2

Meaningful Names

by Tim Ottinger

Introduction

Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source files and the directories that contain them. We name our jar files and war files and ear files. We name and name and name. Because we do

so much of it, we'd better do it well. What follows are some simple rules for creating good names.

Use Intention-Revealing Names

It is easy to say that names should reveal intent. What we want to impress upon you is that we are *serious* about this. Choosing good names takes time but saves more than it takes. So take care with your names and change them when you find better ones. Everyone who reads your code (including you) will be happier if you do.

The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

```
int d; // elapsed time in days
```

The name `d` reveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Choosing names that reveal intent can make it much easier to understand and change code. What is the purpose of this code?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Why is it hard to tell what this code is doing? There are no complex expressions. Spacing and indentation are reasonable. There are only three variables and two constants mentioned. There aren't even any fancy classes or polymorphic methods, just a list of arrays (or so it seems).

The problem isn't the simplicity of the code but the *implicity* of the code (to coin a phrase): the degree to which the context is not explicit in the code itself. The code implicitly requires that we know the answers to questions such as:

1. What kinds of things are in `theList`?
2. What is the significance of the zeroth subscript of an item in `theList`?
3. What is the significance of the value `4`?
4. How would I use the list being returned?

The answers to these questions are not present in the code sample, *but they could have been*. Say that we're working in a mine sweeper game. We find that the board is a list of cells called `theList`. Let's rename that to `gameBoard`.

Each cell on the board is represented by a simple array. We further find that the zeroth subscript is the location of a status value and that a status value of 4 means "flagged." Just by giving these concepts names we can improve the code considerably:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit.

We can go further and write a simple class for cells instead of using an array of `int`s. It can include an intention-revealing function (call it `isFlagged`) to hide the magic numbers. It results in a new version of the function:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

With these simple name changes, it's not difficult to understand what's going on. This is the power of choosing good names.

Avoid Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning. For example, `hp`, `aix`, and `sco` would be poor variable names because they are the names of Unix platforms or variants. Even if you are coding a hypotenuse and `hp` looks like a good abbreviation, it could be disinformative.

Do not refer to a grouping of accounts as an `accountList` unless it's actually a `List`. The word list means something specific to programmers. If the container holding the accounts is not actually a `List`, it may lead to false conclusions.[1] So `accountGroup` or `bunchOfAccounts` or just plain `accounts` would be better.

1. As we'll see later on, even if the container *is* a `List`, it's probably better not to encode the container type into the name.

Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, `XYZControllerForEfficientStorageOfStrings`? The words have frightfully similar shapes.

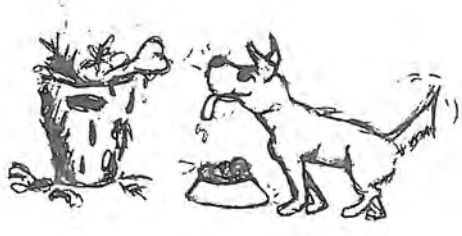
Spelling similar concepts similarly is *information*. Using inconsistent spellings is *disinformation*. With modern Java environments we enjoy automatic code completion. We write a few characters of a name and press some hotkey combination (if that) and are rewarded with a list of possible completions for that name. It is very helpful if names for very similar things sort together alphabetically and if the differences are very obvious, because the developer is likely to pick an object by name without seeing your copious comments or even the list of methods supplied by that class.

