Clean Code

Chapter 3: Functions

Siddhartha Varma

SDE Intern - backend

Core Team

BRO3886

sidv_22

sidv.dev

HtmlUtil.java

```
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, wikiPage);
            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");
        }
    }
```

Contd.

```
    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, wikiPage);
            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();
}
```

Refactored

```
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();
}
```

What makes the refactored code easier to read?

Small

- The first rule of functions is that they should be small.
- The second rule of functions is that *they should be smaller than that*.

How short should a function be?

Three, or four, or five lines long.

Every once in a while, even six lines long.

Red Flags 🚩

```
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, wikiPage);
            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");
        }
    }
```

More Red Flags 🚩

```
    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, wikiPage);
            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();
}
```

Do One Thing

- It should be very clear that `testableHtml()` is doing lots more than one thing.

- Creating buffers
- fetching pages
- searching for inherited pages
- rendering paths
- appending arcane strings
- and generating HTML, among other things.

"Functions should do one thing. They should do it well. They should do it only."

Robert C. Martin (Uncle Bob)

How would I know if I’m doing it right?

- It is hard
- If a function does only those steps that are one level below the stated name of the function
- The reason we write functions is to decompose a larger concept
- (in other words, the name of the function)

Indenting

- 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 makes the function easier to read and understand
- (reads just like a well-written prose)

One Level of Abstraction per Function

- Read like a top-down narrative
- Every function to be followed by those at the next level of abstraction
- Descending one level of abstraction at a time
- The Step-down Rule
- Key to keeping functions short and making sure they do “one thing”
- It is hard.

Avoid Switch Statements

They Break.

Use polymorphism instead.

"You know you are working on clean code when each routine turns out to be pretty much what you expected."

Ward Cunningham (inventor of the Wiki)

Use descriptive names

```
public static String ~~testableHtml~~ renderPageWithSetupsAndTeardowns(PageData pageData,…
```

- Better describes what the function does
- Don’t be afraid to make a name long
- It is better than a short enigmatic name
- It is better than a long descriptive comment

Number of Function Arguments

- Zero, One, Two
- Three (should avoid where possible)
- More than three (yikes)

Flag Arguments (booleans)

NO
Complicates the signature of the method (function does more than one thing).
It does one thing if the flag is true and another if the flag is false.

Have no side effects

- Your function promises to do one thing, but it also does other *hidden things*.

```
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;
  }
}
```

Prefer Exceptions to Returning Error Codes

Command Query Separation:

Functions should either do something or answer something, but not both.

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

Error

```
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;
}
```

- `Error` enum == *Dependency Magnet*

Exception

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

- Yes, try-catch is ugly and confusing
- Only have one method call in try-catch block
- The call is what should be causing the exception

- Never use nested try-catch

"Duplication may be the root of all evil in software"

Robert C. Martin (Uncle Bob)

Don’t Repeat Yourself

- Codd’s normal forms
- Structured programming, Aspect Oriented Programming, Component Oriented Programming
- Strategies for eliminating duplication

Not Clean Coder Sid 🤷

```
@Service
public class SomeServiceImpl implements SomeService {
  @Override
  public Response doSomething() {
    try {
      Success success = feignClient.doSomethingInternal();
      return SuccessResponse.create(success);
    } catch (FeignException e) {
      log.error(e.getMessage());
      return ErrorResponse.create(e);
    }
  }

  @Override
  public Response doSomethingElse() {
    try {
      OtherSuccess success = feignClient.doSomethingElseInternal();
      return SuccessResponse.create(success);
    } catch (FeignException e) {
      log.error(e.getMessage());
      return ErrorResponse.create(e);
    }
  }
```

Clean Coder Sid 😎

```
@Service
public class SomeServiceImpl implements SomeService {
  @Override
  public Response doSomething() {
    Success success = feignClient.doSomethingInternal();
    return SuccessResponse.create(success);
  }

  @Override
  public Response doSomethingElse() {
    OtherSuccess success = feignClient.doSomethingElseInternal();
    return SuccessResponse.create(success);
  }

  ...
}
```

Structured Programming

- Edsger Dijkstra’s rules of structured programming

> Every function and every block within a function, should have one entry and one exit.

- There should only be one return statement
- No break or continue statements in a loop
- and never, ever, any goto statements.

But there’s a catch

If you keep your functions small, then the occasional multiple `return`, `break`, or `continue` statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule.

Recap

- Small
- Do One Thing
- One Level of Abstraction per Function
- Avoid Switch Statements
- Use descriptive names
- Number of Function Arguments
- Have no side effects
- Prefer Exceptions to Returning Error Codes
- Don’t Repeat Yourself
- Structured Programming

These slides are available online at

talks.sidv.dev/2022/clean-code-ch3

Thank You!