have an implied first argument? Of course they do; it is the enclosing global scope. A C function always has access to that scope, so it plays the same role as `this` or `self`. Thus, every function ever written has an implied argument that represents the enclosing scope.

Therefore, when I say that the easiest argument list to understand is the empty argument list, I am not counting the implied enclosing scope. The easiest argument list to understand is the argument list with no declared arguments.

Do such methods or functions exist? Of course they do. `Stack.pop()`, `file.close()`, `semaphore.release()`, `buffer.flush()`: They are very common and very easy to read.

The next-easiest argument list to understand is the list with one declared argument. The call `f(x)` is pretty easy to understand. There's no chance you'll get the arguments out of order.

Two arguments aren't too hard. Keeping track of which comes first is pretty easy—I mean, there's only one way to get it wrong. You can make this easier by making the two arguments commutative.

Three arguments starts to get difficult. There are five ways to get them into the wrong order. Yes, I know that the IDEs have gotten quite helpful with this; but still, why tempt fate?

Arguments are couplings and are therefore hard enough as it is without having to worry about their order in the calling sequence. What's more, they are harder to read than they are to write. The IDE will give you a lot of help while writing, but much less while reading. So three is usually my limit.

Yes, a limit of three is arbitrary. You might be comfortable with more. You do you, I'll do me.

If I need to pass more than three arguments into a function, I try to find a way to group them. I might be able to put them into an object—after all, if four things are so cohesive that they can be passed, as a unit, into a function, then why aren't they already an object? Or, I might put them into a hash map with nice names to identify them.

One of the simplest ways to reduce argument lists is to create fields in the current class to hold the values that we would otherwise pass as arguments. Indeed, one of the reasons that OO is a powerful tool is that it allows us to create contexts that allow functions to communicate without passing a lot of arguments.

Variadic Arguments

Sometimes we want to write functions that take an unlimited number of arguments. In Java, the `String.format` function is like that.

```
public String format(String format, Object… args)
```

I think it's safe to assume that this function really just takes two arguments: the `format`, and the array of `Object`s named `args`.

More Than Three?

Sometimes there is a rationale behind the ordering. For example, it is not hard to keep the arguments of `distance(x1,y1,x2,y2)` in order. On the other hand, if there is no rationale, then there are 23 ways to get the order wrong. Even with the help of the IDE, that's a fairly large risk. It's an even bigger risk while reading the code, since you are not likely to notice when the arguments are out of order. So usually, if there are more than three arguments, I find some other means.

Keyword Arguments

Some languages allow you to name the arguments in your function calls. For example, in C# you can prefix your arguments with the name of the argument:

```
createRental(movie: m, daysRented: 3);
```

When using a language feature like that, I don’t worry as much about the order of the arguments and might, on occasion, pass more than three.

In some languages, hash maps are so easy to define that they provide a nice way to pass named arguments. For example, in Clojure I can pass a hash map as follows:

```
(create-rental {:movie m, :daysRented 3})
```

Ruby provides a similar bit of syntactic sugar:

```
createRental(movie: m, daysRented: 3)
```

In Ruby, if the `createRental` function is declared to take a hash map, then the arguments will automatically be assumed to be keyword/value pairs.

Flag Arguments

Flag arguments are ugly. Passing a `boolean` into a function is generally careless. It immediately complicates the signature of the function, and loudly proclaims that the function does more than one thing. It does one thing if the flag is `true` and another if the flag is `false`. They are also difficult to read. For example:

```
String monthNames[] = getMonths(false);
```

What does this mean? Are we getting the months or not?

The definition of the function shows us that the `boolean` argument is used to select between long and short month names. But there’s no way to know that by looking at the call. The only way to know it is to read the implementation of the function, or at least the ugly Javadoc.

```
/**
 * Returns an array of month names.
 *
 * @param shortened a flag indicating that shortened
 *                  be returned.
 * @return an array of month names.
 */
public static String[] getMonths(final boolean shorte
  if (shortened) {
    return DATE_FORMAT_SYMBOLS.getShortMonths();
  } else {
    return DATE_FORMAT_SYMBOLS.getMonths();
  }
}
```

Boolean flags can be pretty confusing if you don't know what they mean. That glaring `true` or `false` is indicating some kind of option; but if you don't know what that option is, you'll have to click through to the function and read the implementation—and who wants to do that?

The solution, where possible, is to split the function into two functions: one that does the `true` case and one that does the `false` case.

```
String[] getLongMonthNames() {…}
String[] getShortMonthNames() {…}
```

There are, of course, exceptions to this rule. For example, some objects have binary flags within them that are set with a `setFlag(boolean)` method. I'm not going to complain about them.