A truly awful example of disinformative names would be the use of lower-case `L` or uppercase `O` as variable names, especially in combination. The problem, of course, is that they look almost entirely like the constants one and zero, respectively.

```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```

The reader may think this a contrivance, but we have examined code where such things were abundant. In one case the author of the code suggested using a different font so that the differences were more obvious, a solution that would have to be passed down to all future developers as oral tradition or in a written document. The problem is conquered with finality and without creating new work products by a simple renaming.

Make Meaningful Distinctions



Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. For example, because you can't use the same name to refer to two different things in the same scope, you might be tempted to change one name in an arbitrary way. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.[2]

It is not sufficient to add number series or noise words, even though the compiler is satisfied. If names must be different, then they should also mean something different.

---

2. Consider, for example, the truly hideous practice of creating a variable named `klass` just because the name `class` was used for something else.

Number-series naming (a1, a2, .. aN) is the opposite of intentional naming. Such names are not disinformative—they are noninformative; they provide no clue to the author's intention. Consider:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

This function reads much better when source and destination are used for the argument names.

Noise words are another meaningless distinction. Imagine that you have a Product class. If you have another called ProductInfo or ProductData, you have made the names different without making them mean anything different. Info and Data are indistinct noise words like a, an, and the.

Note that there is nothing wrong with using prefix conventions like a and the so long as they make a meaningful distinction. For example you might use a for all local variables and the for all function arguments.[3] The problem comes in when you decide to call a variable theZork because you already have another variable named zork.

Noise words are redundant. The word variable should never appear in a variable name. The word table should never appear in a table name. How is NameString better than Name? Would a Name ever be a floating point number? If so, it breaks an earlier rule about disinformation. Imagine finding one class named Customer and another named CustomerObject. What should you understand as the distinction? Which one will represent the best path to a customer's payment history?

There is an application we know of where this is illustrated. we've changed the names to protect the guilty, but here's the exact form of the error:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

How are the programmers in this project supposed to know which of these functions to call?

In the absence of specific conventions, the variable moneyAmount is indistinguishable from money, customerInfo is indistinguishable from customer, accountData is indistinguishable from account, and theMessage is indistinguishable from message. Distinguish names in such a way that the reader knows what the differences offer.

Use Pronounceable Names

Humans are good at words. A significant part of our brains is dedicated to the concept of words. And words are, by definition, pronounceable. It would be a shame not to take

3. Uncle Bob used to do this in C++ but has given up the practice because modern IDEs make it unnecessary.

advantage of that huge portion of our brains that has evolved to deal with spoken language. So make your names pronounceable.

If you can't pronounce it, you can't discuss it without sounding like an idiot. "Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?" This matters because programming is a social activity.

A company I know has `genymdhms` (generation date, year, month, day, hour, minute, and second) so they walked around saying "gen why emm dee aich emm ess". I have an annoying habit of pronouncing everything as written, so I started saying "gen-yah-mudda-hims." It later was being called this by a host of designers and analysts, and we still sounded silly. But we were in on the joke, so it was fun. Fun or not, we were tolerating poor naming. New developers had to have the variables explained to them, and then they spoke about it in silly made-up words instead of using proper English terms. Compare

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

to

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Intelligent conversation is now possible: "Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"

Use Searchable Names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

One might easily grep for `MAX_CLASSES_PER_STUDENT`, but the number 7 could be more troublesome. Searches may turn up the digit as part of file names, other constant definitions, and in various expressions where the value is used with different intent. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer's search.

Likewise, the name `e` is a poor choice for any variable for which a programmer might need to search. It is the most common letter in the English language and likely to show up in every passage of text in every program. In this regard, longer names trump shorter names, and any searchable name trumps a constant in code.

My personal preference is that single-letter names can ONLY be used as local variables inside short methods. *The length of a name should correspond to the size of its scope*

[N5]. If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name. Once again compare

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

to

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Note that `sum`, above, is not a particularly useful name but at least is searchable. The intentionally named code makes for a longer function, but consider how much easier it will be to find `WORK_DAYS_PER_WEEK` than to find all the places where 5 was used and filter the list down to just the instances with the intended meaning.

Avoid Encodings

We have enough encodings to deal with without adding more to our burden. Encoding type or scope information into names simply adds an extra burden of deciphering. It hardly seems reasonable to require each new employee to learn yet another encoding "language" in addition to learning the (usually considerable) body of code that they'll be working in. It is an unnecessary mental burden when trying to solve a problem. Encoded names are seldom pronounceable and are easy to mis-type.

Hungarian Notation

In days of old, when we worked in name-length-challenged languages, we violated this rule out of necessity, and with regret. Fortran forced encodings by making the first letter a code for the type. Early versions of BASIC allowed only a letter plus one digit. Hungarian Notation (HN) took this to a whole new level.

HN was considered to be pretty important back in the Windows C API, when everything was an integer handle or a long pointer or a `void` pointer, or one of several implementations of "string" (with different uses and attributes). The compiler did not check types in those days, so the programmers needed a crutch to help them remember the types.

In modern languages we have much richer type systems, and the compilers remember and enforce the types. What's more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they're using.

Java programmers don't need type encoding. Objects are strongly typed, and editing environments have advanced such that they detect a type error long before you can run a compile! So nowadays HN and other forms of type encoding are simply impediments. They make it harder to change the name or type of a variable, function, or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.

