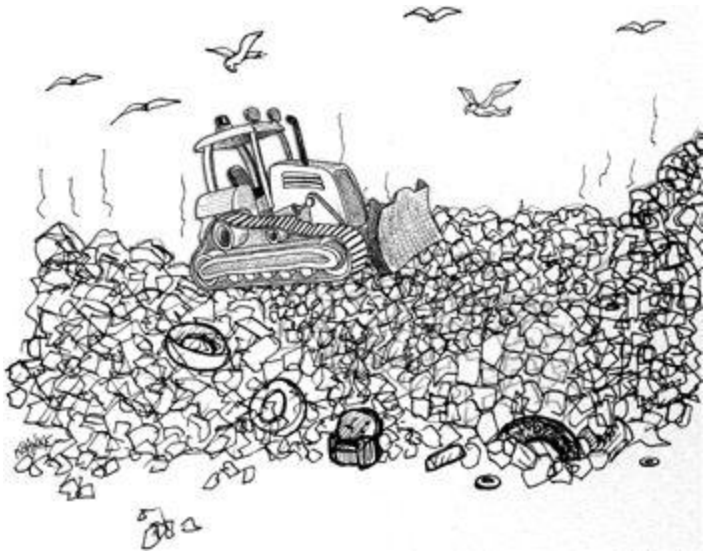
17 Smells and Heuristics



In his wonderful book *Refactoring*,[1] Martin Fowler identified many different "Code Smells." The list that follows includes many of Martin's smells and adds many more of my own. It also includes other pearls and heuristics that I use to practice my trade.

I compiled this list by walking through several different programs and refactoring them. As I made each change, I asked myself *why* I made that change and then wrote the reason down here. The result is a rather long list of things that smell bad to me when I read code.

This list is meant to be read from top to bottom and also to be used as a reference. There is a cross-reference for each heuristic that shows you where it is referenced in the rest of the text in "Appendix C" on page 409.

Comments

C1: *Inappropriate Information*

It is inappropriate for a comment to hold information better held in a different kind of system such as your source code control system, your issue tracking system, or any other record-keeping system. Change

histories, for example, just clutter up source files with volumes of historical and uninteresting text. In general, meta-data such as authors, last-modified-date, SPR number, and so on should not appear in comments. Comments should be reserved for technical notes about the code and design.

C2: *Obsolete Comment*

A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete. If you find an obsolete comment, it is best to update it or get rid of it as quickly as possible. Obsolete comments tend to migrate away from the code they once described. They become floating islands of irrelevance and misdirection in the code.

C3: *Redundant Comment*

A comment is redundant if it describes something that adequately describes itself. For example:

```
i++; // increment i
```

Another example is a Javadoc that says nothing more than (or even less than) the function signature:

```
/**
* @param sellRequest
* @return
* @throws ManagedComponentException
*/
public SellResponse beginSellItem(SellRequest sellRequest)
 throws ManagedComponentException
```

Comments should say things that the code cannot say for itself.

C4: *Poorly Written Comment*

A comment worth writing is worth writing well. If you are going to write a comment, take the time to make sure it is the best comment you can write. Choose your words carefully. Use correct grammar and punctuation. Don’t ramble. Don’t state the obvious. Be brief.

C5: *Commented-Out Code*

It makes me crazy to see stretches of code that are commented out. Who knows how old it is? Who knows whether or not it's meaningful? Yet no one will delete it because everyone assumes someone else needs it or has plans for it.

That code sits there and rots, getting less and less relevant with every passing day. It calls functions that no longer exist. It uses variables whose names have changed. It follows conventions that are long obsolete. It pollutes the modules that contain it and distracts the people who try to read it. Commented-out code is an *abomination*.

When you see commented-out code, *delete it!* Don't worry, the source code control system still remembers it. If anyone really needs it, he or she can go back and check out a previous version. Don't suffer commented-out code to survive.

Environment

E1: *Build Requires More Than One Step*

Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control. You should not need a sequence of arcane commands or context dependent scripts in order to build the individual elements. You should not have to search near and far for all the various little extra JARs, XML files, and other artifacts that the system requires. You *should* be able to check out the system with one simple command and then issue one other simple command to build it.

```
    svn get mySystem
cd mySystem
ant all
```

E2: *Tests Require More Than One Step*

You should be able to run *all* the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE. In the worst case you should be able to issue a single simple command in a shell. Being able to run all the tests is so fundamental and so important that it should be quick, easy, and obvious to do.

Functions

F1: *Too Many Arguments*

Functions should have a small number of arguments. No argument is best, followed by one, two, and three. More than three is very questionable and should be avoided with prejudice. (See "Function Arguments" on page 40.)

F2: *Output Arguments*

Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on. (See "Output Arguments" on page 45.)

F3: *Flag Arguments*

Boolean arguments loudly declare that the function does more than one thing. They are confusing and should be eliminated. (See "Flag Arguments" on page 41.)

F4: *Dead Function*

Methods that are never called should be discarded. Keeping dead code around is wasteful. Don't be afraid to delete the function. Remember, your source code control system still remembers it.

General

G1: *Multiple Languages in One Source File*

Today's modern programming environments make it possible to put many different languages into a single source file. For example, a Java source file might contain snippets of XML, HTML, YAML, JavaDoc, English, JavaScript, and so on. For another example, in addition to HTML a JSP file might contain Java, a tag library syntax, English comments, Javadocs, XML, JavaScript, and so forth. This is confusing at best and carelessly sloppy at worst.

The ideal is for a source file to contain one, and only one, language. Realistically, we will probably have to use more than one. But we should take pains to minimize both the number and extent of extra languages in our source files.

G2: *Obvious Behavior Is Unimplemented*

Following "The Principle of Least Surprise,"[2] any function or class should implement the behaviors that another programmer could reasonably expect. For example, consider a function that translates the name of a day to an `enum` that represents the day.

```
Day day = DayDate.StringToDay(String dayName);
```

We would expect the string "`Monday`" to be translated to `Day.MONDAY`. We would also expect the common abbreviations to be translated, and we would expect the function to ignore case.

When an obvious behavior is not implemented, readers and users of the code can no longer depend on their intuition about function names. They lose their trust in the original author and must fall back on reading the details of the code.

