**Summary of Chapter 3 – 4**

**Chapter 3 Functions**

**Small!**

* First rule of functions is that they should be small
* They should be smaller than that
* Very small functions are better
* Functions should hardly ever be 20 lines long
* Functions should not be 100 lines long
* Lines should not be 150 characters long
* should usually be shorter than Listing 3-2
* Listing 3-2 should really be shortened to Listing 3-3

**Blocks and Indenting**

* Implies that the blocks within if statements, else statements, while statements, and so on should be one line long
* 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
* Functions should not be large enough to hold nested structures
* The indent level of a function should not be greater than one or two
* Makes the functions easier to read and understand

**Do One Thing**

* TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.
* If you can extract another function from it with a name that is not merely a restatement of its implementation

**Sections within Functions**

* Function is divided into sections such as declarations, initializations, and sieve.

**One Level of Abstraction per Function**

* 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
* Very high level of abstraction, such as getHtml();
* Intermediate level of abstraction, such as: String pagePathName = PathParser.render(pagePath);

**Switch Statements**

* It’s hard to make a small switch statement
* Because switch cases is definitely a large function.
* It’s also hard to make a switch statement that does one thing
* Make sure that each switch statement is buried in a low-level class and is never repeated.

**Use Descriptive Names**

* A long descriptive name is better than a short enigmatic name.
* Descriptive names will clarify the design of the module in your mind and help you to improve it.
* Be consistent in your names
* Use the same phrases, nouns, and verbs in the function names you choose for your modules.
* For example, the names includeSetupAndTeardownPages, includeSetupPages, includeSuiteSetupPage, and includeSetupPage.

**Function Arguments**

* **Ideal numbers of arguments for a functions**
* Zero (niladic)
* One (monadic)
* Two (dyadic)
* Three (triadic)
* More than three (polyadic) – requires very justification and then shouldn’t be used anyway

**Common Monadic Forms**

* Event is a less common but a very useful form for a single argument function
* Program is meant to interpret the function call as an event and use the argument to alter the state of the system
* For example, void passwordAttemptFailedNtimes(int attempts)
* Should be very clear to the reader that this is an event.

**Flag Arguments**

* Passing a boolean into a function is a truly terrible practice.

**Dyadic Functions**

* function with two arguments is harder to understand than a monadic function
* For example, writeField(name) is easier to understand than writeField(output-Stream, name).
* The meaning of both is clear, the first glides past the eye, easily depositing its meaning.
* The second requires a short pause until we learn to ignore the first parameter.

**Triads**

* Issues of ordering, pausing, and ignoring are more than doubled

-For example, consider the common overload of assertEquals that takes three arguments: assertEquals(message, expected, actual).

* a triad that is not quite so insidious: assertEquals(1.0, amount, .001)
* requires a double-take, it’s one that’s worth taking.

**Argument Objects**

* 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.

-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.

**Argument Lists**

* for example, String.format("%s worked %.2f hours.", name, hours);
* If all variable arguments are treated the same way, as in the preceding example, then they are the same as a single List argument.

**Verbs and Ketwords**

* 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
* For example, assertEquals might be better written as assertExpectedEqualsActual(expected, actual).

**Chapter 4 Comments**

* The proper use of comments is to compensate for our failure to express ourself in code.

**Comments Do Not Make Up for Bad Code**

* Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments

**Explain Yourself in Code**

* Many programmers have taken this to mean that code is seldom, if ever, a good means for explanation

**Goode Comments**

* good comment is the comment you found a way not to write

**Legal Comments**

* Sometimes our corporate coding standards force us to write certain comments for legal reasons.

-For example, copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file.

**Informative Comments**

* It is sometimes useful to provide basic information with a comment.

**Explanation of Intent**

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

**Clarification**

* It is preferable to find a means to make that argument or return value plain in its own right; but, when it is part of the standard library, or when it is used in a function, it is preferable to find a way to make it clear. If you can't change the code, a helpful explanatory comment can be helpful.

**Warning of Consequences**

* It is very useful to warn some programmers about a certain consequences.

**So For example:**

public static SimpleDateFormat makeStandardHttpDateFormat() { //SimpleDateFormat is not thread safe, //so we need to create each instance independently. SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z"); df.setTimeZone(TimeZone.getTimeZone("GMT")); return df; }

The comment that is given here is reasonable.

**ToDo Comments**

* It explains why the function has a degenerate implementation and what that function’s future should be.

**Amplification**

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

**Bad Comments**

* They are crutches or excuses for poor code or justifications for insufficient decisions.

**Journal Comments**

* A comment from the start of the module every time the programmer edit it.

**Noise Comments**

* Restate the obvious and provide no new information.

/\*\* \* Default constructor. \*/

protected AnnualDateRule() { }

**Don’t Use a Comment When You Can Use a Function or a Variable**

// does the module from the global list 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))

The author of the original code may have written the comment first (unlikely) and then written the code to fulfill the comment. However, the author should then have refactored the code, so that the comment could be removed.

**HTML Comments**

* source code comments is an abomination

**Too Much Information**

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

**Inobvious Connection**

* **The comment and the code should be obvious.**

**Function Headers**

* A well-chosen name for a small function that does one thing is usually better than a comment header.

**Javadocs in Nonpublic Code**

* As useful as javadocs are for public APIs, they are anathema to code that is not intended for public consumption.