Clean Code Notes

Table of contents

- Chapter 1 - Clean Code
- Chapter 2 - Meaningful Names
- Chapter 3 - Functions
- Chapter 4 - Comments
- Chapter 5 - Formatting
- Chapter 6 - Objects and Data Structures
- Chapter 7 - Error Handling
- Chapter 8 - Boundaries
- Chapter 9 - Unit Tests
- Chapter 10 - Classes
- Chapter 11 - Systems
- Chapter 12 - Emergence
- Chapter 13 - Concurrency
- Chapter 14 - Successive Refinement
- Chapter 15 - JUnit Internals
- Chapter 16 - Refactoring SerialDate
- Chapter 17 - Smells and Heuristics

Chapter 1 - Clean Code

This Book is about good programming. It's about how to write good code, and how to transform bad code into good code.

The code represents the detail of the requirements and the details cannot be ignored or abstracted. We may create languages that are closer to the requirements. We can create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision.

Why write bad code?

- Are you in a rush?
- Do you try to go "fast"?

- Do not you have time to do a good job?
- Are you tired of work in the same program/module?
- Does your Boss push you to finish soon?

The previous arguments could create a swamp of senseless code.

If you say "I will back to fix it later" you could fall in the LeBlanc's law "Later equals never"

You are a professional and the code is your responsibility. Let's analyze the following anecdote:

> What if you were a doctor and had a patient who demanded that you stop all the silly hand-washing in preparation for surgery because it was taking too much time? Clearly the patient is the boss; and yet the doctor should absolutely refuse to comply. Why? Because the doctor knows more than the patient about the risks of disease and infection. It would be unprofessional (never mind criminal) for the doctor to comply with the patient.

So too it is unprofessional for programmers to bend to the will of managers who don't understand the risks of making messes.

Maybe sometime you think in go fast to make the deadline. The only way to go fast is to keep the code as clean as possible at all times.

What is Clean Code?

Each experimented programmer has his/her own definition of clean code, but something is clear, a clean code is a code that you can read easily. The clean code is code that has been taken care of.

In his book Uncle Bob says the next:

> Consider this book a description of the Object Mentor School of Clean Code. The techniques and teachings within are the way that we practice our art. We are willing to claim that if you follow these teachings, you will enjoy the benefits that we have enjoyed, and you will learn to write code that is clean and professional. But don't make the mistake of thinking that we are somehow "right" in any absolute sense. There are other schools and other masters that have just as much claim to professionalism as we. It would behoove you to learn from them as well.

The boy Scout Rule

It's not enough to write the code well. The code has to be kept clean over time. We have all seen code rot and degrade as time passes. So we must take an active role in preventing this degradation.

It's a good practice apply the Boy Scout Rule

> Always leave the campground cleaner than you found it.

Chapter 2 - Meaningful Names

Names are everywhere in software. Files, directories, variables functions, etc. Because we do so much of it. We have better do it well.

Use Intention-Revealing Names

It is easy to say that names reveal intent. 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.

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 is used. **If a name requires a comment, then the name does not reveals its intent**.

Does not reveals intention	Reveals intention
`int d; // elapsed time in days`	`int elapsedTimeInDays`

Choosing names that reveal intent can make much easier to understand and change code. Example:

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

This code is simple, but create many questions:

1. What is the content of `theList`?
2. What is the significance of the item `x[0]` in the list?.
3. Why we compare `x[0]` vs `4`?
4. How would i use the returned list?

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 can refactor the previous code as follows:

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

```
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Now we know the next information:

1. `theList` represents the `gameBoard`
2. `x[0]` represents a cell in the board and `4` represents a flagged cell
3. The returned list represents the `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 improve the code writing a simple class for cells instead of using an array of `ints`. It can include an **intention-revealing function** (called it `isFlagged`) to hide the magic numbers. It results in a new function 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;
}
```

Avoid Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meaning vary from our intended meaning.

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. So `accountGroup` or `bunchOfAccounts` or just plain `accounts` would be better.

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

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 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. Example, you create the variable `klass` because the name `class` was used for something else.

In the next function, the arguments are noninformative, `a1` and `a2` doesn't provide clues to the author intention.

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

We can improve the code selecting more explicit argument names:

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

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.

Noise words are redundant. The word variable should never appear in a variable name. The word table should never appear in a table name.