Output Arguments

Arguments are most naturally interpreted as inputs to a function. But sometimes people will use arguments to catch *output* values. This is often because they want to return more than one value from a function. For example, in C# you can use `out` arguments, in C you can use pointers, and in C++ you can pass references as follows.

```
void stats(double* ns, int n, double& mean, double& s
  sum=0;
  for (int i=0; i<n; i++)
    sum += ns[i];
  mean=sum/n;
}
```

I try to avoid output arguments like this. They violate the principle of least surprise.[1] People generally don't expect an argument to catch an output, so if they see the following call, they may wonder why we are passing the sum and mean into the function.

---

1. The Law of Least Astonishment, *PL/1 Bulletin*, 1967.

```
stats(list, listSize, listMean, listSum);
```

Reading a line of code like that can cause the reader to do a double take or to stop and wonder what's going on. That's because they expect data to go into functions through the arguments and out from functions through the return value.

In Go, Python, or Ruby,[2] we could return multiple values from a function.

---

2. Ruby returns a mutable array. Caveat emptor.

```
mean, sum = stats(list);
```

In other languages, we could return an object with two elements.

Or, in all cases, we could create two functions: one for `sum` and the other for `mean`. All else being equal, that's probably the best solution.

In the days before object-oriented programming, it was sometimes necessary to have output arguments. However, much of the need for output arguments disappears in OO languages because the object itself can act as the output argument of its methods. In other words, it might be better for `stats.calculate` to be invoked as follows:

```
stats.calculate(list);
double mean = stats.getMean();
double sum = stats.getSum();
```

Error Codes

Go programmers often use the ability to return multiple values from a function to return error codes along with values:

```
f, err := os.Open("someFile")
if err != nil {
    log.Fatal(err)
}
```

This strategy can also be used in languages with `out` arguments.

```
File f = os.open("someFile", out error);
```

So long as this convention is applied narrowly and consistently, I think it's a reasonable convention; especially in a language like Go that does not have exceptions.

Command Query Separation

CQS is a convention that many programmers adopt when and where appropriate. I've used it, and it is often very helpful. It should be considered where practical.

Under the CQS convention, functions either do something or answer something, but not both. Either a function is a command that creates a side effect (i.e., it changes the state of the system), or it is a query that returns some information. Doing both can lead to confusion. Consider, for example, the following function:

```
public boolean set(String attribute, String value);
```

This function sets the value of a named attribute and returns `true` if it is successful and `false` if no such attribute exists. This leads to odd statements like this:

```
if (set("username", "unclebob"))…
```

Imagine this from the point of view of the reader. What does it mean? Is it asking whether the "`username`" attribute was previously set to "`unclebob`"? Or is it asking whether the "`username`" attribute was successfully set to "`unclebob`"? It's hard to infer the meaning from the call because it's not clear whether the word *set* is a verb or an adjective.

The author intended `set` to be a verb, but in the context of the `if` statement, it *feels* like an adjective. We could try to resolve this by renaming the `set` function to `setAndCheckIfExists`, but that doesn't much help the readability of the `if` statement. The real solution is to separate the command from the query so that the ambiguity cannot occur.

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
    …
}
```

But, as I said, there are situations where conforming to CQS is impractical or just inconvenient. For example, consider the following:

```
int top = stack.pop();
```

That simple line of code violates CQS. It queries the top of the stack and also changes the state of the stack.

Is there a cost to violating CQS in convenient situations like this? Yes: concurrency. If the `pop` function is not made atomic with the proper use of semaphores or locks, then it could be possible for a race condition to cause the top of the stack to be returned to two different threads before that object was properly removed from the stack.

Those of you who are C++ programmers will recognize that the C++ standard template library does not allow you to query and alter a container in a single statement. In that library the above `pop` operation requires two statements that conform to CQS:

```
Thing top = myStack.top();
myStack.pop();
```

In C++, the reason for this, aside from the concurrency argument, is that the `Thing` class may override the assignment operator,[3] and it might therefore throw an exception. If the pop operation both changed the stack and returned the top, then when an exception was thrown by the assignment operator it could leave the stack in an unknown state.

---

3. That's right, folks, in C++ you can override the = operator for the assignment of objects.

So, although I cannot claim to use CQS in every circumstance, I take the advice seriously and violate it only after considering the possible outcomes.

Prefer Exceptions to Returning Error Codes

Exceptions can be a tricky matter. In most of our modern languages, they are a reliable way to transmit error information up the calling stack. In older languages, most notably C++, that reliability is not guaranteed. And in some languages, like Go, exceptions just don't exist.

Caveat Emptor

In general, I prefer exceptions to error codes where possible. Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of `if` statements.

