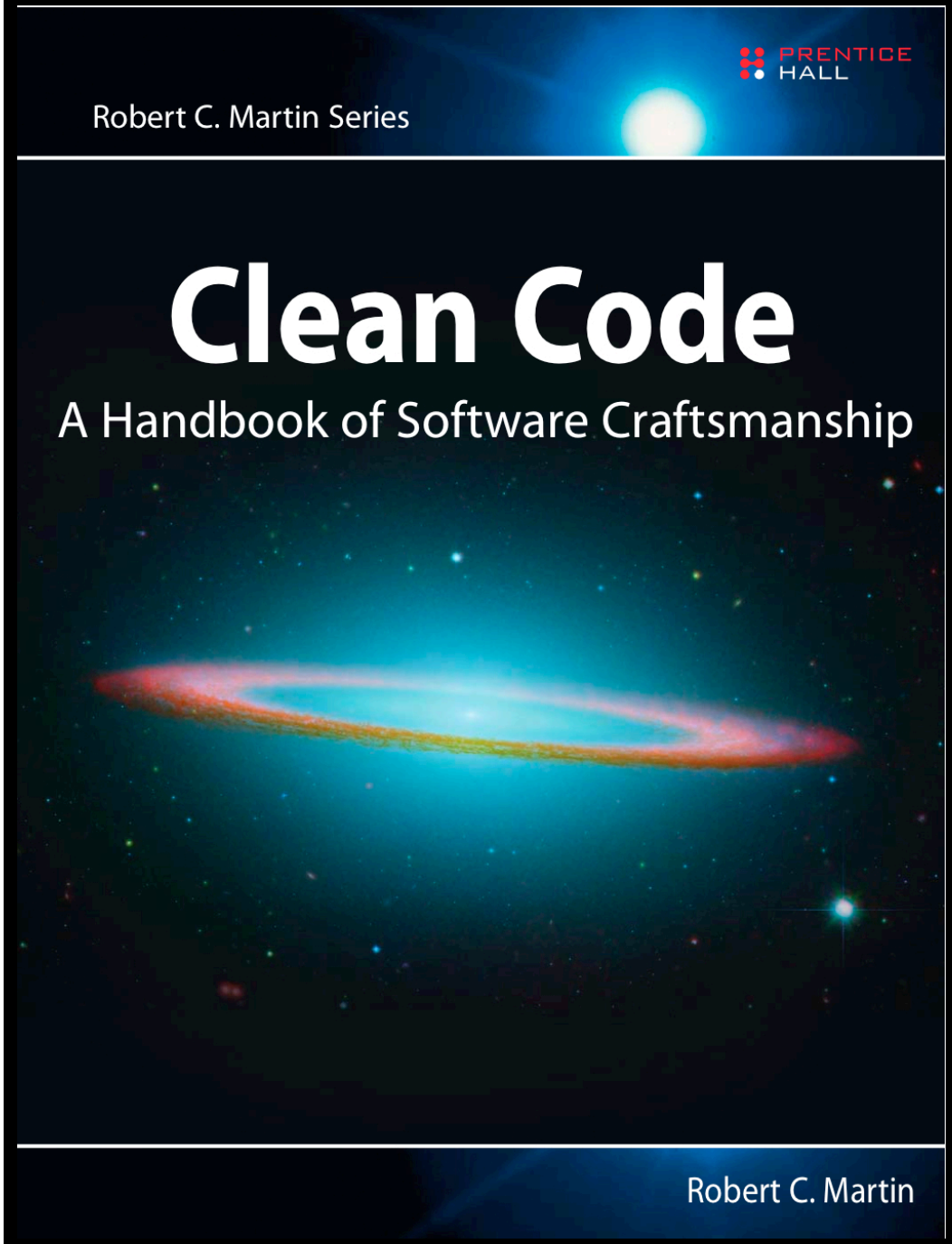


Clean Code III
Functions

Michael Feathers
channeling
Robert C. Martin

Object Mentor, Inc.

objectmentor.com

The First Line of Organization

- In the early days of programming
  - we composed our systems of routines and subroutines.
  - in Fortran it was programs, subprograms, and functions.
- Nowadays only the function survives.

