**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.