```
if (deletePage(page) == E_OK)
```

This does not suffer from verb/adjective confusion but does lead to deeply nested structures. When you return an error code, you create the problem that the caller must deal with the error immediately.

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) ==
      logger.log("page deleted");
      return E_OK;
    } else {
      logger.log("configKey not deleted");
      return E_ERROR;
    }
  } else {
    logger.log("deleteReference from registry failed"
    return E_ERROR;
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

Programmers sometimes use guards to pretend that the structure is not deeply nested. But the lack of those `else` clauses doesn’t mean they aren’t actually there.

```
if (deletePage(page) != E_OK) {
  logger.log("delete failed");
  return E_ERROR;
}

if (registry.deleteReference(page.name) != E_OK) {
  logger.log("deleteReference from registry failed");
  return E_ERROR;
}

if (configKeys.deleteKey(page.name.makeKey()) != E_OK
  logger.log("configKey not deleted");
  return E_ERROR;
}

logger.log("page deleted");
return E_OK;
```

On the other hand, if you use exceptions instead of returned error codes, then the error processing code can be separated from the happy path code and can be simplified:

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

Extract `Try/Catch` **Blocks**

`Try/catch` blocks are ugly in their own right. They confuse the structure of the code and mix error processing with normal processing. So it is better to extract the bodies of the `try` and `catch` blocks out into functions of their own.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) th
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

In the above, the `delete` function is all about error processing. It is easy to understand and then ignore. The `deletePageAndAllReferences` function is all about the processes of fully deleting a `page`. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.

Error Handling Is One Thing

As we have learned, functions should do one thing. Error handling is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword `try` exists in a function, it should be the very first word in the function (after, perhaps, some variable declarations) and that there should be nothing after the `catch`/`finally` blocks.

The Dependency Magnet of Error Codes

If you must return error codes, be careful not to turn them into a dependency magnet.

In C, we used to create the header file *error.h.* It would look like this (though it would usually be much longer):

```
#define E_OK 0
#define E_INVALID 1
#define E_NO_SUCH 2
#define E_LOCKED 3
#define E_OUT_OF_RESOURCES 4
```

Every module in the system would `#include` *error.h.* Anytime a new error code was added to *error.h*, every module in the system had to be recompiled—because the build system used the dates of the source files to determine which were out of date and needed recompilation.

This put a negative pressure on the error codes. Programmers didn't want to add new errors, because then they had to rebuild and redeploy everything, and in those days a rebuild could take hours. So they reused old error codes in “creative” ways instead of adding new ones. And, of course, that led to confusion, disillusion, and heartache.

Modern build environments can be smarter than that. For example, in Java you might use enums in a file named *Error.java*:

```
public enum Error {
   OK,
   INVALID,
   NO_SUCH,
   LOCKED,
   OUT_OF_RESOURCES;
}
```

A file like this is still a dependency magnet because many other classes will import and use it. However, when programmers add a new error code to *Error.java*, only those files that are directly impacted by the change will need to be recompiled and redeployed. Files that simply use the previously existing error codes will not.

So it’s a good idea to know how your build system works, and whether dependency magnets will cause thrashing of the build and require redeployment.

In languages like Go, `Error` is an interface, and any error codes that the programmers want to add implement that interface. That means they can be kept in separate source files and run no risk of becoming a dependency magnet.

Of course, this approach can be used in any language that supports interfaces, or an equivalent kind of dynamic polymorphism. For example, error codes can be independent duck-type classes in Python.

DRY: Don’t Repeat Yourself

We have long felt that duplication of code should be avoided. We’ve heard this advice many times over the years. The authors of *The Pragmatic Programmer*[4] told us this way back in 2000. But they were certainly not the first. Most of Codd’s normal forms for organizing databases are techniques for reducing duplication in the data. Bertrand Meyer told us this way back in 1988 when he wrote about the Single Choice Principle.[5] Suffice it to say that the advice is old.

---

4. [Prag].

---

5. [OOSC].

Simple Repeated Code

So, what does it mean to duplicate or repeat code? There's the obvious case where the same *L* lines of code are repeated over and over in *P* different places. This is often the result of a programmer being a bit overanxious with the mouse. If both *L* and *P* are small, then very little harm is done by the duplication, and the effort to eliminate that duplication may do more harm than good.

For example, if you were to see these two lines repeating twice, it would probably not make much sense to turn them into a function just for the purpose of removing the duplication.

```
count=0
list=[]
```

You might extract them into the `initializeList` function as a convenience to your readers. But that's a different matter. The duplication itself is not very harmful.

What if *L* is likely to grow? For example, what if you need to add a new variable to that initialization?

