



# **Code Review Guidebook**

**A Java Guide**

**Jack Liu**

Copyright © 2019 Jiaqi Liu

PUBLISHED BY PUBLISHER

BOOK-WEBSITE.COM

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

*First printing, March 2019*

# Contents

I	Part One - Reviewing Code	
1	Communication .....	11
1.1	Know What You Want to Say	11
1.2	Know the Code Author	11
1.3	Choose a Style	11
1.4	Make It Look Good	12
1.5	Be a Listener	12
2	You Are Partially Responsible for the Code .....	13
3	Keep Entropy in Mind while Reviewing .....	15
3.1	What is Software Entropy?	15
4	Size of Pull Request .....	17
4.1	How Big is Appropriate?	17
5	Avoid Duplication .....	19
5.1	DRY	19
5.2	How Does Duplication Arise?	19
6	Meaningful Names .....	21
6.1	Use Intention-Revealing Names	21
6.1.1	Variable Declaration should not <i>require</i> a comment .....	21

6.1.2	Promote simplicity and <b>IMPLICITITY</b> .....	22
<b>6.2</b>	<b>Avoid Disinformation</b>	<b>22</b>
<b>6.3</b>	<b>Make Meaningful Distinctions</b>	<b>23</b>
6.3.1	Use Speakable/Pronouncable Names .....	23
<b>6.4</b>	<b>Use Searchable Names</b>	<b>23</b>
<b>6.5</b>	<b>Dont Add Gratuitous Context</b>	<b>23</b>
<b>7</b>	<b>Functions</b> .....	<b>25</b>
<b>7.1</b>	<b>Blocks and Indenting</b>	<b>26</b>
<b>7.2</b>	<b>Do One Thing</b>	<b>26</b>
7.2.1	One Level of Abstraction per Function .....	27
<b>7.3</b>	<b>Reading Code from Top to Bottom: The Stepdown Rule</b>	<b>27</b>
<b>7.4</b>	<b>Switch Statements</b>	<b>28</b>
<b>7.5</b>	<b>Function Arguments</b>	<b>29</b>
7.5.1	Common Monadic Forms .....	30
7.5.2	Flag Arguments .....	30
7.5.3	Dyadic Functions .....	30
<b>7.6</b>	<b>Argument Objects</b>	<b>31</b>
<b>7.7</b>	<b>Argument Lists</b>	<b>31</b>
<b>7.8</b>	<b>Coherent method name and arguments</b>	<b>31</b>
<b>7.9</b>	<b>Have No Side Effects</b>	<b>32</b>
7.9.1	What is Side Effects .....	32
7.9.2	Example .....	32
7.9.3	Output Arguments .....	32
<b>7.10</b>	<b>Command Query Separation</b>	<b>33</b>
<b>7.11</b>	<b>Prefer Exceptions to Returning Error Codes</b>	<b>33</b>
<b>7.12</b>	<b>Extract Try/Catch Blocks</b>	<b>35</b>
<b>8</b>	<b>Constructors</b> .....	<b>37</b>
<b>8.1</b>	<b>Constructors in general</b>	<b>37</b>
<b>8.2</b>	<b>Do Not Pass 'this' out of a Constructor</b>	<b>37</b>
<b>8.3</b>	<b>Avoid JavaBeans Style of Construction</b>	<b>39</b>
<b>8.4</b>	<b>Initializing fields to 0, false, or null is redundant</b>	<b>40</b>
<b>8.5</b>	<b>Constructors Shouldn't Start Threads</b>	<b>42</b>
<b>9</b>	<b>Comments</b> .....	<b>43</b>
<b>9.1</b>	<b>Comments Do Not Make Up for Bad Code</b>	<b>43</b>
<b>9.2</b>	<b>Explain Yourself as Code</b>	<b>43</b>
<b>9.3</b>	<b>Good Comments</b>	<b>44</b>
9.3.1	Legal Comments .....	44
9.3.2	Informative Comments .....	44
9.3.3	Explanation of Intent .....	44
9.3.4	Clarification .....	45
9.3.5	Warning of Consequences .....	45

9.3.6	TODO Comments . . . . .	45
9.3.7	Javadocs in Public APIs . . . . .	46
<b>9.4</b>	<b>Bad Comments</b>	<b>46</b>
9.4.1	Confusing Comments . . . . .	46
9.4.2	Redundant Comments . . . . .	46
9.4.3	Dont Use a Comment When You Can Use a Function or a Variable . . . . .	48
9.4.4	Position Markers . . . . .	48
9.4.5	Closing Brace Comments . . . . .	49
9.4.6	Attributions and Bylines . . . . .	49
9.4.7	Commented-Out Code . . . . .	50
9.4.8	HTML Comments . . . . .	50
9.4.9	Nonlocal Information . . . . .	50
9.4.10	Inobvious Connection . . . . .	51
<b>10</b>	<b>Formatting . . . . .</b>	<b>53</b>
10.1	Size of a File	53
10.2	The Newspaper Metaphor	53
10.3	Vertical Density	53
10.4	Dependent Functions	54
10.5	Conceptual Affinity	54
10.6	Vertical Ordering	54
<b>11</b>	<b>Objects and Data Structures . . . . .</b>	<b>55</b>
11.1	Data Abstraction	55
11.2	Data/Object Anti-Symmetry(VERY IMPORTANT)	55
11.3	The Law of Demeter	57
11.4	Data Transfer Objects	58
<b>12</b>	<b>Error-handling . . . . .</b>	<b>59</b>
12.1	Use Unchecked Exceptions	59
12.2	Define the Normal Flow	59
12.3	Dont Return Null	60
<b>13</b>	<b>Boundaries . . . . .</b>	<b>61</b>
13.1	Using Third-Party Code	61
13.2	Exploring and Learning Boundaries	62
<b>14</b>	<b>Unit Tests . . . . .</b>	<b>63</b>
14.1	The Three Laws of TDD(Test Driven Development)	63
14.2	Clean Tests	63
14.3	F.I.R.S.T.	64

<b>15</b>	<b>Classes</b>	<b>65</b>
<b>15.1</b>	<b>Classes Should Be Small!</b>	<b>65</b>
15.1.1	The Single Responsibility Principle	65
15.1.2	Cohesion	66
<b>15.2</b>	<b>Organizing for Change</b>	<b>66</b>
<b>16</b>	<b>Systems</b>	<b>67</b>
<b>16.1</b>	<b>Scaling Up</b>	<b>67</b>
16.1.1	Approaches for Separation of Concerns	67
<b>17</b>	<b>Concurrency</b>	<b>69</b>
<b>17.1</b>	<b>Concurrency Defense Principles</b>	<b>69</b>
17.1.1	Single Responsibility Principle	69
17.1.2	Limit the Scope of Data	69
17.1.3	Use Copies of Data	70
17.1.4	Threads Should Be as Independent as Possible	70
<b>17.2</b>	<b>Know Library</b>	<b>70</b>
<b>17.3</b>	<b>Review for Execution Models</b>	<b>70</b>
<b>17.4</b>	<b>Keep Synchronized Sections Small</b>	<b>71</b>
<b>18</b>	<b>What is Good-Enough?</b>	<b>73</b>
<b>18.1</b>	<b>Keep Your Users in the Trade-Off</b>	<b>73</b>

## II

## Part Two - Reviewing Architecture

<b>19</b>	<b>What is Design and Architecture?</b>	<b>77</b>
<b>19.1</b>	<b>What is a Good Architecture?</b>	<b>77</b>
<b>20</b>	<b>Text Chapter</b>	<b>79</b>
<b>20.1</b>	<b>Paragraphs of Text</b>	<b>79</b>
<b>20.2</b>	<b>Citation</b>	<b>80</b>
<b>20.3</b>	<b>Lists</b>	<b>80</b>
20.3.1	Numbered List	80
20.3.2	Bullet Points	80
20.3.3	Descriptions and Definitions	80
<b>21</b>	<b>In-text Elements</b>	<b>81</b>
<b>21.1</b>	<b>Theorems</b>	<b>81</b>
21.1.1	Several equations	81
21.1.2	Single Line	81
<b>21.2</b>	<b>Definitions</b>	<b>81</b>
<b>21.3</b>	<b>Notations</b>	<b>82</b>
<b>21.4</b>	<b>Remarks</b>	<b>82</b>
<b>21.5</b>	<b>Corollaries</b>	<b>82</b>

<b>21.6</b>	<b>Propositions</b>	<b>82</b>
21.6.1	Several equations . . . . .	82
21.6.2	Single Line . . . . .	82
<b>21.7</b>	<b>Examples</b>	<b>82</b>
21.7.1	Equation and Text . . . . .	82
21.7.2	Paragraph of Text . . . . .	83
<b>21.8</b>	<b>Exercises</b>	<b>83</b>
<b>21.9</b>	<b>Problems</b>	<b>83</b>
<b>21.10</b>	<b>Vocabulary</b>	<b>83</b>

### III

## Part Two

<b>22</b>	<b>Presenting Information</b> . . . . .	<b>87</b>
<b>22.1</b>	<b>Table</b>	<b>87</b>
<b>22.2</b>	<b>Figure</b>	<b>87</b>
	<b>Bibliography</b> . . . . .	<b>89</b>
	Articles	89
	Books	89
	<b>Index</b> . . . . .	<b>91</b>