```
PhoneNumber phoneString;
// name not changed when type changed!
```

Member Prefixes

You also don't need to prefix member variables with `m_` anymore. Your classes and functions should be small enough that you don't need them. And you should be using an editing environment that highlights or colorizes members to make them distinct.

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Besides, people quickly learn to ignore the prefix (or suffix) to see the meaningful part of the name. The more we read the code, the less we see the prefixes. Eventually the prefixes become unseen clutter and a marker of older code.

Interfaces and Implementations

These are sometimes a special case for encodings. For example, say you are building an ABSTRACT FACTORY for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? `IShapeFactory` and `ShapeFactory`? I prefer to leave interfaces unadorned. The preceding `I`, so common in today's legacy wads, is a distraction at best and too much information at worst. I don't want my users knowing that I'm handing them an interface. I just want them to know that it's a `ShapeFactory`. So if I must encode either the interface or the implementation, I choose the implementation. Calling it `ShapeFactoryImp`, or even the hideous `CShapeFactory`, is preferable to encoding the interface.

Avoid Mental Mapping

Readers shouldn't have to mentally translate your names into other names they already know. This problem generally arises from a choice to use neither problem domain terms nor solution domain terms.

This is a problem with single-letter variable names. Certainly a loop counter may be named `i` or `j` or `k` (though never `l`!) if its scope is very small and no other names can conflict with it. This is because those single-letter names for loop counters are traditional. However, in most other contexts a single-letter name is a poor choice; it's just a place holder that the reader must mentally map to the actual concept. There can be no worse reason for using the name `c` than because `a` and `b` were already taken.

In general programmers are pretty smart people. Smart people sometimes like to show off their smarts by demonstrating their mental juggling abilities. After all, if you can reliably remember that `r` is the lower-cased version of the url with the host and scheme removed, then you must clearly be very smart.

One difference between a smart programmer and a professional programmer is that the professional understands that *clarity is king*. Professionals use their powers for good and write code that others can understand.

Class Names

Classes and objects should have noun or noun phrase names like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Avoid words like `Manager`, `Processor`, `Data`, or `Info` in the name of a class. A class name should not be a verb.

Method Names

Methods should have verb or verb phrase names like `postPayment`, `deletePage`, or `save`. Accessors, mutators, and predicates should be named for their value and prefixed with `get`, `set`, and `is` according to the javabean standard.[4]

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

When constructors are overloaded, use static factory methods with names that describe the arguments. For example,

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

is generally better than