G3: *Incorrect Behavior at the Boundaries*

It seems obvious to say that code should behave correctly. The problem is that we seldom realize just how complicated correct behavior is. Developers often write functions that they think will work, and then trust their intuition rather than going to the effort to prove that their code works in all the corner and boundary cases.

There is no replacement for due diligence. Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm. *Don't rely on your intuition*. Look for every boundary condition and write a test for it.

G4: *Overridden Safeties*

Chernobyl melted down because the plant manager overrode each of the safety mechanisms one by one. The safeties were making it inconvenient

to run an experiment. The result was that the experiment did not get run, and the world saw it's first major civilian nuclear catastrophe.

It is risky to override safeties. Exerting manual control over `serialVersionUID` may be necessary, but it is always risky. Turning off certain compiler warnings (or all warnings!) may help you get the build to succeed, but at the risk of endless debugging sessions. Turning off failing tests and telling yourself you'll get them to pass later is as bad as pretending your credit cards are free money.

G5: *Duplication*

This is one of the most important rules in this book, and you should take it very seriously. Virtually every author who writes about software design mentions this rule. Dave Thomas and Andy Hunt called it the DRY[3] principle (Don't Repeat Yourself). Kent Beck made it one of the core principles of Extreme Programming and called it: "Once, and only once." Ron Jeffries ranks this rule second, just below getting all the tests to pass.

Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright. By folding the duplication into such an abstraction, you increase the vocabulary of the language of your design. Other programmers can use the abstract facilities you create. Coding becomes faster and less error prone because you have raised the abstraction level.

The most obvious form of duplication is when you have clumps of identical code that look like some programmers went wild with the mouse, pasting the same code over and over again. These should be replaced with simple methods.

A more subtle form is the `switch/case` or `if/else` chain that appears again and again in various modules, always testing for the same set of conditions. These should be replaced with polymorphism.

Still more subtle are the modules that have similar algorithms, but that don't share similar lines of code. This is still duplication and should be addressed by using the TEMPLATE METHOD,[4] or STRATEGY[5] pattern.

Indeed, most of the design patterns that have appeared in the last fifteen years are simply well-known ways to eliminate duplication. So too the Codd Normal Forms are a strategy for eliminating duplication in database schemae. OO itself is a strategy for organizing modules and eliminating duplication. Not surprisingly, so is structured programming.

I think the point has been made. Find and eliminate duplication wherever you can.

G6: *Code at Wrong Level of Abstraction*

It is important to create abstractions that separate higher level general concepts from lower level detailed concepts. Sometimes we do this by creating abstract classes to hold the higher level concepts and derivatives to hold the lower level concepts. When we do this, we need to make sure that the separation is complete. We want *all* the lower level concepts to be in the derivatives and *all* the higher level concepts to be in the base class.

For example, constants, variables, or utility functions that pertain only to the detailed implementation should not be present in the base class. The base class should know nothing about them.

This rule also pertains to source files, components, and modules. Good software design requires that we separate concepts at different levels and place them in different containers. Sometimes these containers are base classes or derivatives and sometimes they are source files, modules, or components. Whatever the case may be, the separation needs to be complete. We don't want lower and higher level concepts mixed together.

Consider the following code:

```
  public interface Stack {
Object pop() throws EmptyException;
void push(Object o) throws FullException;
double percentFull();

class EmptyException extends Exception {}
class FullException extends Exception {}
}
```

The `percentFull` function is at the wrong level of abstraction. Although there are many implementations of `Stack` where the concept of

fullness is reasonable, there are other implementations that simply *could not know* how full they are. So the function would be better placed in a derivative interface such as `BoundedStack`.

Perhaps you are thinking that the implementation could just return zero if the stack were boundless. The problem with that is that no stack is truly boundless. You cannot really prevent an `OutOfMemoryException` by checking for

stack.percentFull() < 50.0.

Implementing the function to return 0 would be telling a lie.

The point is that you cannot lie or fake your way out of a misplaced abstraction. Isolating abstractions is one of the hardest things that software developers do, and there is no quick fix when you get it wrong.

G7: *Base Classes Depending on Their Derivatives*

The most common reason for partitioning concepts into base and derivative classes is so that the higher level base class concepts can be independent of the lower level derivative class concepts. Therefore, when we see base classes mentioning the names of their derivatives, we suspect a problem. In general, base classes should know nothing about their derivatives.

There are exceptions to this rule, of course. Sometimes the number of derivatives is strictly fixed, and the base class has code that selects between the derivatives. We see this a lot in finite state machine implementations. However, in that case the derivatives and base class are strongly coupled and always deploy together in the same jar file. In the general case we want to be able to deploy derivatives and bases in different jar files.

Deploying derivatives and bases in different jar files and making sure the base jar files know nothing about the contents of the derivative jar files allow us to deploy our systems in discrete and independent components. When such components are modified, they can be redeployed without having to redeploy the base components. This means that the impact of a change is greatly lessened, and maintaining systems in the field is made much simpler.

G8: *Too Much Information*

Well-defined modules have very small interfaces that allow you to do a lot with a little. Poorly defined modules have wide and deep interfaces that force you to use many different gestures to get simple things done. A well-defined interface does not offer very many functions to depend upon, so coupling is low. A poorly defined interface provides lots of functions that you must call, so coupling is high.

Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better.

Hide your data. Hide your utility functions. Hide your constants and your temporaries. Don't create classes with lots of methods or lots of instance variables. Don't create lots of protected variables and functions for your subclasses. Concentrate on keeping interfaces very tight and very small. Help keep coupling low by limiting information.

G9: *Dead Code*

Dead code is code that isn't executed. You find it in the body of an `if` statement that checks for a condition that can't happen. You find it in the `catch` block of a `try` that never `throws`. You find it in little utility methods that are never called or `switch/case` conditions that never occur.

The problem with dead code is that after awhile it starts to smell. The older it is, the stronger and sourer the odor becomes. This is because dead code is not completely updated when designs change. It still *compiles*, but it does not follow newer conventions or rules. It was written at a time when the system was *different*. When you find dead code, do the right thing. Give it a decent burial. Delete it from the system.