# Part One - Reviewing Code

7.9	Have No Side Effects	
7.10	Command Query Separation	
7.11	Prefer Exceptions to Returning Error Codes	
7.12	Extract Try/Catch Blocks	
<b>8</b>	<b>Constructors</b>	<b>37</b>
8.1	Constructors in general	
8.2	Do Not Pass 'this' out of a Constructor	
8.3	Do Not Inherit or Call a Constructor	
8.4	Initializing fields to 0, false, or null is redundant	
8.5	Constructors Shouldn't Start Threads	
<b>9</b>	<b>Comments</b>	<b>43</b>
9.1	Comments Do Not Make Up for Bad Code	
9.2	Explain Yourself as Code	
9.3	Good Comments	
9.4	Bad Comments	
<b>10</b>	<b>Formatting</b>	<b>53</b>
10.1	Size of a File	
10.2	The Newspaper Metaphor	
10.3	Vertical Density	
10.4	Dependent Functions	
10.5	Conceptual Affinity	
10.6	Vertical Ordering	
<b>11</b>	<b>Objects and Data Structures</b>	<b>55</b>
11.1	Data Abstraction	
11.2	Data/Object Anti-Symmetry(VERY IMPORTANT)	
11.3	The Law of Demeter	
11.4	Data Transfer Objects	
<b>12</b>	<b>Error-handling</b>	<b>59</b>
12.1	Use Unchecked Exceptions	
12.2	Define the Normal Flow	
12.3	Dont Return Null	
<b>13</b>	<b>Boundaries</b>	<b>61</b>
13.1	Using Third-Party Code	
13.2	Exploring and Learning Boundaries	
<b>14</b>	<b>Unit Tests</b>	<b>63</b>
14.1	The Three Laws of TDD(Test Driven Development)	
14.2	Clean Tests	
14.3	F.I.R.S.T.	
<b>15</b>	<b>Classes</b>	<b>65</b>
15.1	Classes Should Be Small!	
15.2	Organizing for Change	
<b>16</b>	<b>Systems</b>	<b>67</b>
16.1	Scaling Up	
<b>17</b>	<b>Concurrency</b>	<b>69</b>
17.1	Concurrency Defense Principles	
17.2	Know Library	
17.3	Review for Execution Models	
17.4	Keep Synchronized Sections Small	
<b>18</b>	<b>What is Good-Enough?</b>	<b>73</b>
18.1	Keep Your Users in the Trade-Off	





# 1. Communication

## 1.1 Know What You Want to Say

Probably the most difficult part of the more formal styles of code review comment used in business is working out exactly what it is you want to say.

Plan what you want to say. Write an outline. Then ask yourself, Does this get across whatever I'm trying to say? Refine it until it does. Jot down the ideas you want to communicate, and plan a couple of strategies for getting them across.

## 1.2 Know the Code Author

You're communicating only if you're conveying information (code changes). The following technique helps you to structure your code review comments:

The **WISDOM** acrostic - understanding the audience (code author)

What do you want them to learn?  
What is their **i**nterest in what you've got to say?  
How **s**ophisticated are they?  
How much **d**etail do they want?  
Whom do you want to **o**wn the information?  
How can you **m**otivate them to listen to you?

## 1.3 Choose a Style

Adjust the style of your delivery to suit your audience. Some people want a formal just the facts briefing. Others like a long, wide-ranging chat before getting down to business. When it comes to written documents, some like to receive large bound reports, while others expect a simple memo or e-mail. If in doubt, ask.

Thinking about the other side's style is important, keeping you, one side of communication as well, clear is also important.

## 1.4 Make It Look Good

Your ideas are important. They deserve a good-looking vehicle to convey them to your audience.

Too many developers (i.e. code reviewers) concentrate solely on content when producing written code review comments. We think this is a mistake. Any chef will tell you that you can slave in the kitchen for hours only to ruin your efforts with poor presentation.

There is no excuse today for producing poor-looking code reviews. Modern word processors (along with layout systems such as markdown) can produce stunning output. You need to learn just a few basic rules.

Making a good-looking review encourages not only you, but also the other side to read it with enjoyable experiences.

## 1.5 Be a Listener

There's one technique that you must use if you want people to listen to you: listen to them. Even if this is a situation where you have all the information, even if this is a formal meeting with you standing in front of 20 suits if you don't listen to them, they won't listen to you.

Encourage people to talk by asking questions. Turn the meeting into a dialog, and you'll make your point more effectively. Who knows, you might even learn something.

A close-up photograph of a red rose, showing the intricate details of its petals and the sharp thorns. The rose is the central focus, with its vibrant red color contrasting against a blurred background of more roses and green foliage.

## 2. You Are Partially Responsible for the Code

Responsibility is something you actively agree to. You make a commitment, by reviewing code, to ensure that something is done right, but you don't necessarily have direct control over every aspect of it. In addition to doing your own personal best, you must analyze the situation for risks that are beyond your control.

When you do accept the responsibility (because you are reviewing) for some codes, you should expect to be held accountable for it. When you see a mistake (as all programmers do) or an error in judgment, speak it out honestly and try to offer options for code authors.

Don't blame someone or something else, it is up to you to provide solutions.



Provide Options, Don't Make lame Excuses





## 3. Keep Entropy in Mind while Reviewing

### 3.1 What is Software Entropy?

While software development is immune from almost all physical laws, entropy hits us hard. Entropy is a term from physics that refers to the amount of disorder in a system. Unfortunately, the laws of thermodynamics guarantee that the entropy in the universe tends toward a maximum. When disorder increases in software, programmers call it software rot.

There are many factors that can contribute to software rot. The most important one seems to be the psychology, or culture, at work on a project. Even if you are a team of one, your projects psychology can be a very delicate thing. Despite the best laid plans and the best people, a project can still experience ruin and decay during its lifetime. Yet there are other projects that, despite enormous difficulties and constant setbacks, successfully fight nature's tendency toward disorder and manage to come out pretty well.

What makes the difference? The broken-window theory explains it.



Spot "Broken Windows" and ask code author to fix it right away.







## 4. Size of Pull Request

### 4.1 How Big is Appropriate?

The story of Stone Soup and Boiled Frogs([https://en.wikipedia.org/wiki/Stone\\_Soup](https://en.wikipedia.org/wiki/Stone_Soup)) told us one thing about dealing with politics of software developments:

You may be in a situation where you know exactly what needs doing and how to do it. The entire system just appears before your eyes you know its right. But ask permission to tackle the whole thing and you'll be met with delays and blank stares. People will form committees, budgets will need approval, and things will get complicated. Everyone will guard their own resources. Sometimes this is called start-up fatigue.

Its time to bring out the stones. Work out what you can reasonably ask for. Develop it well. Once you've got it, show people, and let them marvel. Then say of course, it would be better if we added . Pretend its not important. Sit back and wait for them to start asking you to add the functionality you originally wanted. People find it easier to join an ongoing success. Show them a glimpse of the future and you'll get them to rally around

Code reviewers should judge the size of a PR using SS story as a guidance, is this change small enough to start something which can be shown to people? If it's too big, as code author to break it into multiple PR's



#### Be a Catalyst for Change


Keeping the PR size small has another advantage. Consider the other side. The villagers think about the stones and forget about the rest of the world. We all fall for it, every day. Things just creep up on us.

We've all seen the symptoms. Projects slowly and inexorably get totally out of hand. Most software disasters start out too small to notice, and most project overruns happen a day at a time. Systems drift from their specifications feature by feature, while patch after patch gets added to a piece of code until there's nothing of the original left. Its often the accumulation of small things that breaks morale and teams.



#### As a code reviewer, always remember the Big Picture





## 5. Avoid Duplication

### 5.1 DRY

Unfortunately, knowledge isn't stable. It changes often rapidly. Your understanding of a requirement may change following a meeting with the client. The government changes a regulation and some business logic gets outdated. Tests may show that the chosen algorithm won't work. All this instability means that we spend a large part of our time in maintenance mode, reorganizing and reexpressing the knowledge in our systems.

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong. Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we were designing or coding. Perhaps the environment changes. Whatever the reason, maintenance is not a discrete activity, but a routine part of the entire development process.

When we perform maintenance, we have to find and change the representations of things—those capsules of knowledge embedded in the application. The problem is that it's easy to duplicate knowledge in the specifications, processes, and programs that we develop, and when we do so, we invite a maintenance nightmare—one that starts well before the application ships.

We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain, is to follow what we call the DRY (Don't Repeat Yourself) principle:

**R** EVERY PIECE OF KNOWLEDGE MUST HAVE A SINGLE, UNAMBIGUOUS, AUTHORITATIVE REPRESENTATION WITHIN A SYSTEM.



On duplicated codes, ask code author to remove it

### 5.2 How Does Duplication Arise?

Most of the duplication we see falls into one of the following categories:

- **Imposed duplication** Developers feel they have no choice—the environment seems to require duplication.

- **Inadvertent duplication** Developers dont realize that they are duplicating information.
- **Impatient duplication** Developers get lazy and duplicate because it seems easier.
- **Inter-developer duplication** Multiple people on a team (or on different teams) duplicate a piece of information.

## 6. Meaningful Names

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

Follow the rules above and do not be afraid to make changes on other's code accordingly. It pays in both short and long terms

### 6.1 Use Intention-Revealing Names

Everyone who reads your code (including you) will be happier if you take care with your naming.