Use Pronounceable Names

Imagine you have the variable `genymdhms` (Generation date, year, month, day, hour, minute and second) and imagine a conversation where you need talk about this variable calling it "gen why emm dee aich emm ess". You can consider convert a class like this:

```
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";
  /* ... */
};
```

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.

Avoid Encoding

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. Encoded names are seldom pronounceable and are easy to mis-type. An example of this, is the use of the Hungarian Notation or the use of member prefixes.

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`? Is preferable to leave interfaces unadorned.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.

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.

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.

Don't Be Cute

Cute name	Clean name
`holyHandGranade`	`deleteItems`
`whack`	`kill`
`eatMyShorts`	`abort`

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.

Don't Pun

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

Example: in a class use `add` for create a new value by adding or concatenating two existing values and in another class use `add` for put a simple parameter in a collection, it's a better options use a name like `insert` or `append` instead.

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.

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.

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

Variables like: `firstName`, `lastName`, `street`, `city`, `state`. Taken together it's pretty clear that they form an address, but, what if you saw the variable state being used alone in a method?, you could add context using prefixes like: `addrState` at least readers will understand that the variable is part of a large 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

Don’t Add Gratuitous Context

In an imaginary application called “Gas Station Deluxe,” it is a bad idea to prefix every class with GSD. Example: `GSDAccountAddress`

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

Chapter 3 - Functions

Functions are the first line of organization in any topic.

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.

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 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 easy to read and understand.

Do One Thing

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

Sections within Functions

If you have a function divided in sections like *declarations*, *initialization* etc, it's a obvious symptom of the function is 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.

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.

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,
- To include the setups, we include the suite setup if this is a suite, then we include the regul
- To include the suite setup, we search the parent hierarchy for the “SuiteSetUp” page and add an
- 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.

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.

Use Descriptive Names

> 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.

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.

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 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.

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.

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` !

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)`

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.

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.

Argument Objects

Compare:

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

vs

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

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.

Output Arguments

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.

Prefer Exceptions to Returning Error Codes

Returning error codes from command functions is a subtle violation of command query separation.

Don't Repeat Yourself

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.

Structured Programming

Some programmers follow Edsger Dijkstra's rules of structured programming. 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.

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

Chapter 4 - Comments

Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old comment that propagates lies and misinformation.

If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much—perhaps not at all.

Comments Do Not Make Up for Bad Code

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

Explain Yourself in Code

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

vs

```
if (employee.isEligibleForFullBenefits())
```

Good Comments

Some comments are necessary or beneficial. However the only truly good comment is the comment you found a way not to write.

Legal Comments

Sometimes our corporate coding standards force us to write certain comments for legal reasons. For example, copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file.

Informative Comments

It is sometimes useful to provide basic information with a comment. For example, consider this comment that explains the return value of an abstract method:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

A comment like this can sometimes be useful, but it is better to use the name of the function to convey the information where possible. For example, in this case the comment could be made redundant by renaming the function: `responderBeingTested`.

Explanation of Intent

Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. Example:

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // we are greater because we are the right type.
}
```

Clarification

Sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that's readable. In general it is better to find a way to make that argument or return value clear in its own right; but when its part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful.

Warning of concequences

Sometimes it is useful to warn other programmers about certain consequences.

```
// Don't run unless you
// have some time to kill.
public void _testWithReallyBigFile() {
  writeLinesToFile(10000000);
  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```

TODO Comments

It is sometimes reasonable to leave “To do” notes in the form of //TODO comments. In the following case, the TODO comment explains why the function has a degenerate implementation and what that function's future should be.

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception {
  return null;
}
```

TODOs are jobs that the programmer thinks should be done, but for some reason can’t do at the moment. It might be a reminder to delete a deprecated feature or a plea for someone else to look at a problem. It might be a request for someone else to think of a better name or a reminder to make a change that is dependent on a planned event. Whatever else a TODO might be, it is not an excuse to leave bad code in the system.

Amplification

A comment may be used to amplify the importance of something that may otherwise seem inconsequential.

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Javadocs in Public APIs

There is nothing quite so helpful and satisfying as a well-described public API. The javadocs for the standard Java library are a case in point. It would be difficult, at best, to write Java programs without them.

Bad Comments

Most comments fall into this category. Usually they are crutches or excuses for poor code or justifications for insufficient decisions, amounting to little more than the programmer talking to himself.

Mumbling

Plopping in a comment just because you feel you should or because the process requires it, is a hack. If you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write. Example:

```
public void loadProperties() {

  try {
    String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
    FileInputStream propertiesStream = new FileInputStream(propertiesPath);
    loadedProperties.load(propertiesStream);
  }
  catch(IOException e) {
    // No properties files means all defaults are loaded
  }
}
```

What does that comment in the catch block mean? Clearly meant something to the author, but the meaning not come though all that well. Apparently, if we get an `IOException`, it means that there was no properties file; and in that case all the defaults are loaded. But who loads all the defaults?

Redundant Comments

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis) throws Exception {
  if(!closed) {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

What purpose does this comment serve? It's certainly not more informative than the code. It does not justify the code, or provide intent or rationale. It is not easier to read than the code. Indeed, it is less precise than the code and entices the reader to accept that lack of precision in lieu of true understanding.

Misleading comments

Sometimes, with all the best intentions, a programmer makes a statement in his comments that isn't precise enough to be accurate. Consider for another moment the example of the previous section. The method does not return when `this.closed` becomes `true`. It returns if `this.closed` is `true`; otherwise, it waits for a blind time-out and then throws an exception if `this.closed` is still not true.

Mandated Comments

It is just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment. Comments like this just clutter up the code, propagate lies, and lend to general confusion and disorganization.

```
/**
*
* @param title The title of the CD
* @param author The author of the CD
* @param tracks The number of tracks on the CD
* @param durationInMinutes The duration of the CD in minutes
*/
public void addCD(String title, String author, int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

Journal Comments

Sometimes people add a comment to the start of a module every time they edit it. Example:

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new package com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate class is gone (DG); Ch
getFollowingDayOfWeek() and getNearestDayOfWeek() to correct bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
```

Today we have source code control systems, we don't need this type of logs.

Noise Comments

The comments in the follow examples doesn't provides new information.

```
/**
* Default constructor.
*/
protected AnnualDateRule() {
}
```

```
/** The day of the month. */
private int dayOfMonth;
```

Javadocs comments could enter in this category. Many times they are just redundant noisy comments written out of some misplaced desire to provide documentation.

Don't Use a Comment When You Can Use a Function or a Variable

Example:

```
// does the module from the global list <mod> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

vs

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Position Markers

This type of comments are noising

```
// Actions ///////////////////////////////
```

Closing Brace Comments

Example:

```
public class wc {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try {
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;

      } //while
      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);

    } // try
    catch (IOException e) {
      System.err.println("Error:" + e.getMessage());

    } //catch

  } //main
```

You could break the code in small functions instead to use this type of comments.

Attributions and Bylines

Example:

```
/* Added by Rick */
```

The VCS can manage this information instead.

Commented-Out Code

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

If you don't need anymore, please delete it, you can back later with your VCS if you need it again.

HTML Comments

HTML in source code comments is an abomination, as you can tell by reading the code below.

```
/**
* Task to run fit tests.
* This task runs fitnesse tests and publishes the results.
* <p/>
* <pre>
* Usage:
* &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
* classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
* classpathref=&quot;classpath&quot; /&gt;
* OR
* &lt;taskdef classpathref=&quot;classpath&quot;
* resource=&quot;tasks.properties&quot; /&gt;
* <p/>
* &lt;execute-fitnesse-tests
* suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
* fitnesseport=&quot;8082&quot;
* resultsdir=&quot;${results.dir}&quot;
* resultshtmlpage=&quot;fit-results.html&quot;
* classpathref=&quot;classpath&quot; /&gt;
* </pre>
*/
```

Nonlocal Information

If you must write a comment, then make sure it describes the code it appears near. Don't offer systemwide information in the context of a local comment.

Too Much Information

Don't put interesting historical discussions or irrelevant descriptions of details into your comments.

Inobvious Connection

The connection between a comment and the code it describes should be obvious. If you are going to the trouble to write a comment, then at least you'd like the reader to be able to look at the comment and the code and understand what the comment is talking about

Function Headers

Short functions don't need much description. A well-chosen name for a small function that does one thing is usually better than a comment header.

Javadocs in Nonpublic Code

Javadocs are for public APIs, in nonpublic code could be a distraction more than a help.

Chapter 5 - Formatting

Code formatting is important. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer's first order of business.

Vertical Formatting

Vertical Openness Between Concepts

This concept consist in how to you separate concepts in your code, In the next example we can appreciate it.

