Functions

```
public static String testableHtml(PageData pageData, boolean includeSuiteSetup) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup = PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE_SETUP_NAME, wik
            if (suiteSetup != null) {
                WikiPagePath pagePath = suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .").append(pagePathName).append("\n");
            }
        }
        WikiPage setup = PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath = wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .").append(setupPathName).append("\n");
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
        WikiPage teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath tearDownPath = wikiPage.getPageCrawler().getFullPath(teardown);
            String tearDownPathName = PathParser.render(tearDownPath);
            buffer.append("\n").append("!include -teardown .").append(tearDownPathName).append("\n");
        }
        if (includeSuiteSetup) {
            WikiPage suiteTeardown = PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE_TEARDOWN_NAME
            if (suiteTeardown != null) {
                WikiPagePath pagePath = suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -teardown .").append(pagePathName).append("\n");
            }
        }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
}
```

1st Rule: Functions should be small.

```
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();
}
```


Clean Code ch3 p33

How short should a function be?

"When Kent [Beck] showed me the code, I was struck by how small all the functions were. I was used to functions in Swing programs that took up miles of vertical space. Every function in *this* program was just two, or three, or four lines long. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. That's how short your functions should be!"

-- Robert C. Martin

Clean Code ch3 p34

2nd Rule: Functions should be smaller than that.

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

- Functions should **do one thing**. They should do it well. They should do it only.
- **One level of abstraction** per function.
- The function should **tell a story**.
  - "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."

Clean Code ch3 p35

One Level of Abstraction per Function

- Do not mix low and high levels of abstraction. Mixing them makes the function slower to read.
- **The Stepdown Rule**
  - Read code from top to bottom. It should read like a newspaper. Every function is followed by those at the next level of abstraction.
- Most developers are not sensitive to noticing multiple levels of abstraction, so the skill needs to be developed.
  - Try even: Extract methods until you can extract no more.
  - Give each method a meaningful name.

Clean Code ch3 p36

http://blog.objectmentor.com/articles/2009/09/11/one-thing-extract-till-you-drop

```
public class FizzBuzz {

    private static final int FIZZ = 3;
    private static final int BUZZ = 5;

    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++) {
            System.out.println(textForNumber(i));
        }
    }

    public static String textForNumber(int n) {
        if (multipleOf(FIZZ * BUZZ, n)) {
            return "FizzBuzz";
        }
        if (multipleOf(FIZZ, n)) {
            return "Fizz";
        }
        if (multipleOf(BUZZ, n)) {
            return "Buzz";
        }
        return Integer.toString(n);
    }

    private static boolean multipleOf(int multiplier, int n) {
        return n % multiplier == 0;
    }
}
```


http://stackoverflow.com/questions/1060366/how-long-should-it-take-a-senior-developer-to-solve-fizzbuzz-during-an-interview/1060857#1060857

Switch Statements

(also includes if-else chains)

- It's hard to make switch statements small.
  - They always do N things.
- "[Switch statements] can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance relationship so that the rest of the system can't see them."

Clean Code ch3 p37

What happens to the following code when new employee types must be added?

```
public Money calculatePay(Employee e) throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmpoyeeType(e.type);
    }
}
```

...and what about *isPayday(Employee e, Date date)*, *deliverPay(Employee e, Money pay)* and all other others?

Clean Code ch3 p38

Replace Switch With Polymorphism

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

public interface EmployeeFactory {
    Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}

public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
        switch (r.type) {
          case COMMISSIONED:
            return new CommissionedEmployee(r);
          case HOURLY:
            return new HourlyEmployee(r);
          case SALARIED:
            return new SalariedEmployee(r);
          default:
            throw new InvalidEmpoyeeType(r.type);
        }
    }
}
```


Clean Code ch3 p39

Function Names

- Use descriptive names.

  "You know you are working on clean code when each routine you read turns out to be pretty much what you expected."
  -- Ward Cunningham
- It's easier to choose a name for a short function.
- Don't be afraid of long names.
- Spend as much time as needed to choose the best names.

Clean Code ch3 p39

Function Arguments

- Number of arguments in order of preference: 0, 1, 2
  - Very rarely 3. More than three requires special justification.
- Avoid output arguments.
- Avoid flag arguments.

  render(boolean isSuite) → renderForSuite(), renderForSingleTest()
- Group related arguments to argument objects.

  Circle makeCircle(double x, double y, double radius)
  Circle makeCircle(Point center, double radius)

Clean Code ch3 p40

Have No Side Effects