```
Complex fulcrumPoint = new Complex(23.0);
```

Consider enforcing their use by making the corresponding constructors private.

---

4. http://java.sun.com/products/javabeans/docs/spec.html

Don't Be Cute

If names are too clever, they will be memorable only to people who share the author's sense of humor, and only as long as these people remember the joke. Will they know what the function named `HolyHandGrenade` is supposed to do? Sure, it's cute, but maybe in this case `DeleteItems` might be a better name. Choose clarity over entertainment value.

Cuteness in code often appears in the form of colloquialisms or slang. For example, don't use the name `whack()` to mean `kill()`. Don't tell little culture-dependent jokes like `eatMyShorts()` to mean `abort()`.

Say what you mean. Mean what you say.

Pick One Word per Concept

Pick one word for one abstract concept and stick with it. For instance, it's confusing to have `fetch`, `retrieve`, and `get` as equivalent methods of different classes. How do you remember which method name goes with which class? Sadly, you often have to remember which company, group, or individual wrote the library or class in order to remember which term was used. Otherwise, you spend an awful lot of time browsing through headers and previous code samples.

Modern editing environments like Eclipse and IntelliJ-provide context-sensitive clues, such as the list of methods you can call on a given object. But note that the list doesn't usually give you the comments you wrote around your function names and parameter lists. You are lucky if it gives the parameter *names* from function declarations. The function names have to stand alone, and they have to be consistent in order for you to pick the correct method without any additional exploration.

Likewise, it's confusing to have a `controller` and a `manager` and a `driver` in the same code base. What is the essential difference between a `DeviceManager` and a `ProtocolController`? Why are both not `controllers` or both not `managers`? Are they both Drivers really? The name leads you to expect two objects that have very different type as well as having different classes.

A consistent lexicon is a great boon to the programmers who must use your code.

Don't Pun

Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.

If you follow the "one word per concept" rule, you could end up with many classes that have, for example, an `add` method. As long as the parameter lists and return values of the various `add` methods are semantically equivalent, all is well.

However one might decide to use the word `add` for "consistency" when he or she is not in fact adding in the same sense. Let's say we have many classes where `add` will create a new value by adding or concatenating two existing values. Now let's say we are writing a new class that has a method that puts its single parameter into a collection. Should we call this method `add`? It might seem consistent because we have so many other `add` methods, but in this case, the semantics are different, so we should use a name like `insert` or `append` instead. To call the new method `add` would be a pun.

Our goal, as authors, is to make our code as easy as possible to understand. We want our code to be a quick skim, not an intense study. We want to use the popular paperback model whereby the author is responsible for making himself clear and not the academic model where it is the scholar's job to dig the meaning out of the paper.

Use Solution Domain Names

Remember that the people who read your code will be programmers. So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth. It is not wise to draw every name from the problem domain because we don't want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name.

The name `AccountVisitor` means a great deal to a programmer who is familiar with the VISITOR pattern. What programmer would not know what a `JobQueue` was? There are lots of very technical things that programmers have to do. Choosing technical names for those things is usually the most appropriate course.

Use Problem Domain Names

When there is no "programmer-eese" for what you're doing, use the name from the problem domain. At least the programmer who maintains your code can ask a domain expert what it means.

Separating solution and problem domain concepts is part of the job of a good programmer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain.

Add Meaningful Context

There are a few names which are meaningful in and of themselves—most are not. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces. When all else fails, then prefixing the name may be necessary as a last resort.

Imagine that you have variables named `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `zipcode`. Taken together it's pretty clear that they form an address. But what if you just saw the `state` variable being used alone in a method? Would you automatically infer that it was part of an address?

You can add context by using prefixes: `addrFirstName`, `addrLastName`, `addrState`, and so on. At least readers will understand that these variables are part of a larger structure. Of course, a better solution is to create a class named `Address`. Then, even the compiler knows that the variables belong to a bigger concept.

Consider the method in Listing 2-1. Do the variables need a more meaningful context? The function name provides only part of the context; the algorithm provides the rest. Once you read through the function, you see that the three variables, `number`, `verb`, and `pluralModifier`, are part of the "guess statistics" message. Unfortunately, the context must be inferred. When you first look at the method, the meanings of the variables are opaque.

Listing 2-1
Variables with unclear context.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```

The function is a bit too long and the variables are used throughout. To split the function into smaller pieces we need to create a `GuessStatisticsMessage` class and make the three variables fields of this class. This provides a clear context for the three variables. They are *definitively* part of the `GuessStatisticsMessage`. The improvement of context also allows the algorithm to be made much cleaner by breaking it into many smaller functions. (See Listing 2-2.)

Listing 2-2
Variables have a context.

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Don't Add Gratuitous Context

In an imaginary application called "Gas Station Deluxe," it is a bad idea to prefix every class with `GSD`. Frankly, you are working against your tools. You type `G` and press the completion key and are rewarded with a mile-long list of every class in the system. Is that wise? Why make it hard for the IDE to help you?

Likewise, say you invented a `MailingAddress` class in `GSD`'s accounting module, and you named it `GSDAccountAddress`. Later, you need a mailing address for your customer contact application. Do you use `GSDAccountAddress`? Does it sound like the right name? Ten of 17 characters are redundant or irrelevant.

Shorter names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary.

The names `accountAddress` and `customerAddress` are fine names for instances of the class `Address` but could be poor names for classes. `Address` is a fine name for a class. If I need to differentiate between MAC addresses, port addresses, and Web addresses, I might consider `PostalAddress`, `MAC`, and `URI`. The resulting names are more precise, which is the point of all naming.

Final Words

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. This is a teaching issue rather than a technical, business, or management issue. As a result many people in this field don't learn to do it very well.

People are also afraid of renaming things for fear that some other developers will object. We do not share that fear and find that we are actually grateful when names change (for the better). Most of the time we don't really memorize the names of classes and methods. We use the modern tools to deal with details like that so we can focus on whether the code reads like paragraphs and sentences, or at least like tables and data structure (a sentence isn't always the best way to display data). You will probably end up surprising someone when you rename, just like you might with any other code improvement. Don't let it stop you in your tracks.

Follow some of these rules and see whether you don't improve the readability of your code. If you are maintaining someone else's code, use refactoring tools to help resolve these problems. It will pay off in the short term and continue to pay in the long run.

3

Functions

In the early days of programming we composed our systems of routines and subroutines. Then, in the era of Fortran and PL/1 we composed our systems of programs, subprograms, and functions. Nowadays only the function survives from those early days. Functions are the first line of organization in any program. Writing them well is the topic of this chapter.

Consider the code in Listing 3-1. It's hard to find a long function in FitNesse,[1] but after a bit of searching I came across this one. Not only is it long, but it's got duplicated code, lots of odd strings, and many strange and inobvious data types and APIs. See how much sense you can make of it in the next three minutes.

Listing 3-1
HtmlUtil.java (FitNesse 20070619)

```
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(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

1. An open-source testing tool. www.fitnese.org

Listing 3-1 (continued)
HtmlUtil.java (FitNesse 20070619)

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

Do you understand the function after three minutes of study? Probably not. There's too much going on in there at too many different levels of abstraction. There are strange strings and odd function calls mixed in with doubly nested `if` statements controlled by flags.

However, with just a few simple method extractions, some renaming, and a little restructuring, I was able to capture the intent of the function in the nine lines of Listing 3-2. See whether you can understand *that* in the next 3 minutes.

Listing 3-2
HtmlUtil.java (refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

Unless you are a student of FitNesse, you probably don't understand all the details. Still, you probably understand that this function performs the inclusion of some setup and teardown pages into a test page and then renders that page into HTML. If you are familiar with JUnit,[2] you probably realize that this function belongs to some kind of Web-based testing framework. And, of course, that is correct. Divining that information from Listing 3-2 is pretty easy, but it's pretty well obscured by Listing 3-1.

So what is it that makes a function like Listing 3-2 easy to read and understand? How can we make a function communicate its intent? What attributes can we give our functions that will allow a casual reader to intuit the kind of program they live inside?

Small!