#### 6.1.1 Variable Declaration should not *require* a comment

While it is a good Java practice to comment variables of a class, the name of a variable, function, or class, should have already answered all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

**! If a name requires a comment, then the name does not reveal its intent.**

For example:

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



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



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



### 6.1.2 Promote simplicity and IMPLICITITY

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();

    for (int[] x : theList) {
        if (x[0] == 4) {
            list1.add(x);
        }
    }

    return list1;
}
```



Why is it hard to tell what this code is doing? Although code syntax is not complicated, the problem is not the simplicity of the code but the *implicit* of the code.

**Definition 6.1.1 — Implicit of the Code.** The degree to which the context is not explicit in the code itself

The code implicitly requires that we know the answers to questions such as:

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

The answers to these questions are not present in the code sample, but they could have been.

! The code review should point out the problem and the re-review should make sure that simplicity of the code has not changed.

## 6.2 Avoid Disinformation

### Sub-type Naming

Java usually declares variables by type and always assigns variables with concrete implementation. A variable should not be named by a sub-type name if the type could be different type nature. For example, when a variable's type is Collection and is assigned a ArrayList, it should not be named as something like peopleList; instead peopleGroup or simply persons would be better.

! A variable should not be named by a sub-type name if the type could be different type nature.

### Avoid Names That Vary in Small Ways

Long and similarly-named variables are very hard for people to read and distinguish. For example How long does it take to spot the subtle difference between a XYZControllerForEfficientHandlingOfStrings in one module and, somewhere a little more distant, XYZControllerForEfficientStorageOfStrings?

In addition, developer using IDE uses auto-complete feature. They will often blindly pick up the wrong variable simply because the two looks so similar.

## 6.3 Make Meaningful Distinctions

By all possible means, people could name two things with non-meaningful distinctions. For example, class Product v.s. class ProductInfo. What are the difference between the two? Product can have product info, too. Same issues goes for methods:

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



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

### 6.3.1 Use Speakable/Pronounceable Names

- R** Humans are good at words. A significant part of our brains is dedicated to the concept of words. Words are pronounceable

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



Use Speakable/Pronounceable Names

## 6.4 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.

My personal preference is that single-letter names can ONLY be used as local variables inside short methods. *The length of a name should correspond to the size of its scope* If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name.



Use Searchable Names

## 6.5 Dont Add Gratuitous Context

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

Likewise, say you invented a MailingAddress class in GSDs accounting module, and you named it GSDAccountAddress. Later, you need a mailing address for your customer contact

application. Do you use GSDAccountAddress? Does it sound like the right name? Ten of 17 characters are redundant or irrelevant.

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

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





## 7. Functions

Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. This is a much older truth. The art of programming is, and has always been, the art of language design.

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

To write functions/methods well(and, of course, reviewing other's methods), in addition to following the rules below, you need to let the author know that writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code. So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.

In the end, I wind up with functions that follow the rules I've laid down in this chapter. I don't write them that way to start. I don't think anyone could.

**Golden Rule of Functions**

As small as possible

## 7.1 Blocks and Indenting

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

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

For example:

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

return pageData.getHtml();
}
```



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



! The blocks within `if` statements, `else` statements, `while` statements, and so on should be one line long. The indent level of a function should not be greater than one or two.

## 7.2 Do One Thing

While everybody understands the following rule:

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

the problem with this statement is that it is hard to know what one thing is. Does Listing 13.1 do one thing? It's easy to make the case that it's doing three things:

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

Notice that the three steps of the function are one level of abstraction below the stated name of the function. We can describe the function by describing it in JavaDoc of this method:

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

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

Listing 7.1 has two levels of abstraction, as proved by our ability to shrink it down. But it would be very hard to meaningfully shrink Listing 13.1. We could extract the if statement into a function named `includeSetupsAndTeardownsIfTestPage`, but that simply restates the code without changing the level of abstraction.

### 7.2.1 One Level of Abstraction per Function

In order to make sure our functions are doing one thing, we need to make sure that the statements within our function are all at the same level of abstraction.

It is easy to see how a mixing of `getHtml()` (very high level abstraction) and `string.append("\n")` (very low level abstraction) is violating this rule.

Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. Worse, like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.

! Do NOT mix level of abstraction in a method

## 7.3 Reading Code from Top to Bottom: The Stepdown Rule

We want the code to read like a top-down narrative.<sup>5</sup> 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.

! **The Stepdown Rule** - We want the code to read like a top-down narrative.<sup>5</sup> 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.

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".

## 7.4 Switch Statements

Its hard to make a small switch statement(or an equivalent if-else chains). Its also hard to make a switch statement that does one thing. By their nature, switch statements always do N things. Unfortunately we cant 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**.

Consider Listing 7.4. It shows just one of the operations that might depend on the type of employee.

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



There are several problems with this function:

1. its large, and when new employee types are added, it will grow
2. it very clearly does more than one thing
3. it violates the Single Responsibility Principle because there is more than one reason for it to change
4. it violates the Open Closed Principle because it must change whenever new types are added
5. **WORST** - there are an unlimited number of other functions that will have the same structure. For example we could have isPayday(Employee e), or deliverPay(Employee e) or a host of others. All of which would have the same deleterious structure.

The solution to this problem (see Listing 7.4) is to bury the switch statement in the basement of an ABSTRACT FACTORY and never let anyone see it. The factory will use the switch statement to create appropriate instances of the derivatives of Employee, and the various functions, such as calculatePay, isPayday, and deliverPay, will be dispatched polymorphically through the Employee interface.

```

public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay (Money pay);
}
-----
public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r);
}
-----
public class EmployeeFactoryImpl implements EmployeeFactory {

    @Override
    public Employee makeEmployee(EmployeeRecord r) {
        switch (r.type) {
            case COMMISSIONED:
                return new CommissionedEmployee(r);
            case HOURLY:
                return new HourlyEmployee(r);
            case SALARIED:
                return new SalariedEmployee(r);
            default:
                // exception
        }
    }
}

```



! Bury the switch statement in the basement of an ABSTRACT FACTORY and never let anyone see it

The general rule for switch statements is that they can be tolerated if they

1. appear only once
2. are used to create polymorphic objects
3. are hidden behind an inheritance relationship so that the rest of the system cannot see them

! Reduce switch(or if-else chain) statements as much as possible

## 7.5 Function Arguments

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

Arguments are hard. They take a lot of conceptual power. Readers would have had to interpret it each time they saw it. When you are reading the story told by the module, includeSetupPage() is easier to understand than includeSetupPageInto(newPageContent). **The argument is at a different level of abstraction than the function name** and forces you to know a detail that isn't particularly important at that point.

**R** Arguments are hard. They take a lot of conceptual power.

Arguments are also hard from a testing point of view.

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.

- ! Reduce the number of arguments as much as possible.
- No output argument

### 7.5.1 Common Monadic Forms

There are two very common reasons to pass a single argument into a function:

1. Asking a question about that argument, as in `boolean fileExists(MyFile)`.
2. Operating and transforming it into something else and returning it. For example, transform a file name into content - `InputStream fileOpen(MyFile)`

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.

A somewhat less common, but still very useful form for a single argument function, is an *event*. In this form there is an input argument but no output argument. The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system.

- ! Use monadic argument appropriately

Try to avoid any monadic functions that don't follow these forms. Using an output argument instead of a return value for a transformation is confusing. If a function is going to transform its input argument, the transformation should appear as the return value.

- ! Put output argument as return value

### 7.5.2 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!

- ! Split the method into 2 sub-calls to enhance readability

### 7.5.3 Dyadic Functions

There are times, of course, where two arguments are appropriate. For example, it is perfectly reasonable to have `Point p = new Point(0,0);`. Cartesian points naturally take two arguments. However, the two arguments in this case are *ordered components of a single value*! Whereas `outputStream` and `name`, as in `writeField(outputStream, name)`, have neither a natural cohesion, nor a natural ordering.

Even obvious dyadic functions like `assertEquals(expected, actual)` are problematic. How many times have you put the `actual` where the `expected` should be? The two arguments have no natural ordering. The expected, actual ordering is a convention that requires practice to learn. This is one of the reasons I do not use junit for testing.

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 mechanisms 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.



Design natural-ordering or natural-cohesion bi-arguments

## 7.6 Argument Objects

When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.

Consider, for example, the difference between the two following declarations:

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

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

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

## 7.7 Argument Lists

Sometimes we want to pass a variable number of arguments into a function. Consider, for example:

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

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

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

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

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



## 7.8 Coherent method name and arguments

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.



## 7.9 Have No Side Effects

### 7.9.1 What is Side Effects

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

### 7.9.2 Example

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



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

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



Watch out for side effects in review - does the method do more than what method name implies?

### 7.9.3 Output Arguments

Output argument makes it hard to read. For example:

```
appendFooter(s);
```



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

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

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

! Anything that forces you to check the function signature is equivalent to a double-take which should be avoided

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*

## 7.10 Command Query Separation

Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion. Consider, for example, the following function:

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



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

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



What does it mean? It is hard to answer without reading the source code of the method (or its doc). The real solution is to separate the command from the query so that the ambiguity cannot occur:

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



## 7.11 Prefer Exceptions to Returning Error Codes

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

```

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

```



Note the first line in Listing 7.11. When you return an error code, you create the problem that the caller must *deal with the error immediately*.