```
count=0
odds=0
list=[]
```

If *P* is 2, then you have to make that change in two places. That might be a little inconvenient, but it's a minor issue.

On the other hand, if *P* is large, then you'd have to find all the duplicated instances and add the new initialization. If, on the other hand, you had extracted the initialization out into the `initializeList` function, you could make that change in one place.

So, if *P* is large, even if *L* is small, it is probably best to extract the duplication into a single, nicely named function.

What are the odds that a snippet of *L* lines will need modification later? That's probably a function of *L*. The larger *L* is, the larger the odds that the snippet will need to be changed.

Thus, when *L* is large, even if *P* is small, it probably pays to extract the duplication.

We can describe this with this table.

	L is small.	***L*** is large.
P is small.	Maybe extract.	Extract
P is large.	Extract	Extract

Similar Code

Often we will find repeated snippets of code that are not perfectly duplicated. They may differ by one or two constants or variables. Such snippets often tend to have a larger *L*, and so should be extracted.

Here's an example from the first edition of this book—and one that was in the FitNesse[6] project circa 2001. The snippets in bold are similar and could be extracted.

6. http://fitnesse.org; https://github.com/unclebob/fitnesse

```
public class TestableHtml {
  public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
      if (includeSuiteSetup) {
        WikiPage suiteSetup =
          PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_SETUP_NAME, wikiPage
        if (suiteSetup != null) {
          WikiPagePath pagePath =
            suiteSetup.getPageCrawler().getFullPath(s
          String pagePathName = PathParser.render(pag
          buffer.append("!include -setup .")
            .append(pagePathName)
            .append("\n");
        }
      }
      WikiPage setup =
        PageCrawlerImpl.getInheritedPage("SetUp", wik
      if (setup != null) {
        WikiPagePath setupPath =
          setup.getPageCrawler().getFullPath(setup);
        String setupPathName = PathParser.render(setu
        buffer.append("!include - setup.")
          .append(setupPathName)
          .append("\n");
      }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
      WikiPage teardown =
        PageCrawlerImpl.getInheritedPage("TearDown",
      if (teardown != null) {
        WikiPagePath tearDownPath =
          teardown.getPageCrawler().getFullPath(tearc
        String tearDownPathName = PathParser.render(t
        buffer.append("\n")
          .append("!include - teardown.")
          .append(tearDownPathName)
          .append("\n");
      }
```

```
        if (includeSuiteSetup) {
          WikiPage suiteTeardown =
            PageCrawlerImpl.getInheritedPage(
              SuiteResponder.SUITE_TEARDOWN_NAME,
              wikiPage
            );
          if (suiteTeardown != null) {
            WikiPagePath pagePath =
              suiteTeardown.getPageCrawler().getFullPat
            String pagePathName = PathParser.render(pag
            buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
          }
        }
      }
      pageData.setContent(buffer.toString());
      return pageData.getHtml();
    }
  }
```

Usually the extraction of similar snippets like this simply involves adding an argument or two to the extracted function. Let’s try this.

```
  public class TestableHtml {
    public static String testableHtml(
      PageData pageData,
      boolean includeSuiteSetup) throws Exception {
      WikiPage wikiPage = pageData.getWikiPage();
      StringBuffer buffer = new StringBuffer();
      if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
          includeInheritedPage(SuiteResponder.SUITE_SET
            wikiPage, buffer, "!include -setup .");
        }
        includeInheritedPage("SetUp", wikiPage, buffer,
                             "!include -setup .");
      }
      buffer.append(pageData.getContent());
      if (pageData.hasAttribute("Test")) {
        includeInheritedPage("TearDown", wikiPage, buff
                             "\n!include -teardown .");
        if (includeSuiteSetup) {
          includeInheritedPage(SuiteResponder.SUITE_TEA
          wikiPage, buffer, "!include -teardown .");
        }
      }
      pageData.setContent(buffer.toString());
      return pageData.getHtml();
    }

    private static void
    includeInheritedPage(String pageName, WikiPage wiki
                        StringBuffer buffer,
                        String includeString) {
      WikiPage inheritedPage =
        PageCrawlerImpl.getInheritedPage(pageName, wiki
      if (inheritedPage != null) {
        WikiPagePath pagePath =
          inheritedPage.getPageCrawler().getFullPath(in
        String pagePathName = PathParser.render(pagePat
        buffer.append(includeString)
          .append(pagePathName)
          .append("\n");
      }
    }
  }
```

This is clearly better. It vastly improves the flow of the `testableHtml` function and it makes sure that any modifications to `includeInheritedPage` will be in one place instead of four. However, the number of arguments to our new function is larger than I like. So let's move some of those variables into fields.