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
      Pattern.MULTILINE + Pattern.DOTALL
```

```
      );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
  Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text); match.find(); addChildWidgets(match.group(1));
  }
  public String render() throws Exception { StringBuffer html = new StringBuffer("<b>"); html.app
  }
}
```

As you can see, the readability of the first example is greater than that of the second.

Vertical Density

The vertical density implies close association. So lines of code that are tightly related should appear vertically dense. Check the follow example:

```
public class ReporterConfig {

    /**
     * The class name of the reporter listener */
    private String m_className;

    /**
```

```
     * The properties of the reporter listener */
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) { m_properties.add(property);
    }
```

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
}
```

The second code it's much easier to read. It fits in an "eye-full".

Vertical Distance

Variable Declarations. Variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear at the top of each function,

Instance variables, on the other hand, should be declared at the top of the class. This should not increase the vertical distance of these variables, because in a well-designed class, they are used by many, if not all, of the methods of the class.

There have been many debates over where instance variables should go. In C++ we commonly practiced the so-called scissors rule, which put all the instance variables at the bottom. The common convention in Java, however, is to put them all at the top of the class. I see no reason to follow any other convention. The important thing is for the instance variables to be declared in one well-known place. Everybody should know where to go to see the declarations.

Dependent Functions. If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use.

Conceptual Affinity. Certain bits of code want to be near other bits. They have a certain conceptual affinity. The stronger that affinity, the less vertical distance there should be between them.

Vertical Ordering

In general we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling. This creates a nice flow down the source

code module from high level to low level. *(This is the exact opposite of languages like Pascal, C, and C++ that enforce functions to be defined, or at least declared, before they are used)*

Horizontal Formatting

Horizontal Openness and Density

We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. Example:

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.

Horizontal Alignment

```
public class Example implements Base
{
  private   Socket      socket;
  private   inputStream input;
  protected long        requestProgress;

  public Expediter(Socket      s,
                   inputStream input) {
    this.socket =     s;
    this.input  =     input;
  }
}
```

In modern languages this type of alignment is not useful. The alignment seems to emphasize the wrong things and leads my eye away from the true intent.

```
public class Example implements Base
{
  private Socket socket;
  private inputStream input;
  protected longrequestProgress;
```

```
  public Expediter(Socket s, inputStream input) {

    this.socket = s;
    this.input = input;
  }
}
```

This is a better approach.

Indentation

The indentation it's important because help us to have a visible hierarchy and well defined blocks.

Team Rules

Every programmer has his own favorite formatting rules, but if he works in a team, then the team rules.

A team of developers should agree upon a single formatting style, and then every member of that team should use that style. We want the software to have a consistent style. We don't want it to appear to have been written by a bunch of disagreeing individuals.

Chapter 6 - Objects and Data Structures

Data Abstraction

Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation.

Data/Object Anti-Symmetry

These two examples show the difference between objects and data structures. Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions.

Procedural Shape

```
public class Square {
  public Point topLeft;
  public double side;
```

```
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) { Rectangle r = (Rectangle)shape; return r.height * r.wi
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Polymorphic Shape

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;
```

```
  public double area() {
    return height * width;
  }
}

public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

Again, we see the complimentary nature of these two definitions; they are virtual opposites! This exposes the fundamental dichotomy between objects and data structures:

> Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.

The complement is also true:

> Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.

Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.

The Law of Demeter

There is a well-known heuristic called the Law of Demeter that says a module should not know about the innards of the objects it manipulates.

More precisely, the Law of Demeter says that a method `f` of a class `C` should only call the methods of these:

- `C`
- An object created by `f`
- An object passed as an argument to `f`
- An object held in an instance variable of `C`

The method should not invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.

Data Transfer Objects

The quintessential form of a data structure is a class with public variables and no functions. This is sometimes called a data transfer object, or DTO. DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets, and so on. They often become the first in a series of translation stages that convert raw data in a database into objects in the application code.

Chapter 7 - Error Handling

Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it obscures logic, it's wrong.

Use Exceptions Rather Than Return Codes

Back in the distant past there were many languages that didn't have exceptions. In those languages the techniques for handling and reporting errors were limited. You either set an error flag or returned an error code that the caller could check

Write Your Try-Catch-Finally Statement First

In a way, try blocks are like transactions. Your catch has to leave your program in a consistent state, no matter what happens in the try. For this reason it is good practice to start with a try-catch-finally statement when you are writing code that could throw exceptions. This helps you define what the user of that code should expect, no matter what goes wrong with the code that is executed in the try.

Provide Context with Exceptions

Each exception that you throw should provide enough context to determine the source and location of an error.

Create informative error messages and pass them along with your exceptions. Mention the operation that failed and the type of failure. If you are logging in your application, pass along enough information to be able to log the error in your catch.

Don't Return Null

If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead. If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.

Don't Pass Null

Returning null from methods is bad, but passing null into methods is worse. Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.

Chapter 8 - Boundaries

We seldom control all the software in our systems. Sometimes we buy third-party pack- ages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code with our own.

Using Third-Party Code

There is a natural tension between the provider of an interface and the user of an interface. Providers of third-party packages and frameworks strive for broad applicability so they can work in many environments and appeal to a wide audience. Users, on the other hand, want an interface that is focused on their particular needs. This tension can cause problems at the boundaries of our systems. Example:

```
Map sensors = new HashMap();
Sensor s = (Sensor)sensors.get(sensorId);
```

VS

```
public class Sensors {
  private Map sensors = new HashMap();

  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }
  //snip
}
```

The first code exposes the casting in the Map, while the second is able to evolve with very little impact on the rest of the application. The casting and type management is handled inside the Sensors class.

The interface is also tailored and constrained to meet the needs of the application. It results in code that is easier to understand and harder to misuse. The Sensors class can enforce design and business rules.

Exploring and Learning Boundaries

Third-party code helps us get more functionality delivered in less time. Where do we start when we want to utilize some third-party package? It’s not our job to test the third-party code, but it may be in our best interest to write tests for the third-party code we use.

It's a good idea write some test for learn and understand how to use a third-party code. Newkirk calls such tests learning tests.

Learning Tests Are Better Than Free

The learning tests end up costing nothing. We had to learn the API anyway, and writing those tests was an easy and isolated way to get that knowledge. The learning tests were precise experiments that helped increase our understanding.

Not only are learning tests free, they have a positive return on investment. When there are new releases of the third-party package, we run the learning tests to see whether there are behavioral differences.

Using Code That Does Not Yet Exist

Some times it's necessary work in a module that will be connected to another module under develop, and we have no idea about how to send the information, because the API had not been designed yet. In this cases it's recommendable create an interface for encapsulate the communication with the pending module. In this way we maintain the control over our module and we can test although the second module isn't available yet.

Clean Boundaries

Interesting things happen at boundaries. Change is one of those things. Good software designs accommodate change without huge investments and rework. When we use code that is out of our control, special care must be taken to protect our investment and make sure future change is not too costly.

Chapter 9 - Unit Tests

Test **D**riven **D**evelopment

The Three Laws of TDD

- **First Law** You may not write production code until you have written a failing unit test.
- **Second Law** You may not write more of a unit tests than is sufficient to fail, and not comipling is failing.

- **Third Law** You may not write more production code than is sufficient to pass the currently failing tests.

Clean Tests

If you don't keep your tests clean, you will lose them.

The readability it's very important to keep clean your tests.

One Assert per test

It's recomendable maintain only one asserts per tests, because this helps to maintain each tests easy to understand.

Single concept per Test

This rule will help you to keep short functions.

- **Write one test per each concept that you need to verify**

F.I.R.S.T

- **Fast** Test should be fast.
- **Independient** Test should not depend on each other.
- **Repeatable** Test Should be repeatable in any environment.
- **Self-Validating** Test should have a boolean output. either they pass or fail.
- **Timely** Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test.

Chapter 10 - Classes

Class Organization

Encapsulation

We like to keep our variables and utility functions small, but we're not fanatic about it. Sometimes we need to make a variable or utility function protected so that it can be accessed by a test.

Classes Should be Small

- First Rule: Classes should be small
- Second Rule: **Classes should be smaller than the first rule**

The Single Responsibility Principle

Classes should have one responsibility - one reason to change

SRP is one of the more important concept in OO design. It's also one of the simple concepts to understand and adhere to.

Cohesion

Classes Should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. In general the more variables a method manipulates the more cohesive that method is to its class. A class in which each variable is used by each method is maximally cohesive.

Maintaining Cohesion Results in Many Small Classes

Just the act of breaking large functions into smaller functions causes a proliferation of classes.

Organizing for change

For most systems, change is continual. Every change subjects us to the risk that the remainder of the system no longer works as intended. In a clean system we organize our classes so as to reduce the risk of change.

Isolating from Change

Needs will change, therefore code will change. We learned in OO 101 that there are concrete classes, which contain implementation details (code), and abstract classes, which represent concepts only. A client class depending upon concrete details is at risk when those details change. We can introduce intefaces and abstract classes to help isolate the impact of those details.