```
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;
    }
}
```


Clean Code ch3 p44

Command Query Separation

- A function should either do something or answer something, but not both.
- What does the following mean?

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

- More descriptive name?

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

- Separate the command from the query, so that there will be no ambiguity.

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


Clean Code ch3 p45

How do you write functions like this?

"When I write functions, they come out long an 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."

-- Robert C. Martin

Clean Code ch3 p49

Comments

Comments do not make up for bad code

- "**Comments are, at best, a necessary evil.** If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much – perhaps not at all."
- "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!**"
- "**A comment is an apology** for not choosing a more clear name, or a more reasonable set of parameters, or for the failure to use explanatory variables and explanatory functions. Apologies for making the code unmaintainable, apologies for not using well-known algorithms, apologies for writing 'clever' code, apologies for not having a good version control system, apologies for not having finished the job of writing the code, or for leaving vulnerabilities or flaws in the code, apologies for hand-optimizing C code in ugly ways."

Clean Code ch4 p54
http://butunclebob.com/ArticleS.TimOttinger.ApologizeIncode

Express yourself in code

- Comments easily get out of sync with the code when the code is changed.
- When you get the temptation to clarify some code, instead of writing it down as a comment, write it down as code:

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

if (employee.isEligibleForFullBenefits())
```

- See earlier articles about choosing meaningful names.

Clean Code ch4 p55
http://programmer.97things.oreilly.com/wiki/index.php/Only_the_Code_Tells_the_Truth

Good Comments

- Legal comments; but keep them short (2-3 lines).
  - Mention year, copyright holder, license name and/or link.
- Informative comments; but first consider a better name.

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

- Explanation of intent; when inobvious design decisions. [*]

In some test code:

```
// This is our best attempt to get a race condition
// by creating a large number of threads.
```

- Clarification; but can you trust the comment?

```
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
...
```


Clean Code ch4 p55

Good Comments

- Warning of consequences; if there is no safer solution. [*]
- TODO comments; but remember to go through them later.
- Amplification; to highlight the importance of something seemingly inconsequential.

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

- Javadocs in public APIs; but you must take the effort to always keep them up-to-date and correct.

Clean Code ch4 p58

Bad Comments

- Mumbling; if you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write.
- Redundant comments; do not repeat what the code already says. [*]
- Misleading comments; which are not accurate enough.
- Mandated comments; don't follow conventions blindly.

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

Clean Code ch4 p59

Bad Comments

- Journal comments; a version control system is a much better place for change history.
- Noise comments; after seeing that the comments are not helpful, we begin to ignore them and they become just background noise.
- Position markers; use sparingly or they become noise.

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

- Closing brace comments; short methods don't need any.

```
} // while
```

- Attributions; keep them in version control.

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

- Commented-out code; delete! Version control remembers.

Clean Code ch4 p63

Bad Comments

- HTML comments; source code is for people to read. Any tools should take care of the formatting automatically.
- Non-local information; unrelated, easily gets out of sync.

```
/** Port on which fitnesse would run. Defaults to <b>8082</b>. */
public void setFitnessePort(int fitnessePort) {
    this.fitnessePort = fitnessePort;
}
```

- Too much information; just mention the RFC number or provide a link, don't copy all of its text in a comment.
- Inobvious connection; the connection between the comment and the code should be obvious to the reader. [*]
- Function headers; short well-named functions don't need.
- Javadocs in non-public code; the code and tests are already enough, no need for extra formality.

Clean Code ch4 p69

Course Material

- *Clean Code*, chapter 3: Functions
- *Clean Code*, chapter 4: Comments
- *Clean Code*, chapter 5: Formatting