```
  public class TestableHtml {
    private static StringBuffer buffer;
    private static WikiPage wikiPage;

    public static String testableHtml(
      PageData pageData,
      boolean includeSuiteSetup) throws Exception {
      wikiPage = pageData.getWikiPage();
      buffer = new StringBuffer();
      if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
          includeInheritedPage(SuiteResponder.SUITE_SET
            "!include -setup .");
        }
        includeInheritedPage("SetUp", "!include -setup
      }
      buffer.append(pageData.getContent());
      if (pageData.hasAttribute("Test")) {
        includeInheritedPage("TearDown", "\n!include -t
        if (includeSuiteSetup) {
          includeInheritedPage(SuiteResponder.SUITE_TEA
            "!include -teardown .");
        }
      }
      pageData.setContent(buffer.toString());
      return pageData.getHtml();
    }

    private static void
    includeInheritedPage(String pageName, String incluc
      WikiPage inheritedPage =
        PageCrawlerImpl.getInheritedPage(pageName, wiki
      if (inheritedPage != null) {
        WikiPagePath pagePath =
          inheritedPage.getPageCrawler().getFullPath(ir
        String pagePathName = PathParser.render(pagePat
        buffer.append(includeString)
          .append(pagePathName)
          .append("\n");
      }
    }
  }
```

This is nicer still. If you look closely at the result, you ought to be able to detect more duplication. I'll leave the elimination of that to you.

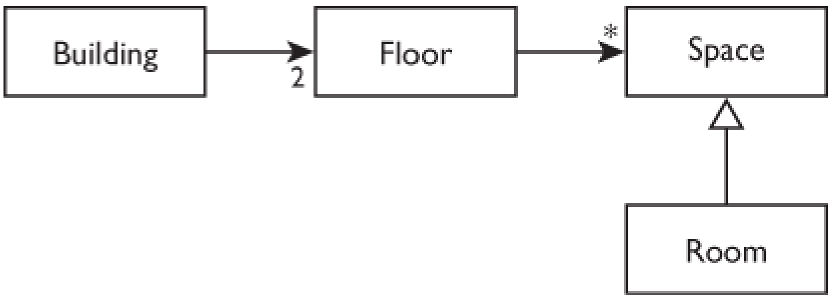
Loop Duplication

Often we find that there are data structures within our application that we must traverse in many different places. The loop that does the traversal is always the same, but the body of that loop is always different.

Nowadays, the popular way to eliminate that duplication is to use a lambda. Older, and sometimes better options are to use the Command pattern, the Strategy pattern, or the TemplateMethod pattern.[7]

7. [GOF95].

As an example of these techniques, consider a problem my team and I faced way back in the late ’90s. We had a data structure that described the internals of a building. Below is a vastly simplified diagram of that data structure.



Our task was to determine several dozen different scorable features of the building by inspecting each of the floors and spaces (and doors and windows and hallways, but never mind that). The scores for these features were determined by a set of C++ classes, each of which had to walk the data structure in order to perform the calculation.

Below is some Ruby code that shows the duplication that could have resulted from all those features.

```
class SpaceFeature
  def score(building)
    spaces = 0
    building.floors.each do |floor|
      floor.spaces.each do |space|
        spaces += 1
      end
    end
    spaces
  end
end

class AreaFeature
  def score(building)
    area = 0
    building.floors.each do |floor|
      floor.spaces.each do |space|
        area += space.width * space.length
      end
    end
    area
  end
end
```

You can see the problem. The nested loops are identical, but the body of the loop is different in each case. Now imagine this repeated several dozen times with a much more complicated data structure and traversal loop.

We can address this in modern languages by using a lambda (sometimes called a *block*) as follows.

```
  class Feature
    def do_score(building, block)
      value = 0
      building.floors.each do |floor|
        floor.spaces.each do |space|
          value = block.call(value, floor, space)
        end
      end
      value
    end
  End

  class SpaceFeature < Feature
    def score(building)
      do_score(building, ->(value, floor, space) {retur
    end
  end

  class AreaFeature < Feature
    def score(building)
      do_score(building, ->(value, floor, space)
                         {return value + (space.width *
    end
  end
```

Now the loop that traverses the data structure exists only in the `Feature` class. If any changes were made to that data structure, the odds are that only that class would need to be changed. The specific features, like

`SpaceFeature` and `AreaFeature`, would likely remain unaffected by such changes.

Another technique would have been to use the TemplateMethod pattern.