Chapter 11 - Systems

Separe Constructing a System from using It

Software Systems should separate the startuo process, when the application objects are constructed and the dependencies are "wired" thogether, from the runtime logic that takes over after startup

Separation from main

One way to separate construction from use is simply to move all aspects of construction to `main`, or modules called by `main`, and to design the rest of the system assuming that all objects have been created constructed and wired up appropriately.

The Abstract Factory Pattern is an option for this kind of approach.

Dependency Injection

A powerful mechanism for separating construction from use is Dependency Injection (DI), the application of Inversion of control (IoC) to dependency management. Inversion of control moves secondary responsibilities from an object to other objects that are dedicated to the purpose, thereby supporting the Single Responsibility Principle. In context of dependency management, an object should not take responsibility for instantiating dependencies itself. Instead, it, should pass this responsibility to another "authoritative" mechanism, thereby inverting the control. Because setup is a global concern, this authoritative mechanism will usually be either the "main" routine or a special-purpose *container*.

Chapter 12 - Emergence

According to Kent Beck, a design is "simple" if it follows these rules

- Run all tests
- Contains no duplication
- Expresses the intent of programmers
- Minimizes the number of classes and methods

Chapter 13 - Concurrency

Concurrence is a decoupling strategy. It helps us decouple what gets fone from when it gets done. In single-threaded applications what and when are so strongly coupled that the state of the entire application can often be determined by looking at the stack backtrace. A programmer who debugs such a system can set a breakpoint, or a sequence of breakpoints, and know the state of the system by which breakpoints are hit.

Decoupling what from when can dramatically improve both the throughput and structures of an application. From a structural point of view the application looks like many lit- tle collaborating computers rather than one big main loop. This can make the system easier to understand and offers some powerful ways to separate concerns.

Miths and Misconceptions

- Concurrency always improves performance. Concurrency can sometimes improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situ- ation is trivial.
- Design does not change when writing concurrent programs. In fact, the design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of what from when usually has a huge effect on the structure of the system.
- Understanding concurrency issues is not important when working with a container such as a Web or EJB container. In fact, you'd better know just what your container is doing and how to guard against the issues of concurrent update and deadlock described later in this chapter. Here are a few more balanced sound bites regarding writing concurrent software:
- Concurrency incurs some overhead, both in performance as well as writing additional code.
- Correct concurrency is complex, even for simple problems.
- Concurrency bugs aren't usually repeatable, so they are often ignored as one-offs instead of the true defects they are.
- Concurrency often requires a fundamental change in design strategy.

Chapter 14 - Successive Refinement

This chapter is a study case. It's recommendable to completely read it to understand more.

Chapter 15 - JUnit Internals

This chapter analize the JUnit tool. It's recommendable to completely read it to understand more.

Chapter 16 - Refactoring SerialDate

This chapter is a study case. It's recommendable to completely read it to understand more.

Chapter 17 - Smells and Heuristics

A reference of code smells from Martin Fowler's *Refactoring* and Robert C Martin's *Clean Code*.

While clean code comes from discipline and not a list or value system, here is a starting point.

Comments

C1: Inappropriate Information

Reserve comments for technical notes referring to code and design.

C2: Obsolete Comment

Update or delete obsolete comments.

C3: Redundant Comment

A redundant comment describes something able to sufficiently describe itself.

C4: Poorly Written Comment

Comments should be brief, concise, correctly spelled.

C5: Commented-Out Code

Ghost code. Delete it.

Environment

E1: Build Requires More Than One Step

Builds should require one command to check out and one command to run.

E2: Tests Require More Than One Step

Tests should be run with one button click through an IDE, or else with one command.

Functions

F1: Too Many Arguments

Functions should have no arguments, then one, then two, then three. No more than three.