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

```

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

```



Another advantage of using Exceptions is that returning error codes usually implies that there is some class or enum in which all the error codes are defined.

```

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

```



Classes like this are a dependency magnet; many other classes must import and use them. Thus, when the Error enum changes, all those other classes need to be recompiled and redeployed.<sup>11</sup> This puts a negative pressure on the Error class. Programmers don't want to add new errors because then they have to rebuild and redeploy everything. So they reuse old error codes instead of adding new ones.

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



Exception-handling, instead of Error Code, reduces coupling

## 7.12 Extract Try/Catch Blocks

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

Functions should do one thing. Error handling is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword try exists in a function, it should be the very first word in the function and that there should be nothing after the catch/finally blocks.



Put try-catch in a method and nothing else



## 8. Constructors

### 8.1 Constructors in general

A conventional distinction is made in Java between the "default constructor" and a "no-argument constructor":

**Definition 8.1.1 — Default constructor.** The default constructor is the constructor provided by the system in the absence of any constructor provided by the programmer. Once a programmer supplies any constructor whatsoever, the default constructor is no longer supplied.

**Definition 8.1.2 — No-args constructor.** A no-argument constructor, on the other hand, is a constructor provided by the programmer which takes no arguments. This distinction is necessary because the behavior of the two kinds of constructor are unrelated: a default constructor has a fixed behavior defined by Java, while the behavior of a no-argument constructor is defined by the application programmer.

! It is important that JavaDoc comments makes clear distinctions between the two types of constructors. Watch out for any discrepancies in author's code.

### 8.2 Do Not Pass 'this' out of a Constructor

Within a class, the `this` Java keyword refers to the native object, the current instance of the class. Within a constructor, you can use the keyword in 3 different ways:

- on the first line, you can call another constructor, using `this(...)`;
- as a qualifier when referencing fields, as in `this.name`;
- as a reference passed to a method of some other object, as in `blah.operation(this)`;

You can get into trouble with the last form. The problem is that, inside a constructor, the object is not yet fully constructed.

**R** An object is only fully constructed after its constructor completely returns, and not before

But when passed as a parameter to a method of some other object, the `this` reference should always refer to a fully-formed object. This problem occurs mainly with listeners. Here's an example which illustrates the point:

```
import java.util.Observable;
import java.util.Observer;

public final class EscapingThisReference {

    // A RadioStation is observed by the people listening to it.
    static final class RadioStation extends Observable {
        // elided
    }

    /**
     * A listener which waits until this object is fully-formed before it lets it be refer
     * Uses a private constructor to first build the object, and then configures the fully
     */
    static final class GoodListener implements Observer {

        /** Factory method. */
        static GoodListener buildListener(String personsName, RadioStation station) {
            //first call the private constructor
            GoodListener result = new GoodListener(personsName);
            //the 'result' object is now fully constructed, and can now be
            //passed safely to the outside world
            station.addObserver(result);
            return result;
        }

        @Override
        public void update(Observable station, Object data) {
            //..elided
        }

        private String personsName;

        /** Private constructor. */
        private GoodListener(String personsName) {
            this.personsName = personsName; //ok
        }
    }

    /**
     * A listener which incorrectly passes a 'this' reference to the outside world before
     */
    static final class BadListenerExplicit implements Observer {

        /** Ordinary constructor. */
        BadListenerExplicit(String personsName, RadioStation station) {
            this.personsName = personsName; //OK
        }
    }
}
```

```

        //DANGEROUS - the 'this' reference shouldn't be passed to the listener,
        //since the constructor hasn't yet completed; it doesn't matter if
        //this is the last statement in the constructor!
        station.addObserver(this);
    }

    @Override
    public void update(Observable station, Object data) {
        //..elided
    }

    private String personsName;
}
/**
* Another listener that passes out a 'this' reference to the outside world before co
* the 'this' reference is implicit, via the anonymous inner class.
*/
static final class BadListenerImplicit {

    /** Ordinary constructor. */
    BadListenerImplicit(String personsName, RadioStation station) {
        this.personsName = personsName; //OK
        //DANGEROUS
        station.addObserver(
            new Observer() {
                @Override
                public void update(Observable observable, Object data) {
                    doSomethingUseful();
                }
            });
    }

    private void doSomethingUseful() {
        //..elided
    }

    private String personsName;
}
}

```



Find and tell code authors to avoid passing **this** out of constructor

### 8.3 Avoid JavaBeans Style of Construction

Constructing a Model Object in many steps, first by calling a no-argument constructor, and then by calling a series of setXXX methods, is something to be avoided, if possible, because:

- it is more complex, since one call is replaced with many
- there is no simple way to ensure that all necessary setXXX methods are called
- there is no simple way to implement a class invariant

- it allows the object to be in intermediate states which are not immediately useful, and perhaps even invalid
- it is not compatible with the (very useful) immutable object pattern
- it does not follow the style of many design patterns, in which concrete classes usually have arguments passed to their constructor. This is a recurring theme in the Design Patterns book. Constructors usually act as a "back door" for data which is needed by an implementation, but which cannot appear in the signature of some higher-level abstraction (since it would pollute such an abstraction with data specific to an implementation)

Of course, JavaBeans follow this pattern of "no-argument constructor plus setXXX". However, JavaBeans were originally intended for a very narrow problem domain - manipulation of graphical components in an IDE. As a model for the construction of typical data-centric objects (Model Objects), JavaBeans seem completely inappropriate. One can even argue strongly that, for Model Objects, the JavaBeans style is an anti-pattern, and should be avoided unless absolutely necessary.

Reading JavaBeans Spec helps you completely understand why JavaBean is not a good idea in this context. It is important to understand Java Beans in terms of how Java is used today. The bean initiates itself as a GUI concept. Java today refers to "bean" in a completely different context defined by its popularities in non-GUI developments.

**Definition 8.3.1** What is a Bean (Historically)? A Java Bean is a reusable software component that can be manipulated visually in a builder tool.

Simply put, this is defined in the old Java Applet in which you click through to configure the applet before you hit run. The bean is used to record what you clicked and later spins up the program accordingly.

But today you do not hear Java Applet anymore, but you still hear about beans. Why? Because bean is usable to other popular contexts. One of them is JPA (or Persistent Storage)

This is very clear and simple. No mention is made here, or anywhere else in the specification, of Model Objects used to encapsulate database records. The whole idea of a Model Object is entirely absent from the Java Bean specification, and for good reason: the manipulation of graphical controls and the representation of database records (See 2.1.1 Beans v. Class Libraries in JavaBeans Spec) are unrelated issues.

The life cycle of a Java Bean, starting with its no-argument constructor, is derived mainly from the task of configuring a graphical control having some default initial state. For the intended domain, this is indeed a reasonable design. For Model Objects, however, the idea of configuration in this sense is entirely meaningless.

The role of a Model Object is to carry data. Since a no-argument constructor can only build an object that is initially empty, then it cannot contribute directly to this aim. It does not form a natural candidate for the design of a Model Object.

To put data into empty Java Beans, frameworks typically call various setXXX methods using reflection. There is nothing to prevent them from calling constructors in exactly the same way.

The widespread adoption of Java Beans as a supposedly appropriate design for Model Objects seems to be an error.



Unless absolutely necessary, do not set through objects.

## 8.4 Initializing fields to 0, false, or null is redundant

One of the most fundamental aspects of a programming language is how it initializes data. For Java, this is defined explicitly in the language specification (<https://docs.oracle.com/javase/8/specs/jls/se8/html/jls-4.html#jls-4.12.5>). For fields and array components, when items are created, they are automatically set to the following default values by the system:



- numbers: 0 or 0.0
- booleans: false
- object references: null

This means that explicitly setting fields to 0, false, or null (as the case may be) is unnecessary and redundant. Since this language feature was included in order to, in part, reduce repetitive coding, its a good idea to take full advantage of it. Insisting that fields should be explicitly initialized to 0, false, or null is an idiom which is likely inappropriate to the Java programming language

Furthermore, setting a field explicitly to 0, false, or null may even cause the same operation to be performed twice (depending on your compiler). For example:

```
public final class Quark {  
  
    public Quark(String aName, double aMass){  
        fName = aName;  
        fMass = aMass;  
    }  
  
    //PRIVATE  
  
    //WITHOUT redundant initialization to default values  
    //private String fName;  
    //private double fMass;  
  
    //WITH redundant initialization to default values  
    private String fName = null;  
    private double fMass = 0.0d;  
}
```

If the bytecode of the Quark class is examined, the duplicated operations become clear (here, Oracles javac compiler was used):

Title in the first language	Title in the second language
<pre> &gt;javap -c -classpath . Quark Compiled from Quark.java public final class Quark extends     java.lang.Object { public Quark(java.lang.String,     double); }  Method Quark(java.lang.String,     double) 0 aload_0 1 invokespecial #1 &lt;Method java.     lang.Object()&gt; 4 aload_0 5 aload_1 6 putfield #2 &lt;Field java.lang.     String fName&gt; 9 aload_0 10 dload_2 11 putfield #3 &lt;Field double     fMass&gt; 14 return </pre>	<pre> &gt;javap -c -classpath . Quark Compiled from Quark.java public final class Quark extends     java.lang.Object { public Quark(java.lang.String,     double); }  Method Quark(java.lang.String,     double) 0 aload_0 1 invokespecial #1 &lt;Method java.     lang.Object()&gt; 4 aload_0 5 aconst_null 6 putfield #2 &lt;Field java.lang.     String fName&gt; 9 aload_0 10 dconst_0 11 putfield #3 &lt;Field double     fMass&gt; 14 aload_0 15 aload_1 16 putfield #2 &lt;Field java.lang.     String fName&gt; 19 aload_0 20 dload_2 21 putfield #3 &lt;Field double     fMass&gt; 24 return </pre>

## 8.5 Constructors Shouldn't Start Threads

There are two problems with starting a thread in a constructor (or static initializer):

1. in a non-final class, it increases the danger of problems with subclasses
2. it opens the door for allowing the this reference to be passed to another object before the object is fully constructed

Theres nothing wrong with creating a thread object in a constructor or static initializer - just dont start it there.