A “Long” function in FitNesse

- Not only is it long, but it’s got
  - duplicated code,
  - lots of odd strings,
  - many strange and inobvious data types and APIs.
- See how much sense you can make of it in the next three minutes…

A “Long” function in FitNesse 1

```
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);
```

A “Long” function in FitNesse 2

```
  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,
              wikiPage
      );
```

A “Long” function in FitNesse 3

```
      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();
}
```

How did you do?

- Do you understand the function after three minutes of study?
- Probably not.
  - There's too much going on in there,
  - at too many different levels of abstraction.
  - There are strange strings
  - odd function calls
  - doubly nested if statements controlled by flags.

  - Ick!

Nothing up my sleve…

- With just a few simple
  - method extractions,
  - some renaming,
  - and a little restructuring,
- I was able to capture the intent of the function.
- See if you can understand the result in the next 3 minutes?

Refactored Function

```
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();
 }
```

You probably don’t understand it all.

- Still you probably understand that it:
  - includes setup and teardown pages into a test page,
  - renders that page into HTML.

What's more…

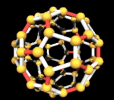
- You also probably realize:
  - That this function belongs to some kind of web-based testing framework.
- Divining that information from the refactored function is pretty easy,
- but it's pretty well obscured by the initial code.

So what is the magic?

- What is it that makes the refactored function easy to read and understand?
- How can we make a function communicate its intent?
- What attributes can we give our functions that will allow a casual reader to intuit the kind of program they live inside?

Small!

The First Rule of Functions.



The rules of functions:

- The first rule:
  - They should be small.
- The second rule:
  - They should be smaller than that.

A Screenful?

- In the ‘80s we used to say that a function should be no bigger than a screenful.
  - Of course VT100 screens were 24 lines by 80 columns,
  - and our editors used 4 lines for administrative purposes.
- Nowadays with a cranked down font and a nice big monitor
  - you can fit 150 characters on a line, and a 100 lines or more on a screen. Lines should not be 150 characters long.

Smaller Than a Screenful

- Functions should not be 100 lines long.
- Functions should hardly ever be 20 lines long.
- Indeed, the refactored function was too long.
  - It should have been:

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

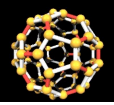
Blocks

- Smallness implies that blocks within:
  - if statements,
  - else statements,
  - while statements,
  - and etc.,
- should be one line long.
- Probably that line should be a function call.
  - Not only does this keep the function small;
  - but it also adds documentary value

Indenting

- Smallness also implies:
  - 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.

Do One Thing



Functions should do one thing.

- They should do it well.
- They should do it only.

Doing More Than One Thing

- The original code does lots more than one thing.
  - It's creating buffers,
  - fetching pages,
  - searching for inherited pages,
  - rendering paths,
  - appending arcane strings,
  - and generating HTML,
  - among other things.
- The re-refactored code is doing one simple thing.
  - including setups and teardowns into test pages.

Or is it?

- It's easy to make the case that it's doing 3 things:
- Determine whether the page is a test page.
- If so, include setups and teardowns.
- Render the page in HTML.
- So which is it?
  - Is the function doing one thing,
  - or three things?

All At Same Level…

- The steps are one level of abstraction below the name of the function.
- A brief TO paragraph:
  - TO RenderPageWithSetupsAndTeardowns we:
    - check to see if 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's steps are one level below the stated name of the function,
  - then the function is doing one thing.

The reason we write functions is to:

- Decompose a larger concept
  - (i.e. the name of the function)
- into a set of steps at the next level of abstraction.

Doing One Thing!

- It should be very clear that
  - The original code contains steps at many different levels of abstraction.
  - So it is clearly doing more than one thing.
- Even the first refactoring has two levels of abstraction,
  - as proved by our ability to shrink it down.
- But it would be very hard to meaninfully shrink the final.
  - We could extract the if statment into a function named includeSetupsAndTeardownsIfTestPage,
  - but that simply restates the code without changing the level of abstraction.

Doing One Thing!

- You can tell that a function is doing more than “one thing”
  - if you can extract a function from it
  - with a name that is not merely a restatement of its implementation.