F2: Output Arguments

Arguments are inputs, not outputs. If somethings state must be changed, let it be the state of the called object.

F3: Flag Arguments

Eliminate boolean arguments.

F4: Dead Function

Discard uncalled methods. This is dead code.

General

G1: Multiple Languages in One Source File

Minimize the number of languages in a source file. Ideally, only one.

G2: Obvious Behavior is Unimplemented

The result of a function or class should not be a surprise.

G3: Incorrect Behavior at the Boundaries

Write tests for every boundary condition.

G4: Overridden Safeties

Overriding safeties and exerting manual control leads to code melt down.

G5: Duplication

Practice abstraction on duplicate code. Replace repetitive functions with polymorphism.

G6: Code at Wrong Level of Abstraction

Make sure abstracted code is separated into different containers.

G7: Base Classes Depending on Their Derivatives

Practice modularity.

G8: Too Much Information

Do a lot with a little. Limit the amount of *things* going on in a class or functions.

G9: Dead Code

Delete unexecuted code.

G10: Vertical Separation

Define variables and functions close to where they are called.

G11: Inconsistency

Choose a convention, and follow it. Remember no surprises.

G12: Clutter

Dead code.

G13: Artificial Coupling

Favor code that is clear, rather than convenient. Do not group code that favors mental mapping over clearness.

G14: Feature Envy

Methods of one class should not be interested with the methods of another class.

G15: Selector Arguments

Do not flaunt false arguments at the end of functions.

G16: Obscured Intent

Code should not be magic or obscure.

G17: Misplaced Responsibility

Use clear function name as waypoints for where to place your code.

G18: Inappropriate Static

Make your functions nonstatic.

G19: Use Explanatory Variables

Make explanatory variables, and lots of them.

G20: Function Names Should Say What They Do

...

G21: Understand the Algorithm

Understand how a function works. Passing tests is not enough. Refactoring a function can lead to a better understanding of it.

G22: Make Logical Dependencies Physical

Understand what your code is doing.

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

Avoid the brute force of switch/case.

G24: Follow Standard Conventions

It doesn't matter what your teams convention is. Just that you have on and everyone follows it.

G25: Replace Magic Numbers with Named Constants

Stop spelling out numbers.

G26: Be Precise

Don't be lazy. Think of possible results, then cover and test them.

G27: Structure Over Convention

Design decisions should have a structure rather than a dogma.

G28: Encapsulate Conditionals

Make your conditionals more precise.

G29: Avoid Negative Conditionals

Negative conditionals take more brain power to understand than a positive.

G31: Hidden Temporal Couplings

Use arguents that make temporal coupling explicit.

G32: Don’t Be Arbitrary

Your code's sturcture should communicate the reason for its structure.

G33: Encapsulate Boundary Conditions

Avoid leaking +1's and -1's into your code.

G34: Functions Should Descend Only One Level of Abstraction

The toughest heuristic to follow. One level of abstraction below the function's described operation can help clarify your code.

G35: Keep Configurable Data at High Levels

High level constants are easy to change.

G36: Avoid Transitive Navigation

Write shy code. Modules should only know about their neighbors, not their neighbor's neighbors.

Names

N1: Choose Descriptive Names

Choose names that are descriptive and relevant.

N2: Choose Names at the Appropriate Level of Abstraction

Think of names that are still clear to the user when used in different programs.

N3: Use Standard Nomenclature Where Possible

Use names that express their task.

N4: Unambiguous Names

Favor clearness over curtness. A long, expressive name is better than a short, dull one.

N5: Use Long Names for Long Scopes

A name's length should relate to its scope.

N6: Avoid Encodings

No not encode names with type or scope information.

N7: Names Should Describe Side-Effects

Consider the side-effects of your function, and include that in its name.

Tests

T1: Insufficient Tests

Test everything that can possibly break

T2: Use a Coverage Tool!

Use your IDE as a coverage tool.

T3: Don’t Skip Trivial Tests

...

T4: An Ignored Test is a Question about an Ambiguity

If your test is ignored, the code is brought into question.

T5: Test Boundary Conditions

The middle is usually covered. Remember the boundaries.

T6: Exhaustively Test Near Bugs

Bugs are rarely alone. When you find one, look nearby for another.

T7: Patterns of Failure Are Revealing

Test cases ordered well will reveal patterns of faiure.

T8: Test Coverage Patterns Can Be Revealing

Similarly, look at the code that is or is not passed in a failure.

T9: Tests Should Be Fast

Slow tests won't get run.