G10: *Vertical Separation*

Variables and function should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope. We don't want local variables declared hundreds of lines distant from their usages.

Private functions should be defined just below their first usage. Private functions belong to the scope of the whole class, but we'd still like to limit the vertical distance between the invocations and definitions. Finding a private function should just be a matter of scanning downward from the first usage.

G11: *Inconsistency*

If you do something a certain way, do all similar things in the same way. This goes back to the principle of least surprise. Be careful with the conventions you choose, and once chosen, be careful to continue to follow them.

If within a particular function you use a variable named `response` to hold an `HttpServletResponse`, then use the same variable name consistently in the other functions that use `HttpServletResponse` objects. If you name a method `processVerificationRequest`, then use a similar name, such as `processDeletionRequest`, for the methods that process other kinds of requests.

Simple consistency like this, when reliably applied, can make code much easier to read and modify.

G12: *Clutter*

Of what use is a default constructor with no implementation? All it serves to do is clutter up the code with meaningless artifacts. Variables that aren't used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source files clean, well organized, and free of clutter.

G13: *Artificial Coupling*

Things that don't depend upon each other should not be artificially coupled. For example, general `enums` should not be contained within more specific classes because this forces the whole application to know about these more specific classes. The same goes for general purpose `static` functions being declared in specific classes.

In general an artificial coupling is a coupling between two modules that serves no direct purpose. It is a result of putting a variable, constant, or

function in a temporarily convenient, though inappropriate, location. This is lazy and careless.

Take the time to figure out where functions, constants, and variables ought to be declared. Don't just toss them in the most convenient place at hand and then leave them there.

G14: *Feature Envy*

This is one of Martin Fowler's code smells.[6] The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it *envies* the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating. For example:

```
  public class HourlyPayCalculator {
 public Money calculateWeeklyPay(HourlyEmployee e) {
  int tenthRate = e.getTenthRate().getPennies();
  int tenthsWorked = e.getTenthsWorked();
  int straightTime = Math.min(400, tenthsWorked);
  int overTime = Math.max(0, tenthsWorked - straightTime);
  int straightPay = straightTime * tenthRate;
  int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
  return new Money(straightPay + overtimePay);
 }
}
```

The `calculateWeeklyPay` method reaches into the `HourlyEmployee` object to get the data on which it operates. The `calculateWeeklyPay` method *envies* the scope of `HourlyEmployee`. It "wishes" that it could be inside `HourlyEmployee`.

All else being equal, we want to eliminate Feature Envy because it exposes the internals of one class to another. Sometimes, however, Feature Envy is a necessary evil. Consider the following:

```
  public class HourlyEmployeeReport {
 private HourlyEmployee employee ;
```

```
  public HourlyEmployeeReport(HourlyEmployee e) {
    this.employee = e;
  }

  String reportHours() {
    return String.format(
      "Name: %s\tHours:%d.%1d\n",
      employee.getName(),
      employee.getTenthsWorked()/10,
      employee.getTenthsWorked()%10);
  }
}
```

Clearly, the `reportHours` method envies the `HourlyEmployee` class. On the other hand, we don't want `HourlyEmployee` to have to know about the format of the report. Moving that format string into the `HourlyEmployee` class would violate several principles of object oriented design.[7] It would couple `HourlyEmployee` to the format of the report, exposing it to changes in that format.

G15: *Selector Arguments*

There is hardly anything more abominable than a dangling `false` argument at the end of a function call. What does it mean? What would it change if it were `true`? Not only is the purpose of a selector argument difficult to remember, each selector argument combines many functions into one. Selector arguments are just a lazy way to avoid splitting a large function into several smaller functions. Consider:

```
  public int calculateWeeklyPay(boolean overtime) {
  int tenthRate = getTenthRate();
  int tenthsWorked = getTenthsWorked();
  int straightTime = Math.min(400, tenthsWorked);
  int overTime = Math.max(0, tenthsWorked - straightTime);
  int straightPay = straightTime * tenthRate;
  double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
  int overtimePay = (int)Math.round(overTime*overtimeRate);
  return straightPay + overtimePay;
}
```

You call this function with a `true` if overtime is paid as time and a half, and with a `false` if overtime is paid as straight time. It's bad enough that you must remember what `calculateWeeklyPay(false)` means whenever you happen to stumble across it. But the real shame of a function like this is that the author missed the opportunity to write the following:

```
    public int straightPay() {
  return getTenthsWorked() * getTenthRate();
}

public int overTimePay() {
  int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
  int overTimePay = overTimeBonus(overTimeTenths);
  return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
  double bonus = 0.5 * getTenthRate() * overTimeTenths;
  return (int) Math.round(bonus);
}
```

Of course, selectors need not be `boolean`. They can be enums, integers, or any other type of argument that is used to select the behavior of the function. In general it is better to have many functions than to pass some code into a function to select the behavior.

G16: *Obscured Intent*

We want code to be as expressive as possible. Run-on expressions, Hungarian notation, and magic numbers all obscure the author's intent. For example, here is the `overTimePay` function as it might have appeared:

```
    public int m_otCalc() {
  return iThsWkd * iThsRte +
   (int) Math.round(0.5 * iThsRte *
    Math.max(0, iThsWkd - 400)
   );
}
```

Small and dense as this might appear, it's also virtually impenetrable. It is worth taking the time to make the intent of our code visible to our readers.

G17: *Misplaced Responsibility*

One of the most important decisions a software developer can make is where to put code. For example, where should the `PI` constant go? Should it be in the `Math` class? Perhaps it belongs in the `Trigonometry` class? Or maybe in the `Circle` class?

The principle of least surprise comes into play here. Code should be placed where a reader would naturally expect it to be. The `PI` constant should go where the trig functions are declared. The OVERTIME_RATE constant should be declared in the `HourlyPay-Calculator` class.

Sometimes we get "clever" about where to put certain functionality. We'll put it in a function that's convenient for us, but not necessarily intuitive to the reader. For example, perhaps we need to print a report with the total of hours that an employee worked. We could sum up those hours in the code that prints the report, or we could try to keep a running total in the code that accepts time cards.