The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that.* This is not an assertion that I can justify. I can't provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I've written several nasty 3,000-line abominations. I've written scads of functions in the 100 to 300 line range. And I've written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small.

In the eighties we used to say that a function should be no bigger than a screen-full. Of course we said that at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. Nowadays with a cranked-down font and a nice big monitor, you can fit 150 characters on a line and a 100 lines or more on a screen. Lines should not be 150 characters long. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.

How short should a function be? In 1999 I went to visit Kent Beck at his home in Oregon. We sat down and did some programming together. At one point he showed me a cute little Java/Swing program that he called *Sparkle*. It produced a visual effect on the screen very similar to the magic wand of the fairy godmother in the movie Cinderella. As you moved the mouse, the sparkles would drip from the cursor with a satisfying scintillation, falling to the bottom of the window through a simulated gravitational field. When Kent showed me the code, I was struck by how small all the functions were. I was used to functions in Swing programs that took up miles of vertical space. Every function in *this* program was just two, or three, or four lines long. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. *That's* how short your functions should be![3]

---

2. An open-source unit-testing tool for Java. www.junit.org
3. I asked Kent whether he still had a copy, but he was unable to find one. I searched all my old computers too, but to no avail. All that is left now is my memory of that program.

How short should your functions be? They should usually be shorter than Listing 3-2! Indeed, Listing 3-2 should really be shortened to Listing 3-3.

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Blocks and Indenting

This implies that the blocks within `if` statements, `else` statements, `while` statements, and so on should be one line long. Probably that line should be a function call. Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.

This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easier to read and understand.

Do One Thing

It should be very clear that Listing 3-1 is doing lots more than one thing. It's creating buffers, fetching pages, searching for inherited pages, rendering paths, appending arcane strings, and generating HTML, among other things. Listing 3-1 is very busy doing lots of different things. On the other hand, Listing 3-3 is doing one simple thing. It's including setups and teardowns into test pages.

The following advice has appeared in one form or another for 30 years or more.

> ***FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.***

The problem with this statement is that it is hard to know what "one thing" is. Does Listing 3-3 do one thing? It's easy to make the case that it's doing three things:

1. Determining whether the page is a test page.
2. If so, including setups and teardowns.
3. Rendering the page in HTML.

So which is it? Is the function doing one thing or three things? Notice that the three steps of the function are one level of abstraction below the stated name of the function. We can describe the function by describing it as a brief *TO*[4] paragraph:

> *TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.*

If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction.

It should be very clear that Listing 3-1 contains steps at many different levels of abstraction. So it is clearly doing more than one thing. Even Listing 3-2 has two levels of abstraction, as proved by our ability to shrink it down. But it would be very hard to meaningfully shrink Listing 3-3. We could extract the `if` statement into a function named `includeSetupsAndTeardownsIfTestPage`, but that simply restates the code without changing the level of abstraction.

So, another way to know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation [G34].

Sections within Functions

Look at Listing 4-7 on page 71. Notice that the `generatePrimes` function is divided into sections such as *declarations*, *initializations*, and *sieve*. This is an obvious symptom of doing more than one thing. Functions that do one thing cannot be reasonably divided into sections.

One Level of Abstraction per Function

In order to make sure our functions are doing "one thing," we need to make sure that the statements within our function are all at the same level of abstraction. It is easy to see how Listing 3-1 violates this rule. There are concepts in there that are at a very high level of abstraction, such as `getHtml()`; others that are at an intermediate level of abstraction, such as: `String pagePathName = PathParser.render(pagePath);` and still others that are remarkably low level, such as: `.append("\n")`.

Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. Worse,

---

4. The LOGO language used the keyword "TO" in the same way that Ruby and Python use "def." So every function began with the word "TO." This had an interesting effect on the way functions were designed.

like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.

Reading Code from Top to Bottom: *The Stepdown Rule*

We want the code to read like a top-down narrative.[5] We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. I call this *The Stepdown Rule*.

To say this differently, we want to be able to read the program as though it were a set of *TO* paragraphs, each of which is describing the current level of abstraction and referencing subsequent *TO* paragraphs at the next level down.

> *To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
>
> *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
>
> *To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.*
>
> *To search the parent. . .*

It turns out to be very difficult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do "one thing." Making the code read like a top-down set of *TO* paragraphs is an effective technique for keeping the abstraction level consistent.

Take a look at Listing 3-7 at the end of this chapter. It shows the whole `testableHtml` function refactored according to the principles described here. Notice how each function introduces the next, and each function remains at a consistent level of abstraction.

Switch Statements

It's hard to make a small `switch` statement.[6] Even a `switch` statement with only two cases is larger than I'd like a single block or function to be. It's also hard to make a `switch` statement that does one thing. By their nature, `switch` statements always do *N* things. Unfortunately we can't always avoid `switch` statements, but we *can* make sure that each `switch` statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism.