## 9. Comments

### 9.1 Comments Do Not Make Up for Bad Code

One of the more common motivations for writing comments is bad code. We write a module and we know it is confusing and disorganized. We know it's a mess. So we say to ourselves, Ooh, I'd better comment that! No! You'd better clean it! 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.

### 9.2 Explain Yourself as Code

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. Which would you rather see? This:

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

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

It takes only a few seconds of thought to explain most of your intent in code. In many cases it's simply a matter of creating a function that says the same thing as the comment you want to write.



If there is a comment for a piece of logic, it might be a sign of review comments saying "this should be turned into a method"

## 9.3 Good Comments

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

### 9.3.1 Legal Comments

#### An Example

```
/*
 * Copyright <some author>
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 * implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */
```

### 9.3.2 Informative Comments

#### Regular Expression

Regex is hard to read. A comment tells what that regex does in an easy way. For example,

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
    "\\ d*:\\ d*:\\ d* \\ w*, \\ w* \\ d*, \\ d*"
);
```



In this case the comment lets us know that the regular expression is intended to match a time and date that were formatted with the `SimpleDateFormat.format` function using the specified format string. Still, it might have been better, and clearer, if this code had been moved to a special class that converted the formats of dates and times. Then the comment would likely have been superfluous.

### 9.3.3 Explanation of Intent

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

```
// This is our best attempt to get a race condition
// by creating large number of threads.
for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread = new
        WidgetBuilderThread(widgetBuilder, text , parent , failFlag );

    Thread thread = new Thread(widgetBuilderThread);
    thread.start ();
}
```



### 9.3.4 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 it's part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful.

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
    WikiPagePath ba = PathParser.parse("PageB.PageA");

    assertTrue (a.compareTo(a) == 0); // a == a
    assertTrue (a.compareTo(b) != 0); // a != b
    assertTrue (ab.compareTo(ab) == 0); // ab == ab
    assertTrue (a.compareTo(b) == -1); // a < b
    assertTrue (aa.compareTo(ab) == -1); // aa < ab
    assertTrue (ba.compareTo(bb) == -1); // ba < bb
    assertTrue (b.compareTo(a) == 1); // b > a
    assertTrue (ab.compareTo(aa) == 1); // ab > aa
    assertTrue (bb.compareTo(ba) == 1); // bb > ba
}
```



### 9.3.5 Warning of Consequences

Sometimes it is useful to warn other programmers about certain consequences. For example, you might want to put "Do not use Google API for this method because it cannot handle such such business case"

### 9.3.6 TODO Comments

TODO's are jobs that the programmer thinks should be done, but for some reason can't do at the moment.

Nowadays, most good IDEs provide special gestures and features to locate all the TODO comments, so it's not likely that they will get lost. Still, you don't want your code to be littered with TODOs. So scan through them regularly and eliminate the ones you can.

! When you see a relevant TODO during review process, see if this can be resolved in the same PR

### 9.3.7 Javadocs in Public APIs

There is nothing quite so helpful and satisfying as a well-described public API. If you are writing a public API, then you should certainly write good javadocs for it. But keep in mind the rest of the advice in this chapter. Javadocs can be just as misleading, nonlocal, and dishonest as any other kind of comment.

! During review process, check if implementation matches what Javadocs describes .

## 9.4 Bad Comments

### 9.4.1 Confusing Comments

When you review someone else's comments and pauses, this might be a sign of incomplete or insufficient comments. You should ask the code author to elaborate it more on that.

### 9.4.2 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");
        }
    }
}
```



Listing 9.4.2 shows a simple function with a header comment that is completely redundant. The comment probably takes longer to read than the code itself.

What purpose does this comment serve? Its 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.

#### Method comments

- ! should be more informative than the code
- Justify the code, i.e. provide intent or rationale
- must be easier to read than the code

Now consider the legion of useless and redundant javadocs in Listing 4-2 taken from Tomcat. These comments serve only to clutter and obscure the code. They serve no documentary purpose at all

```
public abstract class ContainerBase
implements Container, Lifecycle , Pipeline ,
MBeanRegistration, Serializable {
    /**
     * The processor delay for this component.
     */
    protected int backgroundProcessorDelay = -1;

    /**
     * The lifecycle event support for this component.
     */
    protected LifecycleSupport lifecycle = new
    LifecycleSupport(this);

    /**
     * The container event listeners for this Container.
     */
    protected ArrayList listeners = new ArrayList();

    /**
     * The Loader implementation with which this Container is
     * associated .
     */
    protected Loader loader = null;

    /**
     * The Logger implementation with which this Container is
     * associated .
     */
    protected Log logger = null;

    /**
     * Associated logger name.
     */
    protected String logName = null;
```



Avoid the comments that says the name of a variable in anothe way like the ones shown in Listing 9.4.2

In addition, those are "noisy" comments that you should really comment on during code review:

```

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

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

/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
    return dayOfMonth;
}

```



#### 9.4.3 Dont Use a Comment When You Can Use a Function or a Variable

During review process, if you see the author put a comment on top of a section of code, this is a sign of potentially saying something on that: see if the comment can be replaced by a set of more expressive code. For example, consider the following stretch of code:

```

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

```



This could be rephrased without the comment as

```

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

```



On seeing a comment, see if that comment can be replaced by a set more expressive code

#### 9.4.4 Position Markers

Example of position markers

```

// Actions //////////////////////////////////////

```





! There are rare times when it makes sense to gather certain functions together beneath a banner like this. But in general they are clutter that should be eliminated

### 9.4.5 Closing Brace Comments

#### Example of closing brace comments

```
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
```



Although this might make sense for long functions with deeply nested structures, it serves only to clutter the kind of small and encapsulated functions that we prefer. So if you find yourself wanting to mark your closing braces, try to shorten your functions instead.

! When you see a closing brace comment, this is a sign of suggesting shortening functions

### 9.4.6 Attributions and Bylines

#### Example of closing brace comments

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



Source code control systems are very good at remembering who added what, when. There is no need to pollute the code with little bylines.

### 9.4.7 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()));
```



Few practices are as odious as commenting-out code. Don't do this!

There was a time, back in the sixties, when commenting-out code might have been useful. But we've had good source code control systems for a very long time now. Those systems will remember the code for us. We don't have to comment it out any more. Just delete the code. We won't lose it. Promise.



Ask PR author to remove any commented out code, if any

### 9.4.8 HTML Comments

HTML in source code comments is an abomination, as you can tell by reading the code below. It makes the comments hard to read in the one place where they should be easy to read. The editor/IDE. If comments are going to be extracted by some tool (like Javadoc) to appear in a Web page, then it should be the responsibility of that tool, and not the programmer, to adorn the comments with appropriate HTML.

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



### 9.4.9 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. Consider, for example, the javadoc comment below. Aside from the fact that it is horribly redundant, it also offers information about

the default port. And yet the function has absolutely no control over what that default is. The comment is not describing the function, but some other, far distant part of the system. Of course there is no guarantee that this comment will be changed when the code containing the default is changed.

```
/**
 * Port on which fitnessse would run. Defaults to <b>8082</b>.
 *
 * @param fitnesssePort
 */
public void setFitnesssePort (int fitnesssePort ) {
    this.fitnesssePort = fitnesssePort ;
}
```



#### 9.4.10 Inobvious Connection

Lets consider this example:

```
/*
 * start with an array that is big enough to hold all the pixels
 * (plus filter bytes), and an extra 200 bytes for header info
 */
this.pngBytes = new byte[(((this.width + 1) * this.height * 3)
    + 200)];
```



What is filter byte? The code itself has nothing to do with filter byte.

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.





## 10. Formatting

### 10.1 Size of a File

File size should be no more than 500 lines of code **approximately**

### 10.2 The Newspaper Metaphor

A source file should be like a newspaper article, in which a headline tells you what the story is all about and as you do down you see more details.

A Java class name should be simple and explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not.

The topmost parts of the source file should provide the high-level concepts and algorithms. Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.

- ! Java Class name should be self-explanatory
- The class doc should provide high-level concepts and algorithms.

### 10.3 Vertical Density

The vertical density implies close association. So lines of code that are tightly related should appear vertically dense. Listing 10.3 is an example of how too much comments screws up vertical density:

```
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);  
    }  
  
    ...  
}
```



! Avoid doc on instance variables whenever you can

## 10.4 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.

! If one function calls another, they should be vertically close, and the caller should be above the callee.