One way to make this decision is to look at the names of the functions. Let's say that our report module has a function named `getTotalHours`. Let's also say that the module that accepts time cards has a `saveTimeCard` function. Which of these two functions, by it's name, implies that it calculates the total? The answer should be obvious.

Clearly, there are sometimes performance reasons why the total should be calculated as time cards are accepted rather than when the report is printed. That's fine, but the names of the functions ought to reflect this. For example, there should be a `computeRunning-TotalOfHours` function in the timecard module.

G18: *Inappropriate Static*

`Math.max(double a, double b)` is a good static method. It does not operate on a single instance; indeed, it would be silly to have to say `new Math().max(a,b)` or even `a.max(b)`. All the data that `max` uses comes from its two arguments, and not from any "owning" object. More to the

point, there is almost *no chance* that we'd want `Math.max` to be polymorphic.

Sometimes, however, we write static functions that should not be static. For example, consider:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

Again, this seems like a reasonable `static` function. It doesn't operate on any particular object and gets all it's data from it's arguments. However, there is a reasonable chance that we'll want this function to be polymorphic. We may wish to implement several different algorithms for calculating hourly pay, for example, `OvertimeHourlyPayCalculator` and `StraightTimeHourlyPayCalculator`. So in this case the function should not be static. It should be a nonstatic member function of `Employee`.

In general you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you'll want it to behave polymorphically.

G19: *Use Explanatory Variables*

Kent Beck wrote about this in his great book *Smalltalk Best Practice Patterns*[8] and again more recently in his equally great book *Implementation Patterns*.[9] One of the more powerful ways to make a program readable is to break the calculations up into intermediate values that are held in variables with meaningful names.

Consider this example from FitNesse:

```
  Matcher match = headerPattern.matcher(line);
if(match.find())
{
 String key = match.group(1);
 String value = match.group(2);
 headers.put(key.toLowerCase(), value);
}
```

The simple use of explanatory variables makes it clear that the first matched group is the *key*, and the second matched group is the *value*.

It is hard to overdo this. More explanatory variables are generally better than fewer. It is remarkable how an opaque module can suddenly become transparent simply by breaking the calculations up into well-named intermediate values.

G20: *Function Names Should Say What They Do*

Look at this code:

```
Date newDate = date.add(5);
```

Would you expect this to add five days to the date? Or is it weeks, or hours? Is the `date` instance changed or does the function just return a new `Date` without changing the old one? *You can't tell from the call what the function does*.

If the function adds five days to the date and changes the date, then it should be called `addDaysTo` or `increaseByDays`. If, on the other hand, the function returns a new date that is five days later but does not change the date instance, it should be called `daysLater` or `daysSince`.

If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

G21: *Understand the Algorithm*

Lots of very funny code is written because people don't take the time to understand the algorithm. They get something to work by plugging in enough `if` statements and flags, without really stopping to consider what is really going on.

Programming is often an exploration. You *think* you know the right algorithm for something, but then you wind up fiddling with it, prodding and poking at it, until you get it to "work." How do you know it "works"? Because it passes the test cases you can think of.

There is nothing wrong with this approach. Indeed, often it is the only way to get a function to do what you think it should. However, it is not sufficient to leave the quotation marks around the word "work."

Before you consider yourself to be done with a function, make sure you *understand* how it works. It is not good enough that it passes all the tests. You must *know*[10] that the solution is correct.

Often the best way to gain this knowledge and understanding is to refactor the function into something that is so clean and expressive that it is *obvious* how it works.

G22: *Make Logical Dependencies Physical*

If one module depends upon another, that dependency should be physical, not just logical. The dependent module should not make assumptions (in other words, logical dependencies) about the module it depends upon. Rather it should explicitly ask that module for all the information it depends upon.

For example, imagine that you are writing a function that prints a plain text report of hours worked by employees. One class named `HourlyReporter` gathers all the data into a convenient form and then passes it to `HourlyReportFormatter` to print it. (See Listing 17-1.)

Listing 17-1 `HourlyReporter.java`

```
 public class HourlyReporter {
private HourlyReportFormatter formatter;
private List<LineItem> page;
private final int PAGE_SIZE = 55;

public HourlyReporter(HourlyReportFormatter formatter) {
  this.formatter = formatter;
  page = new ArrayList<LineItem>();
}

public void generateReport(List<HourlyEmployee> employees) {
  for (HourlyEmployee e : employees) {
    addLineItemToPage(e);
    if (page.size() == PAGE_SIZE)
      printAndClearItemList();
  }
  if (page.size() > 0)
```

```
      printAndClearItemList();
    }

    private void printAndClearItemList() {
      formatter.format(page);
      page.clear();
    }

    private void addLineItemToPage(HourlyEmployee e) {
      LineItem item = new LineItem();
      item.name = e.getName();
      item.hours = e.getTenthsWorked() / 10;

      item.tenths = e.getTenthsWorked() % 10;
      page.add(item);
    }

    public class LineItem {
      public String name;
      public int hours;
      public int tenths;
    }
  }
```

This code has a logical dependency that has not been physicalized. Can you spot it? It is the constant `PAGE_SIZE`. Why should the `HourlyReporter` know the size of the page? Page size should be the responsibility of the `HourlyReportFormatter`.

The fact that PAGE_SIZE is declared in `HourlyReporter` represents a misplaced responsibility [G17] that causes `HourlyReporter` to assume that it knows what the page size ought to be. Such an assumption is a logical dependency. `HourlyReporter` depends on the fact that `HourlyReportFormatter` can deal with page sizes of 55. If some implementation of `HourlyReportFormatter` could not deal with such sizes, then there would be an error.

We can physicalize this dependency by creating a new method in `HourlyReport-Formatter` named `getMaxPageSize(). HourlyReporter`

will then call that function rather than using the `PAGE_SIZE` constant.

G23: *Prefer Polymorphism to If/Else or Switch/Case*

This might seem a strange suggestion given the topic of Chapter 6. After all, in that chapter I make the point that switch statements are probably appropriate in the parts of the system where adding new functions is more likely than adding new types.

First, most people use switch statements because it's the obvious brute force solution, not because it's the right solution for the situation. So this heuristic is here to remind us to consider polymorphism before using a switch.