---

5. [KP78], p. 37.
6. And, of course, I include if/else chains in this.

Consider Listing 3-4. It shows just one of the operations that might depend on the type of employee.

Listing 3-4
Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

There are several problems with this function. First, it's large, and when new employee types are added, it will grow. Second, it very clearly does more than one thing. Third, it violates the Single Responsibility Principle[7] (SRP) because there is more than one reason for it to change. Fourth, it violates the Open Closed Principle[8] (OCP) because it must change whenever new types are added. But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure. For example we could have

```
isPayday(Employee e, Date date),
```

or

```
deliverPay(Employee e, Money pay),
```

or a host of others. All of which would have the same deleterious structure.

The solution to this problem (see Listing 3-5) is to bury the `switch` statement in the basement of an ABSTRACT FACTORY,[9] and never let anyone see it. The factory will use the `switch` statement to create appropriate instances of the derivatives of `Employee`, and the various functions, such as `calculatePay`, `isPayday`, and `deliverPay`, will be dispatched polymorphically through the `Employee` interface.

My general rule for `switch` statements is that they can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance

7. a. http://en.wikipedia.org/wiki/Single_responsibility_principle
   b. http://www.objectmentor.com/resources/articles/srp.pdf
8. a. http://en.wikipedia.org/wiki/Open/closed_principle
   b. http://www.objectmentor.com/resources/articles/ocp.pdf
9. [GOF].

Listing 3-5
Employee and Factory

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

relationship so that the rest of the system can't see them [G23]. Of course every circumstance is unique, and there are times when I violate one or more parts of that rule.

Use Descriptive Names

In Listing 3-7 I changed the name of our example function from `testableHtml` to `SetupTeardownIncluder.render`. This is a far better name because it better describes what the function does. I also gave each of the private methods an equally descriptive name such as `isTestable` or `includeSetupAndTeardownPages`. It is hard to overestimate the value of good names. Remember Ward's principle: "*You know you are working on clean code when each routine turns out to be pretty much what you expected.*" Half the battle to achieving that principle is choosing good names for small functions that do one thing. The smaller and more focused a function is, the easier it is to choose a descriptive name.

Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment. Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.

Don't be afraid to spend time choosing a name. Indeed, you should try several different names and read the code with each in place. Modern IDEs like Eclipse or IntelliJ make it trivial to change names. Use one of those IDEs and experiment with different names until you find one that is as descriptive as you can make it.

Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.

Be consistent in your names. Use the same phrases, nouns, and verbs in the function names you choose for your modules. Consider, for example, the names `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage`, and `includeSetupPage`. The similar phraseology in those names allows the sequence to tell a story. Indeed, if I showed you just the sequence above, you'd ask yourself: "What happened to `includeTeardownPages`, `includeSuiteTeardownPage`, and `includeTeardownPage`?" How's that for being "*. . . pretty much what you expected.*"

Function Arguments

The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn't be used anyway.

Arguments are hard. They take a lot of conceptual power. That's why I got rid of almost all of them from the example. Consider, for instance, the `StringBuffer` in the example. We could have passed it around as an argument rather than making it an instance variable, but then our readers would have had to interpret it each time they saw it. When you are reading the story told by the module, `includeSetupPage()` is easier to understand than `includeSetupPageInto(newPageContent)`. The argument is at a different level of abstraction than the function name and forces you to know a detail (in other words, `StringBuffer`) that isn't particularly important at that point.

Arguments are even harder from a testing point of view. Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly. If there are no arguments, this is trivial. If there's one argument, it's not too hard. With two arguments the problem gets a bit more challenging. With more than two arguments, testing every combination of appropriate values can be daunting.

Output arguments are harder to understand than input arguments. When we read a function, we are used to the idea of information going *in* to the function through arguments and *out* through the return value. We don't usually expect information to be going out through the arguments. So output arguments often cause us to do a double-take.

One input argument is the next best thing to no arguments. `SetupTeardownIncluder.render(pageData)` is pretty easy to understand. Clearly we are going to *render* the data in the `pageData` object.

Common Monadic Forms

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in `boolean fileExists("MyFile")`. Or you may be operating on that argument, transforming it into something else and *returning it*. For example, `InputStream fileOpen("MyFile")` transforms a file name `String` into an `InputStream` return value. These two uses are what readers expect when they see a function. You should choose names that make the distinction clear, and always use the two forms in a consistent context. (See Command Query Separation below.)

A somewhat less common, but still very useful form for a single argument function, is an *event*. In this form there is an input argument but no output argument. The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system, for example, `void passwordAttemptFailedNtimes(int attempts)`. Use this form with care. It should be very clear to the reader that this is an event. Choose names and contexts carefully.