## 10.5 Conceptual Affinity

Anything conceptually related should be vertically placed close to each other. For example, a group of functions perform a similar operation (share a common naming scheme and perform variations of the same basic task)

! Place conceptually related pieces close to each other vertically

## 10.6 Vertical Ordering

In general we want function call dependencies to point in the downward direction. we expect the most important concepts to come first, and we expect them to be expressed with the least amount of polluting detail. We expect the low-level details to come last. This allows us to skim source files, getting the gist from the first few functions, without having to immerse ourselves in the details.

! Put the most important concept about a class on top.

# 11. Objects and Data Structures

## 11.1 Data Abstraction

Data Abstraction represents more than just a data structure. The methods enforce an access policy. For example:

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian (double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar (double r, double theta );  
}
```

You can read the individual coordinates independently, but **you must set the coordinates together as an atomic operation.**

## 11.2 Data/Object Anti-Symmetry(VERY IMPORTANT)

We know Java is an *Object Oriented Language* as opposed to *Procedure Language*, such as C. Let's start with 3 data structures and 1 procedure used in procedure language. Listing 11.2 show the difference between objects and data structures.

```

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.width;
        } else if (shape instanceof
            Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.
            radius;
        }
        throw new NoSuchShapeException
        ();
    }
}

```

Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions.

The Geometry class operates on the three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the Geometry class.

Consider what would happen if a perimeter() function were added to Geometry, you notice the following:

- The shape classes would be unaffected. Any other classes that depended upon the shapes would also be unaffected.
- On the other hand, if I add a new shape, I must change all the functions in Geometry to deal with it.

Now hold the points above in your memory for just a moment and let's consider the object-oriented solution in Listing 11.2

```

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 {

    public static final double PI = 3.141592653589793;

    private Point center;
    private double radius;

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

```

When `perimeter()` function is added to `Geometry`, you notice the following:

- The shape classes would be affected
- On the other hand, if I add a new shape, `Geometry` is not changed

*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.*

So, the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO.

In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate.

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.

! When we want to add new data types rather than new functions, Objects and OO are better. However if we want to add new functions, procedure code and data structure will be more appropriate. 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.

### 11.3 The Law of Demeter

A module should not know about the innards of the objects it manipulates. An object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.

#### Law of Demeter

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. e.g. `final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();` violates this rule

#### 11.4 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. **Somewhat more common in Java is the bean form.**

## 12. Error-handling

### 12.1 Use Unchecked Exceptions

Although checked exception brings benefits, it comes with prices. The price of checked exceptions is an Open/Closed Principle violation. If you throw a checked exception from a method in your code and the `catch` is three levels above, you must declare that exception in the signature of each method between you and the `catch`. This means that a change at a low level of the software can force signature changes on many higher levels. The changed modules must be rebuilt and redeployed, even though nothing they care about changed.

### 12.2 Define the Normal Flow

`try-catch` is great, but sometimes it looks awkward if the catch block does some "exceptional" processing other than logging-and-throwing-exception. For example:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(
        employee.getID());
    m_total += expenses.getTotal ();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```



What's awkward about this is that it wraps a flow of normal logic inside the catch block. The exception clutters the logic. Wouldn't it be better if we didn't have to deal with the special case? If we didn't, our code would look much simpler. It would look like this:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.  
    getID());  
m_total += expenses.getTotal () ;
```



Can we make the code that simple? It turns out that we can. We can change the ExpenseReportDAO so that it always returns a MealExpense object. If there are no meal expenses, it returns a MealExpense object that returns the per diem as its total:

```
public class PerDiemMealExpenses implements MealExpenses {  
  
    public int getTotal () {  
        // return the per diem default  
    }  
}
```



This is called the **SPECIAL CASE PATTERN**. You create a class or configure an object so that it handles a special case for you. When you do, the client code doesn't have to deal with exceptional behavior. That behavior is encapsulated in the special case object.



Create a class or configure an object so that it handles a special case for you

### 12.3 Dont Return Null

When we return null, we are essentially creating work for ourselves and foisting problems upon our callers. All it takes is one missing null check to send an application spinning out of control.



If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object, such as Optional instead.



## 13. Boundaries

Interesting things happen at boundaries(e.g. between our code and 3rd-party libraries). Change(e.g. from a 3rd-party library) 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.

Code at the boundaries needs clear separation and tests that define expectations. We should avoid letting too much of our code know about the third-party particulars. It's better to depend on something you control than on something you don't control, lest it end up controlling you.

We manage third-party boundaries by having very few places in the code that refer to them. We may wrap them in our own object, or we may use an ADAPTER to convert from our perfect interface to the provided interface. Either way our code speaks to us better, promotes internally consistent usage across the boundary, and has fewer maintenance points when the third-party code changes. The sections below discusses how to achieve this in detail and, of course, guides your review process.

### 13.1 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.

For example, `java.util.Map` is a very useful object, but what if we would like a map that just supports inserts and retrievals, but not delete? When you pass this map around system, you cannot guarantee that other maintainers would not misused it by calling a delete method.

A cleaner way to use Map, for example, might look like the following:

```
public class Sensors {  
  
    private Map sensors = new HashMap();  
    public Sensor getById(String id) {  
        return (Sensor) sensors.get(id);  
    }  
    // snip  
}
```



**The interface at the boundary (Map) is hidden.** It is able to evolve with very little impact on the rest of the application. The misuse of calling delete is not possible because you never expose that method in this class.

This 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. Therefore this class can enforce design and business rules.



Try not to pass Maps (or any other interface at a boundary) around your system. If you use a boundary interface like Map, keep it inside the class, or close family of classes, where it is used. Avoid returning it from, or accepting it as an argument to, public APIs.

## 13.2 Exploring and Learning Boundaries

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. Here is why: when we code 3rd-party libraries into our production system, We would not be surprised to find ourselves bogged down in long debugging sessions trying to figure out whether the bugs we are experiencing are in our code or theirs.

Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code. Such tests are called *learning tests*(or boundary tests).



Write learning tests to make sure 3rd-parity API does the right things for us.

In learning tests we call the third-party API, as we expect to use it in our application. We're essentially doing controlled experiments that check our understanding of that API. The tests focus on what we want out of the API.

To summarize the benefits of learning tests:

- Writing learning tests is a very good way to learn a 3rd-party API.
- When there are new releases of the third-party package, we run the learning tests to see whether there are behavioral differences.
- Learning tests verify that the third-party packages we are using work the way we expect them to.





## 14. Unit Tests

Test code is just as important as production code.

We have barely scratched the surface of tests. Tests are as important to the health of a project as the production code is. Perhaps they are even more important, because tests preserve and enhance the flexibility, **maintainability**, and reusability of the production code. So keep your tests constantly clean. Work to make them expressive and succinct. Invent testing APIs that act as domain-specific language that helps you write the tests. If you let the tests rot, then your code will rot too. Keep your tests clean.

### 14.1 The Three Laws of TDD(Test Driven Development)

#### The Three Laws of TDD

1. You may not write production code until you have written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
3. You may not write more production code than is sufficient to pass the currently failing test.

### 14.2 Clean Tests

What makes a clean test? Three things:

1. Readability
2. Readability
3. Readability

What makes tests readable? The same thing that makes all code readable: clarity, simplicity, and density of expression. To be more specific:

- No code duplication

- Less code details that interfere with the expressions of test
- Clear separation of setup, run, and test results (with the help of domain-specific testing language to simplify and shorten tests)

### 14.3 F.I.R.S.T.

Clean tests follow five rules.

**Fast** Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot.

**Independent** Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, then the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

**Repeatable** Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available.

**Self-Validating** The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

**Timely** The tests need to be written in a timely fashion. 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. You may decide that some production code is too hard to test. You may not design the production code to be testable.





## 15. Classes

### 15.1 Classes Should Be Small!

The fact that classes should be small is not an issue to be discussed, the question, during review, process is *how small should it be?* With functions we measured size by counting physical lines. With classes we use a different measure. We count *responsibilities*.

The name of a class should describe what responsibilities it fulfills. If we cannot derive a concise name for a class, then it's likely too large and has too many responsibilities.

We should also be able to write a brief description of the class in about 25 words, without using the words "if", "and", "or", or "but".

! A class's single responsibility should be shown by a self-descriptive name and a short class description.

#### 15.1.1 The Single Responsibility Principle

**Definition 15.1.1 — Single Responsibility Principle (SRP).** A class or module should have one and only one, reason to change.

Trying to identify responsibilities (reasons to change) often helps us recognize and create better abstractions in our code. During code review, you could ask author to break a class into separate ones by methods

! Review to make sure classes satisfies SRP and ask the author to break classes into multiples if not.

#### Why Do We Often Violate SRP?

Getting software to work and making software clean are two very different activities. Most of us have limited room in our heads, so we focus on getting our code to work more than organization and cleanliness. This is wholly appropriate. Maintaining a separation of concerns is just as important in our programming activities as it is in our programs

The problem is that too many of us think that we are done once the program works. We fail to

switch to the other concern of organization and cleanliness.

**This is an important reason we do code review - Once we review to make sure code works, we would want to do a second review to make sure code is also clean**

- ! There should be at least 2 review process:
1. make sure code works
  2. make sure code is clean

### 15.1.2 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.

In general it is neither advisable nor possible to create such maximally cohesive classes; on the other hand, we would like cohesion to be high. When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole.

Since a class should have small number of instance variables, the strategy of keeping functions small and keeping parameter lists short can sometimes lead to a proliferation of instance variables that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to get out of the larger class. You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive.