Second, the cases where functions are more volatile than types are relatively rare. So *every* switch statement should be suspect.

I use the following "ONE SWITCH" rule: *There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.*

G24: *Follow Standard Conventions*

Every team should follow a coding standard based on common industry norms. This coding standard should specify things like where to declare instance variables; how to name classes, methods, and variables; where to put braces; and so on. The team should not need a document to describe these conventions because their code provides the examples.

Everyone on the team should follow these conventions. This means that each team member must be mature enough to realize that it doesn't matter a whit where you put your braces so long as you all agree on where to put them.

If you would like to know what conventions I follow, you'll see them in the refactored code in Listing B-7 on page 394, through Listing B-14.

G25: *Replace Magic Numbers with Named Constants*

This is probably one of the oldest rules in software development. I remember reading it in the late sixties in introductory COBOL,

FORTRAN, and PL/1 manuals. In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.

For example, the number 86,400 should be hidden behind the constant `SECONDS_PER_DAY`. If you are printing 55 lines per page, then the constant 55 should be hidden behind the constant `LINES_PER_PAGE`.

Some constants are so easy to recognize that they don't always need a named constant to hide behind so long as they are used in conjunction with very self-explanatory code. For example:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Do we really need the constants `FEET_PER_MILE`, `WORK_HOURS_PER_DAY`, and `TWO` in the above examples? Clearly, the last case is absurd. There are some formulae in which constants are simply better written as raw numbers. You might quibble about the `WORK_HOURS_PER_DAY` case because the laws or conventions might change. On the other hand, that formula reads so nicely with the 8 in it that I would be reluctant to add 17 extra characters to the readers' burden. And in the `FEET_PER_MILE` case, the number 5280 is so very well known and so unique a constant that readers would recognize it even if it stood alone on a page with no context surrounding it.

Constants like 3.141592653589793 are also very well known and easily recognizable. However, the chance for error is too great to leave them raw. Every time someone sees 3.1415927535890793, they know that it is π, and so they fail to scrutinize it. (Did you catch the single-digit error?) We also don't want people using 3.14, 3.14159, 3.142, and so forth. Therefore, it is a good thing that `Math.PI` has already been defined for us.

The term "Magic Number" does not apply only to numbers. It applies to any token that has a value that is not self-describing. For example:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

There are two magic numbers in this assertion. The first is obviously 7777, though what it might mean is not obvious. The second magic number is "`John Doe`," and again the intent is not clear.

It turns out that "`John Doe`" is the name of employee #7777 in a well-known test database created by our team. Everyone in the team knows that when you connect to this database, it will have several employees already cooked into it with well-known values and attributes. It also turns out that "`John Doe`" represents the sole hourly employee in that test database. So this test should really read:

```
  assertEquals(
HOURLY_EMPLOYEE_ID,
Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26: *Be Precise*

Expecting the first match to be the *only* match to a query is probably naive. Using floating point numbers to represent currency is almost criminal. Avoiding locks and/or transaction management because you don't think concurrent update is likely is lazy at best. Declaring a variable to be an `ArrayList` when a `List` will due is overly constraining. Making all variables `protected` by default is not constraining enough.

When you make a decision in your code, make sure you make it *precisely*. Know why you have made it and how you will deal with any exceptions. Don't be lazy about the precision of your decisions. If you decide to call a function that might return `null`, make sure you check for `null`. If you query for what you think is the only record in the database, make sure your code checks to be sure there aren't others. If you need to deal with currency, use integers[11] and deal with rounding appropriately. If there is the possibility of concurrent update, make sure you implement some kind of locking mechanism.

Ambiguities and imprecision in code are either a result of disagreements or laziness. In either case they should be eliminated.

G27: *Structure over Convention*

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance. For example, switch/cases with nicely named enumerations are inferior to base classes with abstract methods. No one is forced to implement the `switch/case` statement the same way each time; but the

base classes do enforce that concrete classes have all abstract methods implemented.

G28: *Encapsulate Conditionals*

Boolean logic is hard enough to understand without having to see it in the context of an `if` or `while` statement. Extract functions that explain the intent of the conditional.

For example:

```
if (shouldBeDeleted(timer))
```

is preferable to

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29: *Avoid Negative Conditionals*

Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives. For example:

```
if (buffer.shouldCompact())
```

is preferable to

```
if (!buffer.shouldNotCompact())
```

G30: *Functions Should Do One Thing*

It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than *one thing*, and should be converted into many smaller functions, each of which does *one thing*.

For example:

```
  public void pay() {
 for (Employee e : employees) {
  if (e.isPayday()) {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
  }
 }
}
```

This bit of code does three things. It loops over all the employees, checks to see whether each employee ought to be paid, and then pays the employee. This code would be better written as:

```
  public void pay() {
 for (Employee e : employees)
  payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
 if (e.isPayday())
  calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
 Money pay = e.calculatePay();
 e.deliverPay(pay);
}
```

Each of these functions does one thing. (See “Do One Thing” on page 35.)

G31: *Hidden Temporal Couplings*

Temporal couplings are often necessary, but you should not hide the coupling. Structure the arguments of your functions such that the order in which they should be called is obvious. Consider the following:

```
  public class MoogDiver {
 Gradient gradient;
 List<Spline> splines;

 public void dive(String reason) {
  saturateGradient();
  reticulateSplines();
  diveForMoog(reason);
 }
 …
}
```

The order of the three functions is important. You must saturate the gradient before you can reticulate the splines, and only then can you dive for the moog. Unfortunately, the code does not enforce this temporal coupling. Another programmer could call `reticulate-Splines` before `saturateGradient` was called, leading to an `UnsaturatedGradientException`. A better solution is:

```
  public class MoogDiver {
Gradient gradient;
List<Spline> splines;

public void dive(String reason) {
  Gradient gradient = saturateGradient();
  List<Spline> splines = reticulateSplines(gradient);
  diveForMoog(splines, reason);
}
…
}
```

This exposes the temporal coupling by creating a bucket brigade. Each function produces a result that the next function needs, so there is no reasonable way to call them out of order.

You might complain that this increases the complexity of the functions, and you'd be right. But that extra syntactic complexity exposes the true temporal complexity of the situation.

Note that I left the instance variables in place. I presume that they are needed by private methods in the class. Even so, I want the arguments in place to make the temporal coupling explicit.

G32: *Don't Be Arbitrary*

Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it. If a structure appears consistently throughout the system, others will use it and preserve the convention. For example, I was recently merging changes to FitNesse and discovered that one of our committers had done this:

```
public class AliasLinkWidget extends ParentWidget
{
 public static class VariableExpandingWidgetRoot {
  …

 …
}
```

The problem with this was that `VariableExpandingWidgetRoot` had no need to be inside the scope of `AliasLinkWidget`. Moreover, other unrelated classes made use of `AliasLinkWidget.VariableExpandingWidgetRoot`. These classes had no need to know about `AliasLinkWidget`.

Perhaps the programmer had plopped the `VariableExpandingWidgetRoot` into `AliasWidget` as a matter of convenience, or perhaps he thought it really needed to be scoped inside `AliasWidget`. Whatever the reason, the result wound up being arbitrary. Public classes that are not utilities of some other class should not be scoped inside another class. The convention is to make them public at the top level of their package.

G33: *Encapsulate Boundary Conditions*

Boundary conditions are hard to keep track of. Put the processing for them in one place. Don't let them leak all over the code. We don't want swarms of `+1`s and `-1`s scattered hither and yon. Consider this simple example from FIT:

```
if(level + 1 < tags.length)
{
 parts = new Parse(body, tags, level + 1, offset + endTag);
 body = null;
}
```

Notice that `level+1` appears twice. This is a boundary condition that should be encapsulated within a variable named something like `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
```

```
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

G34: *Functions Should Descend Only One Level of Abstraction*

The statements within a function should all be written at the same level of abstraction, which should be one level below the operation described by the name of the function. This may be the hardest of these heuristics to interpret and follow. Though the idea is plain enough, humans are just far too good at seamlessly mixing levels of abstraction. Consider, for example, the following code taken from FitNesse:

```
   public String render() throws Exception
{
  StringBuffer html = new StringBuffer("<hr");
  if(size > 0)
    html.append(" size=\"").append(size + 1).append("\"");
  html.append(">");

  return html.toString();
}
```

A moment's study and you can see what's going on. This function constructs the HTML tag that draws a horizontal rule across the page. The height of that rule is specified in the `size` variable.

Now look again. This method is mixing at least two levels of abstraction. The first is the notion that a horizontal rule has a size. The second is the syntax of the `HR` tag itself. This code comes from the `HruleWidget` module in FitNesse. This module detects a row of four or more dashes and converts it into the appropriate HR tag. The more dashes, the larger the size.

I refactored this bit of code as follows. Note that I changed the name of the `size` field to reflect its true purpose. It held the number of extra dashes.

```
   public String render() throws Exception
{
```

```
  HtmlTag hr = new HtmlTag("hr");
  if (extraDashes > 0)
    hr.addAttribute("size", hrSize(extraDashes));
  return hr.html();
}

private String hrSize(int height)
{
  int hrSize = height + 1;
  return String.format("%d", hrSize);
}
```

This change separates the two levels of abstraction nicely. The `render` function simply constructs an HR tag, without having to know anything about the HTML syntax of that tag. The `HtmlTag` module takes care of all the nasty syntax issues.

Indeed, by making this change I caught a subtle error. The original code did not put the closing slash on the HR tag, as the XHTML standard would have it. (In other words, it emitted `<hr>` instead of `<hr/>`.) The `HtmlTag` module had been changed to conform to XHTML long ago.

Separating levels of abstraction is one of the most important functions of refactoring, and it's one of the hardest to do well. As an example, look at the code below. This was my first attempt at separating the abstraction levels in the `HruleWidget.render method`.

```
  public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (size > 0) {
    hr.addAttribute("size", ""+(size+1));
  }
  return hr.html();
}
```

My goal, at this point, was to create the necessary separation and get the tests to pass. I accomplished that goal easily, but the result was a function that *still* had mixed levels of abstraction. In this case the mixed levels were the construction of the HR tag and the interpretation and formatting

of the `size` variable. This points out that when you break a function along lines of abstraction, you often uncover new lines of abstraction that were obscured by the previous structure.

G35: *Keep Configurable Data at High Levels*

If you have a constant such as a default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to that low-level function called from the high-level function. Consider the following code from FitNesse:

```
    public static void main(String[] args) throws Exception
{
 Arguments arguments = parseCommandLine(args);
 …
}

public class Arguments
{
 public static final String DEFAULT_PATH = ".";
 public static final String DEFAULT_ROOT = "FitNesseRoot";
 public static final int DEFAULT_PORT = 80;
 public static final int DEFAULT_VERSION_DAYS = 14;
 …
}
```

The command-line arguments are parsed in the very first executable line of FitNesse. The default values of those arguments are specified at the top of the `Argument` class. You don't have to go looking in low levels of the system for statements like this one:

```
if (arguments.port == 0) // use 80 by default
```

The configuration constants reside at a very high level and are easy to change. They get passed down to the rest of the application. The lower levels of the application do not own the values of these constants.

G36: *Avoid Transitive Navigation*

In general we don't want a single module to know much about its collaborators. More specifically, if `A` collaborates with `B`, and `B`

collaborates with `C`, we don't want modules that use `A` to know about `C`. (For example, we don't want `a.getB().getC().doSomething();`.)

This is sometimes called the Law of Demeter. The Pragmatic Programmers call it "Writing Shy Code."[12] In either case it comes down to making sure that modules know only about their immediate collaborators and do not know the navigation map of the whole system.

If many modules used some form of the statement `a.getB().getC()`, then it would be difficult to change the design and architecture to interpose a `Q` between `B` and `C`. You'd have to find every instance of `a.getB().getC()` and convert it to `a.getB().getQ().getC()`. This is how architectures become rigid. Too many modules know too much about the architecture.

Rather we want our immediate collaborators to offer all the services we need. We should not have to roam through the object graph of the system, hunting for the method we want to call. Rather we should simply be able to say:

```
myCollaborator.doSomething().
```

Java

J1: *Avoid Long Import Lists by Using Wildcards*

If you use two or more classes from a package, then import the whole package with

```
import package.*;
```

Long lists of imports are daunting to the reader. We don't want to clutter up the tops of our modules with 80 lines of imports. Rather we want the imports to be a concise statement about which packages we collaborate with.

Specific imports are hard dependencies, whereas wildcard imports are not. If you specifically import a class, then that class *must* exist. But if you import a package with a wildcard, no particular classes need to exist. The import statement simply adds the package to the search path when hunting for names. So no true dependency is created by such imports, and they therefore serve to keep our modules less coupled.

There are times when the long list of specific imports can be useful. For example, if you are dealing with legacy code and you want to find out what classes you need to build mocks and stubs for, you can walk down the list of specific imports to find out the true qualified names of all those classes and then put the appropriate stubs in place. However, this use for specific imports is very rare. Furthermore, most modern IDEs will allow you to convert the wildcarded imports to a list of specific imports with a single command. So even in the legacy case it's better to import wildcards.

Wildcard imports can sometimes cause name conflicts and ambiguities. Two classes with the same name, but in different packages, will need to be specifically imported, or at least specifically qualified when used. This can be a nuisance but is rare enough that using wildcard imports is still generally better than specific imports.

J2: ***Don't Inherit Constants***

I have seen this several times and it always makes me grimace. A programmer puts some constants in an interface and then gains access to those constants by inheriting that interface. Take a look at the following code:

```
  public class HourlyEmployee extends Employee {
 private int tenthsWorked;
 private double hourlyRate;

 public Money calculatePay() {
   int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
   int overTime = tenthsWorked - straightTime;
   return new Money(
     hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
   );
 }
 …
}
```

Where did the constants `TENTHS_PER_WEEK` and `OVERTIME_RATE` come from? They might have come from class `Employee`; so let's take a look at that:

```
public abstract class Employee implements PayrollConstants {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
```

Nope, not there. But then where? Look closely at class `Employee`. It implements `PayrollConstants`.

```
public interface PayrollConstants {
  public static final int TENTHS_PER_WEEK = 400;
  public static final double OVERTIME_RATE = 1.5;
}
```

This is a hideous practice! The constants are hidden at the top of the inheritance hierarchy. Ick! Don't use inheritance as a way to cheat the scoping rules of the language. Use a static import instead.

```
import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  …
}
```

J3: *Constants versus Enums*

Now that `enum`s have been added to the language (Java 5), use them! Don't keep using the old trick of `public static final int`s. The meaning of `int`s can get lost. The meaning of `enum`s cannot, because they belong to an enumeration that is named.

What's more, study the syntax for `enum`s carefully. They can have methods and fields. This makes them very powerful tools that allow much more expression and flexibility than `int`s. Consider this variation on the payroll code:

```
  public class HourlyEmployee extends Employee {
 private int tenthsWorked;
 HourlyPayGrade grade;

 public Money calculatePay() {
  int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
  int overTime = tenthsWorked - straightTime;
  return new Money(
   grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
  );
 }
 …
}
public enum HourlyPayGrade {
 APPRENTICE {
  public double rate() {
   return 1.0;
  }
 },
 LEUTENANT_JOURNEYMAN {
  public double rate() {
   return 1.2;
  }
 },
 JOURNEYMAN {
  public double rate() {
   return 1.5;
  }
 },
 MASTER {
  public double rate() {
   return 2.0;
  }
```

```
    };

    public abstract double rate();
}
```

Names

N1: *Choose Descriptive Names*

Don't be too quick to choose a name. Make sure the name is descriptive. Remember that meanings tend to drift as software evolves, so frequently reevaluate the appropriateness of the names you choose.

This is not just a "feel-good" recommendation. Names in software are 90 percent of what make software readable. You need to take the time to choose them wisely and keep them relevant. Names are too important to treat carelessly.

Consider the code below. What does it do? If I show you the code with well-chosen names, it will make perfect sense to you, but like this it's just a hodge-podge of symbols and magic numbers.

```
public int x() {
int q = 0;
int z = 0;
for (int kk = 0; kk < 10; kk++) {
  if (l[z] == 10)
  {
    q += 10 + (l[z + 1] + l[z + 2]);
    z += 1;
  }
  else if (l[z] + l[z + 1] == 10)
  {
    q += 10 + l[z + 2];
    z += 2;
  } else {
    q += l[z] + l[z + 1];
    z += 2;
  }
```

```
    }
    return q;
  }
```

Here is the code the way it should be written. This snippet is actually less complete than the one above. Yet you can infer immediately what it is trying to do, and you could very likely write the missing functions based on that inferred meaning. The magic numbers are no longer magic, and the structure of the algorithm is compellingly descriptive.

```
  public int score() {
  int score = 0;
  int frame = 0;
  for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
    if (isStrike(frame)) {
      score += 10 + nextTwoBallsForStrike(frame);
      frame += 1;
    } else if (isSpare(frame)) {
      score += 10 + nextBallForSpare(frame);
      frame += 2;
    } else {
      score += twoBallsInFrame(frame);
      frame += 2;
    }
  }
  return score;
}
```

The power of carefully chosen names is that they overload the structure of the code with description. That overloading sets the readers' expectations about what the other functions in the module do. You can infer the implementation of `isStrike()` by looking at the code above. When you read the `isStrike` method, it will be "pretty much what you expected."[13]

```
  private boolean isStrike(int frame) {
  return rolls[frame] == 10;
}
```

N2: ***Choose Names at the Appropriate Level of Abstraction***

Don't pick names that communicate implementation; choose names the reflect the level of abstraction of the class or function you are working in. This is hard to do. Again, people are just too good at mixing levels of abstractions. Each time you make a pass over your code, you will likely find some variable that is named at too low a level. You should take the opportunity to change those names when you find them. Making code readable requires a dedication to continuous improvement. Consider the `Modem` interface below:

```
  public interface Modem {
  boolean dial(String phoneNumber);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedPhoneNumber();
}
```

At first this looks fine. The functions all seem appropriate. Indeed, for many applications they are. But now consider an application in which some modems aren't connected by dialling. Rather they are connected permanently by hard wiring them together (think of the cable modems that provide Internet access to most homes nowadays). Perhaps some are connected by sending a port number to a switch over a USB connection. Clearly the notion of phone numbers is at the wrong level of abstraction. A better naming strategy for this scenario might be:

```
  public interface Modem {
  boolean connect(String connectionLocator);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedLocator();
}
```

Now the names don't make any commitments about phone numbers. They can still be used for phone numbers, or they could be used for any other kind of connection strategy.

N3: ***Use Standard Nomenclature Where Possible***

Names are easier to understand if they are based on existing convention or usage. For example, if you are using the DECORATOR pattern, you should use the word `Decorator` in the names of the decorating classes. For example, `AutoHangupModemDecorator` might be the name of a class that decorates a `Modem` with the ability to automatically hang up at the end of a session.

Patterns are just one kind of standard. In Java, for example, functions that convert objects to string representations are often named `toString`. It is better to follow conventions like these than to invent your own.

Teams will often invent their own standard system of names for a particular project. Eric Evans refers to this as a *ubiquitous language* for the project.[14] Your code should use the terms from this language extensively. In short, the more you can use names that are overloaded with special meanings that are relevant to your project, the easier it will be for readers to know what your code is talking about.

N4: *Unambiguous Names*

Choose names that make the workings of a function or variable unambiguous. Consider this example from FitNesse:

```
  private String doRename() throws Exception
{
 if(refactorReferences)
  renameReferences();
 renamePage();

 pathToRename.removeNameFromEnd();
 pathToRename.addNameToEnd(newName);
 return PathParser.render(pathToRename);
}
```

The name of this function does not say what the function does except in broad and vague terms. This is emphasized by the fact that there is a function named `renamePage` inside the function named `doRename`! What do the names tell you about the difference between the two functions? Nothing.

A better name for that function is `renamePageAndOptionallyAllReferences`. This may seem long, and it is, but it's only called from one place in the module, so it's explanatory value outweighs the length.

N5: *Use Long Names for Long Scopes*

The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.

Variable names like `i` and `j` are just fine if their scope is five lines long. Consider this snippet from the old standard "Bowling Game":

```
   private void rollMany(int n, int pins)
{
 for (int i=0; i<n; i++)
   g.roll(pins);
}
```

This is perfectly clear and would be obfuscated if the variable `i` were replaced with something annoying like `rollCount`. On the other hand, variables and functions with short names lose their meaning over long distances. So the longer the scope of the name, the longer and more precise the name should be.

N6: *Avoid Encodings*

Names should not be encoded with type or scope information. Prefixes such as `m_` or `f` are useless in today's environments. Also project and/or subsystem encodings such as `vis_` (for visual imaging system) are distracting and redundant. Again, today's environments provide all that information without having to mangle the names. Keep your names free of Hungarian pollution.

N7: *Names Should Describe Side-Effects*

Names should describe everything that a function, variable, or class is or does. Don't hide side effects with a name. Don't use a simple verb to describe a function that does more than just that simple action. For example, consider this code from TestNG:

```
  public ObjectOutputStream getOos() throws IOException {
 if (m_oos == null) {
  m_oos = new ObjectOutputStream(m_socket.getOutputStream());
 }
 return m_oos;
}
```

This function does a bit more than get an "oos"; it creates the "oos" if it hasn't been created already. Thus, a better name might be `createOrReturnOos`.

Tests

T1: *Insufficient Tests*

How many tests should be in a test suite? Unfortunately, the metric many programmers use is "That seems like enough." A test suite should test everything that could possibly break. The tests are insufficient so long as there are conditions that have not been explored by the tests or calculations that have not been validated.

T2: *Use a Coverage Tool!*

Coverage tools reports gaps in your testing strategy. They make it easy to find modules, classes, and functions that are insufficiently tested. Most IDEs give you a visual indication, marking lines that are covered in green and those that are uncovered in red. This makes it quick and easy to find `if` or `catch` statements whose bodies haven't been checked.

T3: *Don't Skip Trivial Tests*

They are easy to write and their documentary value is higher than the cost to produce them.

T4: *An Ignored Test Is a Question about an Ambiguity*

Sometimes we are uncertain about a behavioral detail because the requirements are unclear. We can express our question about the requirements as a test that is commented out, or as a test that annotated

with `@Ignore`. Which you choose depends upon whether the ambiguity is about something that would compile or not.

T5: *Test Boundary Conditions*

Take special care to test boundary conditions. We often get the middle of an algorithm right but misjudge the boundaries.

T6: *Exhaustively Test Near Bugs*

Bugs tend to congregate. When you find a bug in a function, it is wise to do an exhaustive test of that function. You'll probably find that the bug was not alone.

T7: *Patterns of Failure Are Revealing*

Sometimes you can diagnose a problem by finding patterns in the way the test cases fail. This is another argument for making the test cases as complete as possible. Complete test cases, ordered in a reasonable way, expose patterns.

As a simple example, suppose you noticed that all tests with an input larger than five characters failed? Or what if any test that passed a negative number into the second argument of a function failed? Sometimes just seeing the pattern of red and green on the test report is enough to spark the "Aha!" that leads to the solution. Look back at page 267 to see an interesting example of this in the `SerialDate` example.

T8: *Test Coverage Patterns Can Be Revealing*

Looking at the code that is or is not executed by the passing tests gives clues to why the failing tests fail.

T9: *Tests Should Be Fast*

A slow test is a test that won't get run. When things get tight, it's the slow tests that will be dropped from the suite. So *do what you must* to keep your tests fast.

Conclusion

This list of heuristics and smells could hardly be said to be complete. Indeed, I'm not sure that such a list can *ever* be complete. But perhaps completeness should not be the goal, because what this list *does* do is imply a value system.

Indeed, that value system has been the goal, and the topic, of this book. Clean code is not written by following a set of rules. You don't become a software craftsman by learning a list of heuristics. Professionalism and craftsmanship come from values that drive disciplines.

Bibliography

[Refactoring]: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

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

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

[Beck97]: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

[Beck07]: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2008.

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

[DDD]: *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.