Try to avoid any monadic functions that don't follow these forms, for example, `void includeSetupPageInto(StringBuffer pageText)`. Using an output argument instead of a return value for a transformation is confusing. If a function is going to transform its input argument, the transformation should appear as the return value. Indeed, `StringBuffer transform(StringBuffer in)` is better than `void transform-(StringBuffer out)`, even if the implementation in the first case simply returns the input argument. At least it still follows the form of a transformation.

Flag Arguments

Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false!

In Listing 3-7 we had no choice because the callers were already passing that flag in, and I wanted to limit the scope of refactoring to the function and below. Still, the method call `render(true)` is just plain confusing to a poor reader. Mousing over the call and seeing `render(boolean isSuite)` helps a little, but not that much. We should have split the function into two: `renderForSuite()` and `renderForSingleTest()`.

Dyadic Functions

A function with two arguments is harder to understand than a monadic function. For example, `writeField(name)` is easier to understand than `writeField(output-Stream, name)`.[10] Though the meaning of both is clear, the first glides past the eye, easily depositing its meaning. The second requires a short pause until we learn to ignore the first parameter. And *that*, of course, eventually results in problems because we should never ignore any part of code. The parts we ignore are where the bugs will hide.

There are times, of course, where two arguments are appropriate. For example, `Point p = new Point(0,0);` is perfectly reasonable. Cartesian points naturally take two arguments. Indeed, we'd be very surprised to see `new Point(0)`. However, the two arguments in this case *are ordered components of a single value!* Whereas `output-Stream` and `name` have neither a natural cohesion, nor a natural ordering.

Even obvious dyadic functions like `assertEquals(expected, actual)` are problematic. How many times have you put the `actual` where the `expected` should be? The two arguments have no natural ordering. The `expected, actual` ordering is a convention that requires practice to learn.

Dyads aren't evil, and you will certainly have to write them. However, you should be aware that they come at a cost and should take advantage of what mechanims may be available to you to convert them into monads. For example, you might make the `writeField` method a member of `outputStream` so that you can say `outputStream.writeField(name)`. Or you might make the `outputStream` a member variable of the current class so that you don't have to pass it. Or you might extract a new class like `FieldWriter` that takes the `outputStream` in its constructor and has a `write` method.

Triads

Functions that take three arguments are significantly harder to understand than dyads. The issues of ordering, pausing, and ignoring are more than doubled. I suggest you think very carefully before creating a triad.

For example, consider the common overload of `assertEquals` that takes three arguments: `assertEquals(message, expected, actual)`. How many times have you read the `message` and thought it was the `expected`? I have stumbled and paused over that particular triad many times. In fact, *every time I see it,* I do a double-take and then learn to ignore the message.

On the other hand, here is a triad that is not quite so insidious: `assertEquals(1.0, amount, .001)`. Although this still requires a double-take, it's one that's worth taking. It's always good to be reminded that equality of floating point values is a relative thing.

---

10. I just finished refactoring a module that used the dyadic form. I was able to make the `outputStream` a field of the class and convert all the `writeField` calls to the monadic form. The result was much cleaner.

Argument Objects

When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own. Consider, for example, the difference between the two following declarations:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Reducing the number of arguments by creating objects out of them may seem like cheating, but it's not. When groups of variables are passed together, the way `x` and `y` are in the example above, they are likely part of a concept that deserves a name of its own.

Argument Lists

Sometimes we want to pass a variable number of arguments into a function. Consider, for example, the `String.format` method:

```
String.format("%s worked %.2f hours.", name, hours);
```

If the variable arguments are all treated identically, as they are in the example above, then they are equivalent to a single argument of type `List`. By that reasoning, `String.format` is actually dyadic. Indeed, the declaration of `String.format` as shown below is clearly dyadic.

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

So all the same rules apply. Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

Verbs and Keywords

Choosing good names for a function can go a long way toward explaining the intent of the function and the order and intent of the arguments. In the case of a monad, the function and argument should form a very nice verb/noun pair. For example, `write(name)` is very evocative. Whatever this "name" thing is, it is being "written." An even better name might be `writeField(name)`, which tells us that the "name" thing is a "field."

This last is an example of the *keyword* form of a function name. Using this form we encode the names of the arguments into the function name. For example, `assertEquals` might be better written as `assertExpectedEqualsActual(expected, actual)`. This strongly mitigates the problem of having to remember the ordering of the arguments.

Have No Side Effects

Side effects are lies. Your function promises to do one thing, but it also does other *hidden* things. Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies.

Consider, for example, the seemingly innocuous function in Listing 3-6. This function uses a standard algorithm to match a `userName` to a `password`. It returns `true` if they match and `false` if anything goes wrong. But it also has a side effect. Can you spot it?

Listing 3-6
`UserValidator.java`

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

The side effect is the call to `Session.initialize()`, of course. The `checkPassword` function, by its name, says that it checks the password. The name does not imply that it initializes the session. So a caller who believes what the name of the function says runs the risk of erasing the existing session data when he or she decides to check the validity of the user.