Reading code from top to bottom.

- We want the code to read like a top-down narrative.
- We want every function to be followed by those at the next level of abstraction,
  - We can read the program, descending one level of abstraction at a time.
- We want to read the program as if it were a set of TO paragraphs,
  - each of which describes the current level of abstraction
  - and references subsequent TO paragraphs at the next level down.

To Paragraphs:

- To include the setups and teardowns we
  - include setups,
  - then include the test page content,
  - then include the teardowns.
- To include the setups we
  - include the suite setup if this is a suite,
  - then include the regular setup.
- To include the suite setup we
  - search the parent hiearchy for the “SuiteSetUp” page
  - add an !include with the path of that page.
- To search the parent...

That’s how you do ONE THING.

Use descriptive names.

Example

- I changed the name of our example function
  - from testableHtml
  - To renderPageWithSetupsAndTeardowns.
    - This is a far better name.
- I also gave the private methods a descriptive name
  - such as isTestable
  - includeSetupAndTeardownPages.
- It is hard to overestimate the value of good names.

Remember Ward's principle:

- "You know you are working on clean code when each routine turns out to be pretty much what you expected."
- Half the battle to achieving that principle is
  - choosing good names
    - for small functions
    - that do one thing.

The Naming Heuristic

- The smaller and more focused a function is,
  - the easier it is to choose a descriptive name.
- Conversely, if you can't choose a descriptive name
  - Your function is probably too big
  - And does more than ONE THING.

Long Names

- Don’t be afraid to make a name long.
- A long descriptive name is better than
  - a short enigmatic name.
  - a long descriptive comment.
- Use a naming convention that allows multiple words to be easily read in the function names
  - Like Camel Case or Underscores.
    - IncludeSetUpAndTearDown
    - Include_setup_and_teardown
- Make use of those multiple words to give the function a name that says what it does.

It Takes Time

- Don’t be afraid to spend time choosing a name.
- Indeed, you should try several different names
  - and read the code with each in place.
- Modern IDEs like Eclipse or IntelliJ make it trivial to change names.
- Use one of those IDEs and experiment with different names until you find one that is as descriptive as you can make it.

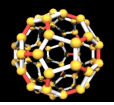
Names and Design

- Choosing descriptive names will clarify the design of the module in your mind,
  - and help you to improve it.
- Hunting for a good name often results in a favorable restructuring of the code.

Consistent Names

- Use the same phrases, nouns, and verbs in the function names you choose for your modules.
  - Consider, for example, the names
    - includeSetupAndTeardownPages,
    - includeSetupPages,
    - includeSuiteSetupPage,
    - includeSetupPage.
  - The similarity of those names allows the sequence to tell a story.
  - Indeed, if I showed you just the sequence above, you'd ask yourself:
    - "What happened to includeTeardownPages, includeSuiteTeardownPage, and includeTeardownPage?"
    - How's that for being "...pretty much what you expected."

No more than three arguments.



How many arguments?

- he 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.
- That's why I got rid of almost all of them from the example.
- Consider, for example, the StringBuffer in the example.
  - We could have passed it around as an argument
  - rather than making it an instance variable;
  - but then our readers would have had to interpret it each time they saw it.

Arguments are hard.

- 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 (i.e. StringBuffer) that isn't particularly important at that point.

Output arguments

- Harder to understand than input arguments.
- 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.

Common Monadic Forms

- There are two common reasons to pass a single argument into a function.
  - You may be asking a question about that argument as in: boolean fileExists(“MyFile”).
  - Or you may be operating on that argument,
    - transforming it into something else and returning it.
    - For example: InputStream fileOpen(“MyFile”) transforms a String into an InputStream return value.
- These two uses are what readers expect when they see a function.
  - You should choose names that make the distinction clear.

Flag Arguments

- Passing a boolean into a function is a truly terrible practice.
- It immediatly 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!

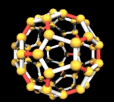
Dyadic Functions

- A function with two arguments is harder to understand than a monadic function.
  - writeField(name) is easier to understand than writeField(outputStream, name).
    - the first glides past the eye. easily depositing its meaning.
    - The second requires a short pause until we learn to ignore the first parameter.
  - We should never ignore any part of the code.
    - The parts we ignore are where the bugs will hide.