```
class Feature
  def score(building)
    value = 0
    building.floors.each do |floor|
      floor.spaces.each do |space|
        value = score_space(value, floor, space)
      end
    end
    value
  end

  def score_space(value, floor, space)
    raise "Not Implemented"
  end
end

class SpaceFeature < Feature
  def score_space(value, floor, space)
    value+1
  end
end

class AreaFeature < Feature
  def score_space(value, floor, space)
    value + (space.width * space.length)
  end
end
```

I actually like this better than the lambda solution—and it's the one we used way back in the '90s in C++. Back then we called it "Write a Loop Once" because the *Design Patterns* book had not yet been published.

I'll leave you to check out the Strategy and Command patterns for yourself.

Accidental versus Essential Duplication

So far we have been talking about essential duplication. But sometimes two stretches of code appear to be duplicates, when in fact they are not. Those are accidental duplicates.

Essential duplicates change together. When one changes, they all change. They evolve together.

Accidental duplicates do not change together. They evolve apart from each other. So it is very important not to combine them into a common function.

You can tell an accidental duplication from an essential duplication by considering the Single Responsibility Principle (SRP).[8] If two similar stretches of code are in two modules that are responsible to different actors, then they are likely to evolve separately as those actors ask for different changes. If you extract them into a common function, then you are likely to break the system for one actor while you try to satisfy the other.

---

8. See Chapter 19, "The SOLID Principles."

On the other hand, if two similar stretches of code are in modules responsible to the same actor, then they are likely to evolve together. If you forget to change one of those stretches, then you will break the system for that actor.

Side Effects

In the years since the first edition of this book was written, and with the rise of functional programming, the definition of the term *side effect* has gone through a transition. Twenty years ago most programmers thought that a function with a side effect had a primary purpose but then also did something else on the side. The example I used in the first edition was a function that checked the validity of a login password but then also initialized the current session if the password was found to be valid.

Since then, the definition has been generalized. Nowadays, we define a side effect to be any state change that outlives the function, even if that state change is the primary intent of the function. In other words, a function with a side effect is impure.

So, for example, if `stack.pop()` removes the first item on the top of the `stack`, then it leaves that `stack` in an altered state; the alteration of that state is a side effect.

Side effects create complications for software systems because they introduce an insidious form of coupling called *temporal coupling*—coupling in time. A statement with a side effect separates the code above it from the code below it with a change of state. That change of state means that the order of those lines of code must be preserved.

So you can't close a file until you've opened it, and you can't open it if it's already open. You can't release a semaphore until you've seized it. You can't free a block before you've allocated it. You can't release a graphics context until you've taken possession of it. Et cetera.

Side effect functions often come in pairs, like `open/close`, `seize/release`, or `malloc/free`. One creates the side effect and the other undoes it. So side effects are like The Sith: "always two there are."

We Are Bad at This

The proof that we are bad at managing side effects is the fact that almost all of our major languages now feature *garbage collection*. Garbage collection has been added to our languages to cure the fact that we are just so bloody awful at balancing `malloc` with `free`. We've screwed this up so often, and relegated so many systems to be rebooted every midnight, that we finally created this horrible hack of a feature as a desperate measure to relieve us from the burden of cleaning up memory side effects.

How many languages do you know that have gone to great lengths to help programmers manage the ownership of memory blocks? How many have used reference counting, or RAII,[9] or God knows what? And why? Because we poor, feeble programmers cannot seem to figure out how to return the books we've borrowed from the library.

9. Resource Acquisition Is Initialization—the key insight behind the old `auto_ptr` (now `unique_ptr` and `shared_ptr`) facilities in C++.

The problem is that facilities like garbage collection only address one of the many Sith-like creatures that we must deal with. We don't have garbage collection for file descriptors, or graphics contexts, or semaphores, or. … Well, I think you get the point. We are bad at this.

When order matters, temporal couplings can leak out into other parts of the code, creating situations that are as difficult to debug as a memory leak. I'm sure you've had the experience of debugging a misbehaving system only to find that if you swap the order of two adjacent function calls, the system suddenly works—and you have no idea why.

Temporal couplings created by side effects are at the root of all concurrent update problems, all reentrancy problems, all race conditions, and nearly all initialization and finalization problems. Temporal couplings are big trouble. So how do we avoid them?

Functional Languages

You can avoid a lot of side effects by using a functional language. This is not a cure-all. It is still possible to create temporal couplings with functional languages—but most functional languages have facilities that make it very obvious when you are creating them.

The primary mechanism of functional languages that suppresses side effects is the lack of an assignment statement. In a functional language, you are not allowed to change the state of a variable. In fact, there are no variables in functional languages. Named values cannot be varied.

Some of you may have used a functional language before, and may now be ripping this page into little pieces out of frustration. Please relax. Let me explain.

Functional languages allow names to be assigned to values. The source code may look a lot like an assignment statement. However, in a strictly functional language, while you may be able to reassign the name, you cannot overwrite the value.

For example, I found the following in a Clojure program I wrote a while ago.