- ! Keep the number of instance variables small and breaks a class apart when there are many

#### Application of Cohesion - Keeping Classes Small

Consider a large function with many variables declared within it. Lets say you want to extract one small part of that function into a separate function. However, the code you want to extract uses four of the variables declared in the function. Must you pass all four of those variables into the new function as arguments?

Not at all! If we promoted those four variables to instance variables of the class, then we could extract the code without passing any variables at all. It would be easy to break the function up into small pieces.

Unfortunately, this also means that our classes lose cohesion because they accumulate more and more instance variables that exist solely to allow a few functions to share them. But wait! If there are a few functions that want to share certain variables, doesn't that make them a class in their own right? Of course it does. When classes lose cohesion, split them!

So breaking a large function into many smaller functions often gives us the opportunity to split several smaller classes out as well. This gives our program a much better organization and a more transparent structure.

- ! During review process, see if splitting methods could lead to smaller classes.

## 15.2 Organizing for Change



## 16. Systems

### 16.1 Scaling Up

The hard part of enterprise applications is all about scaling up while making the code clean. A system **architecture** must allow scalability and extensibilities too. How do we make that happen?

Software systems are unique compared to physical systems. Their architectures can grow incrementally, **if** we maintain the proper separation of concerns.


Certainly, *separation of concerns* is the most important topic we shall discuss to make a clean, extensible, and scalable system below.

! Reviewing code of an initial system needs extra effort on evaluating Separation of Concerns.

#### 16.1.1 Approaches for Separation of Concerns

- Dependency Injection
- Factory Pattern





## 17. Concurrency

### 17.1 Concurrency Defense Principles

While reviewing code, here are series of principles that the code under review should follow in order to defend a system from the problem of concurrent code.

#### 17.1.1 Single Responsibility Principle

In order for concurrency-related code to obey SRP, here are a few things to consider:

- Concurrency-related code has its own life cycle of development, change, and tuning.
- Concurrency-related code has its own challenges, which are different from and often more difficult than nonconcurrency-related code.
- The number of ways in which miswritten concurrency-based code can fail makes it challenging enough without the added burden of surrounding application code.



Keep your concurrency-related code separate from other code

#### 17.1.2 Limit the Scope of Data

As we saw, two threads modifying the same field of a shared object can interfere with each other, causing unexpected behavior. One solution is to use the synchronized keyword to protect a critical section in the code that uses the shared object. It is important to restrict the number of such critical sections. The more places shared data can get updated, the more likely:

- You will forget to protect one or more of those places effectively breaking all code that modifies that shared data.
- There will be duplication of effort required to make sure everything is effectively guarded (violation of DRY).
- It will be difficult to determine the source of failures.



Take data encapsulation to heart; severely limit the access of any data that may be shared.



### 17.1.3 Use Copies of Data

A good way to avoid shared data is to avoid sharing the data in the first place. In some situations it is possible to copy objects and treat them as read-only. In other cases it might be possible to copy objects, collect results from multiple threads in these copies and then merge the results in a single thread.

If there is an easy way to avoid sharing objects, the resulting code will be far less likely to cause problems. You might be concerned about the cost of all the extra object creation. It is worth experimenting to find out if this is in fact a problem. However, if using copies of objects allows the code to avoid synchronizing, the savings in avoiding the intrinsic lock will likely make up for the additional creation and garbage collection overhead.

! Try make it possible to have each thread manipulate just "copies" of data

### 17.1.4 Threads Should Be as Independent as Possible

Consider writing your threaded code such that each thread exists in its own world, sharing no data with any other thread. Each thread processes one client request, with all of its required data coming from an unshared source and stored as local variables. This makes each of those threads behave as if it were the only thread in the world and there were no synchronization requirements.

! Attempt to partition data into independent subsets than can be operated on by independent threads, possibly in different processors.

## 17.2 Know Library

Inexperience developer are not aware of all concurrency libraries in Java; it is reviewer's responsibility to let them know and they should also study it. For example, if you see a class with a `HashMap` plus a `Lock`, you should tell them to use `ConcurrentHashMap` instead.

! Ask people working on concurrency code to review the classes available to them such as `java.util.concurrent`, `java.util.concurrent.atomic`, and `java.util.concurrent.locks`.

## 17.3 Review for Execution Models

We you are reviewing concurrency-related code, your first task is to identify the execution model that's being used for implementation:

- **Producer-Consumer** One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it. The queue between the producers and consumers is a *bound resource* (Resources of a fixed size or number used in a concurrent environment. Examples include database connections and fixed-size read/write buffers.). This means producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume. Coordination between the producers and consumers via the queue involves producers and consumers signaling each other. The producers write to the queue and signal that the queue is no longer empty. Consumers read from the queue and signal that the queue is no longer full. Both potentially wait to be notified when they can continue.
- **Readers-Writers** When you have a shared resource that primarily serves as a source of information for readers, but which is occasionally updated by writers, throughput is an issue. Emphasizing throughput can cause starvation and the accumulation of stale information. Allowing updates can impact throughput. Coordinating readers so they do not read something

a writer is updating and vice versa is a tough balancing act. Writers tend to block many readers for a long period of time, thus causing throughput issues. The challenge is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput and avoiding starvation. A simple strategy makes writers wait until there are no readers before allowing the writer to perform an update. If there are continuous readers, however, the writers will be starved. On the other hand, if there are frequent writers and they are given priority, throughput will suffer. Finding that balance and avoiding concurrent update issues is what the problem addresses.

- **Dining Philosophers** Imagine a number of philosophers sitting around a circular table. A fork is placed to the left of each philosopher. There is a big bowl of spaghetti in the center of the table. The philosophers spend their time thinking unless they get hungry. Once hungry, they pick up the forks on either side of them and eat. A philosopher cannot eat unless he is holding two forks. If the philosopher to his right or left is already using one of the forks he needs, he must wait until that philosopher finishes eating and puts the forks back down. Once a philosopher eats, he puts both his forks back down on the table and waits until he is hungry again. Replace philosophers with threads and forks with resources and this problem is similar to many enterprise applications in which processes compete for resources. Unless carefully designed, systems that compete in this way can experience deadlock, livelock, throughput, and efficiency degradation.

Most concurrent problems you will likely encounter will be some variation of these three problems. Study these algorithms and solutions of them on your own so that when you come across code addressing concurrent problems, you'll be more prepared to review them.



Learn these basic algorithms and understand their solutions.

## 17.4 Keep Synchronized Sections Small

The `synchronized` keyword introduces a lock. All sections of code guarded by the same lock are guaranteed to have only one thread executing through them at any given time. Locks are expensive because they create delays and add overhead. So we don't want to litter our code with `synchronized` statements. On the other hand, critical sections must be guarded. So we want to design our code with as few critical sections as possible. Some naive programmers try to achieve this by making their critical sections very large. However, extending synchronization beyond the minimal critical section increases contention and degrades performance.



Keep synchronized sections as small as possible.





## 18. What is Good-Enough?

Code reviewers should assume that **good code is not about code itself. It's about software users.**

### 18.1 Keep Your Users in the Trade-Off

Normally you're writing software for other people. Often you'll remember to get requirements from them.<sup>2</sup> But how often do you ask them how good they want their software to be? Sometimes there'll be no choice. If you're working on pacemakers, the space shuttle, or a low-level library that will be widely disseminated, the requirements will be more stringent and your options more limited. However, if you're working on a brand new product, you'll have different constraints. The marketing people will have promises to keep, the eventual end users may have made plans based on a delivery schedule, and your company will certainly have cash-flow constraints. It would be unprofessional to ignore these users' requirements simply to add new features to the program, or to polish up the code just one more time. We're not advocating panic: it is equally unprofessional to promise impossible time scales and to cut basic engineering corners to meet a deadline.

The scope and quality of the system produced should be specified as part of that system's requirements.

#### ! Make Quality a Requirements Issue

Often you'll be in situations where trade-offs are involved. Surprisingly, many users would rather use software with some rough edges today than wait a year for the multimedia version. Many IT departments with tight budgets would agree. Great software today is often preferable to perfect software tomorrow. If you give your users something to play with early, their feedback will often lead you to a better eventual solution.

#### ! Make sure PR size is not too big. Break big ones to make sure small and good pieces go in so that users can play early





# Part Two - Reviewing Architecture

<b>19</b>	<b>What is Design and Architecture? . . . .</b>	<b>77</b>
19.1	What is a Good Architecture?	
<b>20</b>	<b>Text Chapter . . . . .</b>	<b>79</b>
20.1	Paragraphs of Text	
20.2	Citation	
20.3	Lists	
<b>21</b>	<b>In-text Elements . . . . .</b>	<b>81</b>
21.1	Theorems	
21.2	Definitions	
21.3	Notations	
21.4	Remarks	
21.5	Corollaries	
21.6	Propositions	
21.7	Examples	
21.8	Exercises	
21.9	Problems	
21.10	Vocabulary	





## 19. What is Design and Architecture?

### 19.1 What is a Good Architecture?

- ! The goal of software architecture is to minimize the human resources required to build and maintain the required system.

The measure of design quality is simply the measure of the effort required to meet the needs of the customer. If that effort is low, and stays low throughout the lifetime of the system, the design is good. If that effort grows with each new release, the design is bad.