Triads

- Functions that take three arguments are significantly harder to understand than dyads.
  - The issues of ordering, pausing, and ignoring are more than doubled.
- Consider the common overload of assertEquals that takes three arguments:
  - assertEquals(message, expected, actual).
  - How many times have you read the message and thought it was the expected?
  - I have stumbled and paused over that particular triad many times.
  - In fact, every time I see it I do a double-take and then learn to ignore the message.

No side-effects.



Side-effects are lies.

- Your function promises to do one thing,
  - but it also does other, hidden, things.
    - to the variables of it's own class.
    - to the parameters passed into the function,
    - to system globals.
- They are devious and damaging mistruths that result in
  - strange temporal couplings
  - and order dependencies.

Side Effects

- Consider the seemingly innocuous function that uses a standard algorithm to match a userName to a password. It returns true if they match, and false if anything goes wrong.
- But it also has a side-effect.
  - Can you spot it?

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;
  }
}
```

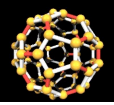
Side Effects

- 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 they decide to check the validity of the user.

Temporal Couplings

- The side-effect creates a temporal coupling.
  - checkPassword can only be called at certain times
    - (i.e. 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”.

Command Query Separation



Asking vs. Telling

- 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.

Example

- Consider, for example, the following function:

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

- It sets the value of a named attribute
  - returns true if it is successful
  - false if no such attribute exists.

Example

- This leads to odd statements like this:

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

- What does that mean?
  - Is it asking whether the “username” attribute was
    - previously set to “unclebob”?
    - successfully set to “unclebob”?
  - It’s hard to infer the meaning from the call because it’s not clear whether the word “set” is a verb or an adjective.

Example

- The author intended set to be a verb,
  - but in the context of the if statement it feels like an adjective.
  - So the statement reads as:
    - “If the username attribute was previously set to unclebob”
  - and not as:
    - “set the username attribute to unclebob and if that worked then...”.
  - We could try to resolve this by renaming the set function to setAndCheckIfExists,
    - but that doesn’t much help the readability of the if statement.

Example

- The real solution is to separate the command from the query
  - so that the ambiguity cannot occurr.

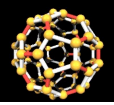
```
if (attributeExists("username")) {

  setAttribute("username", "unclebob");

  ...

}
```

Prefer exceptions to returning error codes.



Returning errror codes

- A subtle violation of command query separation.
- It promotes commands being used as expressions in the predicates of if statements.

- if (deletePage(page) == E_OK)

- This leads to deeply nested structures.
- The caller must deal with the error immediatly.

Returning Error Codes

```
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;
}
```

Using Exceptions

- If you use exceptions 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());
}
```

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.

Extract try/catch blocks

```
 public void delete(Page page) {
   try {
     deletePageAndAllReferences(page);
   }
   catch (Exception e) {
     logError(e);
   }
 }

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

Error handling is one thing.

- A function that handles errors should do nothing else.
- This implies that the keyword try should be the very first word in the function;
- and that there should be nothing after the catch/finally blocks.

Structured Programming

- Dijkstra said:
  - every function,
  - and every block within a function,
  - should have one entry and one exit.
- Following these rules means
  - there should only be one return statement in a function,
  - no break or continue statements in a loop,
  - and never, ever, any goto statements.

Structured Programming Eclipsed

- While we are sympathetic to the goals and disciplines of structured programming,
  - those rules serve little benefit when functions are very small.
  - It is only in larger functions that such rules provide significant benefit.
- So in small functions 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.
  - But goto should still be avoided.

Conclusion

Domain Specific Languages

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

Programs as Stories

- 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 DSL that they use to tell that story.
  - Part of that DSL 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 DSL they define to tell their own small part of the story.

The “Clean Code” project.

- Articles:
  - The “Args” article.
  - The “Clean Code” book.

Contact Information

- Robert C. Martin
  unclebob@objectmentor.com
- Website:
  www.objectmentor.com
- FitNesse:
  www.fitnesse.org