```
(defn update-clouds-age [ms world]
  (let [clouds (:clouds world)
        decay (Math/pow glc/cloud-decay-rate ms)
        clouds (map #(update % :concentration * decay
        clouds (filter #(> (:concentration %) 1) clou
        clouds (doall clouds)]
    (assoc world :clouds clouds)))
```

The *name* `clouds` has been assigned and reassigned several times. However, the previous values have not been overwritten—they still exist. The name `clouds` does not represent a slot in memory that has been overwritten by the reassignments. Rather, the four versions of `clouds` refer to four different blocks of memory that retain their values. The name `clouds` does not represent a variable, because nothing has varied.

Now, go get some tape and repair this page.

Why would functional languages prevent overwriting values? Because overwriting is a side effect. It separates the lines above it and the lines below it by a change of state. It creates a temporal coupling. The order matters.

Of course, you can still create code with side effects when using functional languages. Most of the time this is because the language has special facilities for doing so. For example, Clojure has a special data type called an `atom` that represents a *variable* whose value can be changed using a set of special functions.

And, of course, there are still operating system facilities that have side effects. Files still need to be opened and closed, for example.

More generally, it is possible to simulate any system with a functional language—including a system that supports side effects. For example, I could write a Lua execution environment in even the strictest functional language, and then write programs in Lua that had blatant side effects.

Now, this is not a book about functional programming, and so I'm not going to continue this at any length. Suffice it to say that using a functional language can help you manage side effects; but it cannot, and will not, eliminate them.

If you want a much more in-depth discussion of these matters, I suggest you read my book *Functional Design*.[10]

---

10. [Functional Design].

Object-Oriented Languages

One of the big complaints that has lately been levied against object-oriented programming languages (OOPLs) is that they promote side effects. It's easy to see why this complaint has gotten traction. Objects are containers of state, and the methods of objects often change their state, leading to side effects and temporal couplings.

However, what is not often argued is the fact that OOPLs have very good facilities to suppress and hide side effects. For instance, consider this simplistic and contrived example in C++.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class LineCounter {
  public:
   static int count(const char* fileName);
  private:
   static int nLines;
   static ifstream file;

   static void initialize(const char* fileName);
   static void countTheLines();
   static int finalize();
};

int LineCounter::nLines;
ifstream LineCounter::file;
```

```
void LineCounter::initialize(const char* fileName) {
  nLines = 0;
  file.open(fileName);
}

void LineCounter::countTheLines() {
  string line;
  while (getline (file, line))
    nLines++;
}

int LineCounter::finalize() {
  file.close();
  return nLines;
}

int LineCounter::count(const char* fileName) {
  initialize(fileName);
  countTheLines();
  return finalize();
}
```

What I want you to notice in this example is that every one of the private methods in the `LineCounter` class has a side effect; however, the sole public method, which calls them, does not. This means that none of the users of `LineCounter` are exposed to a temporal coupling. Assuming this is a single-threaded application, the side effects are entirely hidden.

In this example, we have used the `LineCounter` as a protected namespace. There is no `LineCounter` object. The class is just a place that allows us to hide the variables and functions that might cause temporal couplings.

Notice that I added the single-threaded constraint. In a multithreaded environment, those static variables could be exposed to concurrent update issues. We could solve this by surrounding the `count` function with a semaphore. Or we could solve it by turning the `LineCounter` class into a real object by getting rid of the static variables and functions and forcing users to create an instance before calling `count` .

The point is that the facilities of most OOPLs provide a handy way to hide side effects behind encapsulation boundaries, thus preventing temporal couplings from leaking out into the application at large.

Structured Programming

In 1968, Edsger Dijkstra wrote a paper that stirred up the programming community for the better part of a decade. Back in those days the ideas he expressed in that paper were deeply controversial and hotly debated. Some felt that Dijkstra was a fool. Others thought him a god.

But by the time the ’80s rolled around, Dijkstra’s simple idea of Structured Programming was well accepted. By the ’90s, it was hard to find a language that allowed any other style.

The paper, of course, was “Go To Statement Considered Harmful.” It was published as a letter to the editor in the March 1968 issue of the *Communications of the ACM*. The title was given to it by the editor at the time: Niklaus Wirth.[11] And that paper changed the world—of programming.

---

[11]. The inventor of the Pascal language.

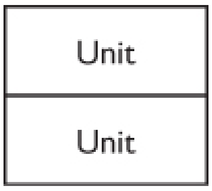
The idea behind the paper is very simple. All programs should be written using only three control structures: sequence, selection, and iteration.

This was revolutionary at the time because most programming languages used jumps (`GOTO`) as the primary, or even the sole, control structure, and programmers were used to jumping hither and yon within their programs. Such programs often became rats’ nests of control flow.