Why keeping effort low is important? When the stakeholders change their minds about a feature, that change should be simple and easy to make. The difficulty in making such a change should be proportional only to the scope of the change, and not to the shape of the change.

It is this difference between scope and shape that often drives the growth in software development costs. It is the reason that costs grow out of proportion to the size of the requested changes. It is the reason that the first year of development is much cheaper than the second, and the second year is much cheaper than the third.





## 20. Text Chapter

### 20.1 Paragraphs of Text

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim.



Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetur.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

## 20.2 Citation

This statement requires citation [1]; this one is more specific [2, page 162].

## 20.3 Lists

Lists are useful to present information in a concise and/or ordered way<sup>1</sup>.

### 20.3.1 Numbered List

1. The first item
2. The second item
3. The third item

### 20.3.2 Bullet Points

- The first item
- The second item
- The third item

### 20.3.3 Descriptions and Definitions

**Name** Description

**Word** Definition

**Comment** Elaboration

---

<sup>1</sup>Footnote example...



## 21. In-text Elements

### 21.1 Theorems

This is an example of theorems.

#### 21.1.1 Several equations

This is a theorem consisting of several equations.

**Theorem 21.1.1 — Name of the theorem.** In  $E = \mathbb{R}^n$  all norms are equivalent. It has the properties:

$$||\mathbf{x}| - |\mathbf{y}|| \leq |\mathbf{x} - \mathbf{y}| \quad (21.1)$$

$$||\sum_{i=1}^n \mathbf{x}_i|| \leq \sum_{i=1}^n ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \quad (21.2)$$

#### 21.1.2 Single Line

This is a theorem consisting of just one line.

**Theorem 21.1.2** A set  $\mathcal{D}(G)$  is dense in  $L^2(G)$ ,  $|\cdot|_0$ .

### 21.2 Definitions

This is an example of a definition. A definition could be mathematical or it could define a concept.

**Definition 21.2.1 — Definition name.** Given a vector space  $E$ , a norm on  $E$  is an application, denoted  $||\cdot||$ ,  $E$  in  $\mathbb{R}^+ = [0, +\infty[$  such that:

$$||\mathbf{x}|| = 0 \Rightarrow \mathbf{x} = \mathbf{0} \quad (21.3)$$

$$||\lambda \mathbf{x}|| = |\lambda| \cdot ||\mathbf{x}|| \quad (21.4)$$

$$||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}|| \quad (21.5)$$

### 21.3 Notations

**Notation 21.1.** Given an open subset  $G$  of  $\mathbb{R}^n$ , the set of functions  $\varphi$  are:

1. Bounded support  $G$ ;
2. Infinitely differentiable;

a vector space is denoted by  $\mathcal{D}(G)$ .

### 21.4 Remarks

This is an example of a remark.



The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field  $\mathbb{K} = \mathbb{R}$ , however, established properties are easily extended to  $\mathbb{K} = \mathbb{C}$ .

### 21.5 Corollaries

This is an example of a corollary.

**Corollary 21.5.1 — Corollary name.** The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field  $\mathbb{K} = \mathbb{R}$ , however, established properties are easily extended to  $\mathbb{K} = \mathbb{C}$ .

### 21.6 Propositions

This is an example of propositions.

#### 21.6.1 Several equations

**Proposition 21.6.1 — Proposition name.** It has the properties:

$$||\mathbf{x}| - |\mathbf{y}|| \leq \|\mathbf{x} - \mathbf{y}\| \quad (21.6)$$

$$\left\| \sum_{i=1}^n \mathbf{x}_i \right\| \leq \sum_{i=1}^n \|\mathbf{x}_i\| \quad \text{where } n \text{ is a finite integer} \quad (21.7)$$

#### 21.6.2 Single Line

**Proposition 21.6.2** Let  $f, g \in L^2(G)$ ; if  $\forall \varphi \in \mathcal{D}(G)$ ,  $(f, \varphi)_0 = (g, \varphi)_0$  then  $f = g$ .

### 21.7 Examples

This is an example of examples.

#### 21.7.1 Equation and Text

■ **Example 21.1** Let  $G = \{x \in \mathbb{R}^2 : |x| < 3\}$  and denoted by:  $x^0 = (1, 1)$ ; consider the function:

$$f(x) = \begin{cases} e^{|x|} & \text{si } |x - x^0| \leq 1/2 \\ 0 & \text{si } |x - x^0| > 1/2 \end{cases} \quad (21.8)$$

The function  $f$  has bounded support, we can take  $A = \{x \in \mathbb{R}^2 : |x - x^0| \leq 1/2 + \varepsilon\}$  for all  $\varepsilon \in ]0; 5/2 - \sqrt{2}[$ . ■

### 21.7.2 Paragraph of Text

■ **Example 21.2 — Example name.** Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. ■

## 21.8 Exercises

This is an example of an exercise.

**Exercise 21.1** This is a good place to ask a question to test learning progress or further cement ideas into students' minds. ■

## 21.9 Problems

**Problem 21.1** What is the average airspeed velocity of an unladen swallow?

## 21.10 Vocabulary

Define a word to improve a students' vocabulary.

**Vocabulary 21.1 — Word.** Definition of word.





# Part Two

<b>22</b>	<b>Presenting Information</b> .....	<b>87</b>
22.1	Table	
22.2	Figure	
	<b>Bibliography</b> .....	<b>89</b>
	Articles	
	Books	
	<b>Index</b> .....	<b>91</b>



# 22. Presenting Information

## 22.1 Table

Treatments	Response 1	Response 2
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

Table 22.1: Table caption

Referencing Table 22.1 in-text automatically.

## 22.2 Figure

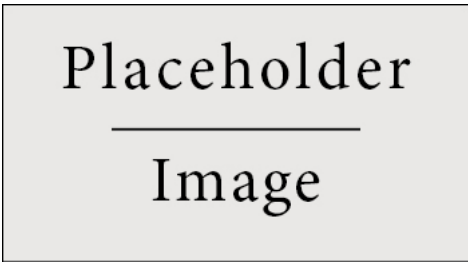


Figure 22.1: Figure caption

Referencing Figure 22.1 in-text automatically.







## Bibliography

### Articles

- [1] James Smith. “Article title”. In: 14.6 (Mar. 2013), pages 1–8 (cited on page 80).

### Books

- [2] John Smith. *Book title*. 1st edition. Volume 3. 2. City: Publisher, Jan. 2012, pages 123–200 (cited on page 80).



# Index

## B

### Boundaries

- Exploring and Learning Boundaries... 62
- Using Third-Party Code ..... 61

## C

Citation ..... 80

### Classes

- Classes Should Be Small ..... 65
- Organizing for Change ..... 66

### Comments

- Bad Comments ..... 46
- Comments Do Not Make Up for Bad Code  
43
- Explain Yourself in Code ..... 43
- Good Comments ..... 44

### Concurrency

- Concurrency Defense Principles ..... 69
- Keep Synchronized Sections Small ... 71
- Know Library ..... 70
- Review for Execution Models ..... 70

### Constructors

- Avoid JavaBeans Style of Construction 39
- Constructors Shouldn't Start Threads . 42
- default constructor ..... 37
- Do Not Pass 'this' out of a Constructor 37
- Initializing fields to 0, false, or null is re-  
dundant ..... 40

no-argument constructor ..... 37

Corollaries ..... 82

## D

Definitions ..... 81

## E

### Error-handling

- Define the Normal Flow ..... 59
- Dont Return Null ..... 60
- Use Unchecked Exceptions ..... 59

Examples ..... 82

Equation and Text ..... 82

Paragraph of Text ..... 83

Exercises ..... 83

## F

Figure ..... 87

### Fomratting

Conceptual Affinity ..... 54

### Formatting

Dependent Functions ..... 54

Size of a File ..... 53

The Newspaper Metaphor ..... 53

Vertical Density ..... 53

Vertical Ordering ..... 54

### Functions

Blocks and Indenting ..... 26

Argument Lists .....	31
Argument Objects .....	31
Coherent method name and arguments	31
Command Query Separation .....	33
Do One Thing .....	26
Extract Try/Catch Blocks .....	35
Function Arguments .....	29
Have No Side Effects .....	32
Reading Code from Top to Bottom: The Stepdown Rule .....	27
Switch Statements .....	28
Functions Prefer Exceptions to Returning Er- ror Codes .....	33

## L

Lists .....	80
Bullet Points .....	80
Descriptions and Definitions .....	80
Numbered List .....	80

## N

Naming	
Avoid Disinformation .....	22
Dont Add Gratuitous Context .....	23
Make Meaningful Distinctions .....	23
Use Intention-Revealing Names .....	21
Use Searchable Names .....	23
Use Speakable/Pronouncable Names ..	23
Notations .....	82

## O

Objects and Data Structures	
Data Abstraction .....	55
Data Transfer Objects .....	58
Data/Object Anti-Symmetry .....	55
The Law of Demeter .....	57

## P

Paragraphs of Text .....	79
Problems .....	83
Propositions .....	82
Several Equations .....	82
Single Line .....	82

## R

Remarks .....	82
---------------	----

## S

Systems	
Scaling Up .....	67

## T

Table .....	87
Theorems .....	81
Several Equations .....	81
Single Line .....	81

## U

Unit Tests	
Clean Tests .....	63
F.I.R.S.T. ....	64
The Three Laws of TDD .....	63

## V

Vocabulary .....	83
------------------	----