This side effect creates a temporal coupling. That is, `checkPassword` can only be called at certain times (in other words, when it is safe to initialize the session). If it is called out of order, session data may be inadvertently lost. Temporal couplings are confusing, especially when hidden as a side effect. If you must have a temporal coupling, you should make it clear in the name of the function. In this case we might rename the function `checkPasswordAndInitializeSession`, though that certainly violates "Do one thing."

Output Arguments

Arguments are most naturally interpreted as *inputs* to a function. If you have been programming for more than a few years, I'm sure you've done a double-take on an argument that was actually an *output* rather than an input. For example:

```
appendFooter(s);
```

Does this function append `s` as the footer to something? Or does it append some footer to `s`? Is `s` an input or an output? It doesn't take long to look at the function signature and see:

```
public void appendFooter(StringBuffer report)
```

This clarifies the issue, but only at the expense of checking the declaration of the function. Anything that forces you to check the function signature is equivalent to a double-take. It's a cognitive break and should be avoided.

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 `this` is *intended* to act as an output argument. In other words, it would be better for `appendFooter` to be invoked as

```
report.appendFooter();
```

In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.

Command Query Separation

Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads 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. So the statement reads as "If the `username` attribute was previously set to `unclebob`" and not "set the `username` attribute to `unclebob` and if that worked then. . . ." 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");
  ...
}
```

Prefer Exceptions to Returning Error Codes

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

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) throws Exception {
  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

Functions should do one thing. Error handing 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 and that there should be nothing after the `catch/finally` blocks.

The `Error.java` Dependency Magnet

Returning error codes usually implies that there is some class or enum in which all the error codes are defined.

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

Classes like this are a *dependency magnet;* many other classes must import and use them. Thus, when the `Error enum` changes, all those other classes need to be recompiled and redeployed.[11] This puts a negative pressure on the `Error` class. Programmers don't want

11. Those who felt that they could get away without recompiling and redeploying have been found—and dealt with.

to add new errors because then they have to rebuild and redeploy everything. So they reuse old error codes instead of adding new ones.

When you use exceptions rather than error codes, then new exceptions are *derivatives* of the exception class. They can be added without forcing any recompilation or redeployment.[12]

Don't Repeat Yourself[13]

Look back at Listing 3-1 carefully and you will notice that there is an algorithm that gets repeated four times, once for each of the `SetUp`, `SuiteSetUp`, `TearDown`, and `SuiteTearDown` cases. It's not easy to spot this duplication because the four instances are intermixed with other code and aren't uniformly duplicated. Still, the duplication is a problem because it bloats the code and will require four-fold modification should the algorithm ever have to change. It is also a four-fold opportunity for an error of omission.

This duplication was remedied by the `include` method in Listing 3-7. Read through that code again and notice how the readability of the whole module is enhanced by the reduction of that duplication.

Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it. Consider, for example, that all of Codd's database normal forms serve to eliminate duplication in data. Consider also how object-oriented programming serves to concentrate code into base classes that would otherwise be redundant. Structured programming, Aspect Oriented Programming, Component Oriented Programming, are all, in part, strategies for eliminating duplication. It would appear that since the invention of the subroutine, innovations in software development have been an ongoing attempt to eliminate duplication from our source code.

Structured Programming

Some programmers follow Edsger Dijkstra's rules of structured programming.[14] Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one `return` statement in a function, no `break` or `continue` statements in a loop, and never, *ever,* any `goto` statements.

---

12. This is an example of the Open Closed Principle (OCP) [PPP02].
13. The DRY principle. [PRAG].
14. [SP72].

While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit.

So if you keep your functions small, then the occasional multiple `return`, `break`, or `continue` statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule. On the other hand, `goto` only makes sense in large functions, so it should be avoided.

How Do You Write Functions Like This?

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, 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. 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. 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 rules I've laid down in this chapter. I don't write them that way to start. I don't think anyone could.

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 classes are the nouns. This is not some throwback to the hideous old notion that the nouns and verbs in a requirements document are the first guess of the classes and functions of a system. Rather, this is a much older truth. 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.

This chapter has been about the mechanics of writing functions well. If you follow the rules 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.

SetupTeardownIncluder

Listing 3-7
SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;


  public static String render(PageData pageData) throws Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }
```

Listing 3-7 (continued)
SetupTeardownIncluder.java

```
  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();
  }

  private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
  }

  private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
  }

  private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());
  }

  private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
      includeSuiteTeardownPage();
  }

  private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
  }

  private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
  }

  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
  }

  private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName = getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }

  private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
  }

  private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
      .append("\n!include ")
```

Listing 3-7 (continued)
SetupTeardownIncluder.java

```
        .append(arg)
        .append(" .")
        .append(pagePathName)
        .append("\n");
    }
  }
```

Bibliography

[KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

[PPP02]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[PRAG]: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]: *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.