Dijkstra said that all programs should be ordered as recursively applied units with a single entry and a single exit. The units came in three varieties.

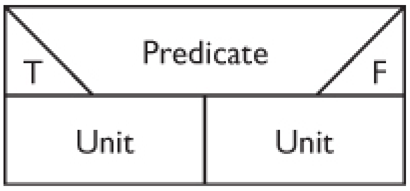
Sequence

A sequence is two (or more) units that are executed one after the other. Each such unit has a single entry at the top and a single exit at the bottom. A unit might be as simple as a single statement, like `i=0;` , or it might be a function call, or its own sequence of units.



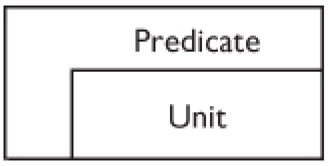
Selection

A selection is a boolean predicate that selects one of two units below it. The predicate is any boolean expression, such as `i<0`.

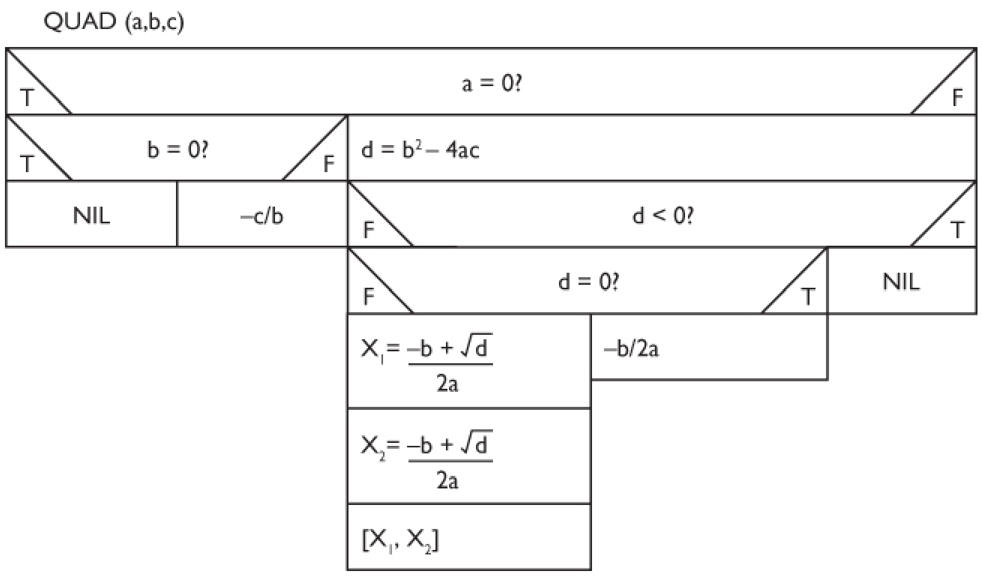


Iteration

An iteration is a predicate that executes the unit below it until the predicate is false.



These units can be recursively applied into a program as follows.



This style of diagram is called a *Nassi–Schneiderman flowchart*. The diagram itself depicts the flow of control through a quadratic equation solver.

Dijkstra's rationale for restricting control flow to these structures was based on the argument that such structures are relatively easy to reason about. You can follow each path and evaluate the outcomes. However, if `goto` statements are allowed to transfer control from any point in the program to any other point in the program, then following the paths, and therefore reasoning about the problem, becomes impractical.

Nowadays, our languages restrict control flow for us. Most of us don't use languages that have a `goto` statement. Those of us who do, tend not to use it.

When writing clean functions, it is helpful to keep this recursive application of the three structures in mind.

This Is Too Much to Constantly Keep in Mind

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, and then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code, and temporal couplings. But I also have a suite of unit tests that cover every one of those clumsy lines of code.

So then I massage and refine that code, splitting out functions, changing names, eliminating duplication, and making my functions purer. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.

In the end, I wind up with functions that follow the attributes and heuristics I've laid down in the last two chapters. I don't write them that way to start. I don't think anyone could.

First, make it work. Then, make it right.

Conclusion

Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and contexts are the nouns. It is a very old truth that the art of programming is, and has always been, the art of language design.

Master programmers think of systems as stories to be told rather than programs to be written. They use the facilities of their chosen programming language to construct a much richer and more expressive language that can be used to tell that story. Part of that domain-specific language is the hierarchy of functions that describe all the actions that take place within that system. In an artful act of recursion, those actions are written to use the very domain-specific language they define to tell their own small part of the story.

The last two chapters have been about the mechanics of writing functions well. If you follow the ideas herein, your functions will be short, well named, and nicely organized. But never forget that your real goal is to tell the story of the system, and that the functions you write need to fit cleanly together into a clear and precise language to help you with that telling.