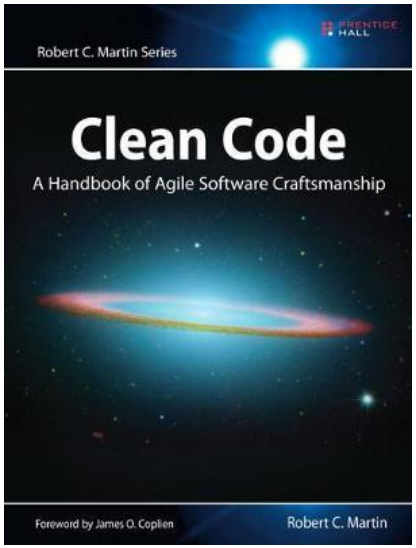
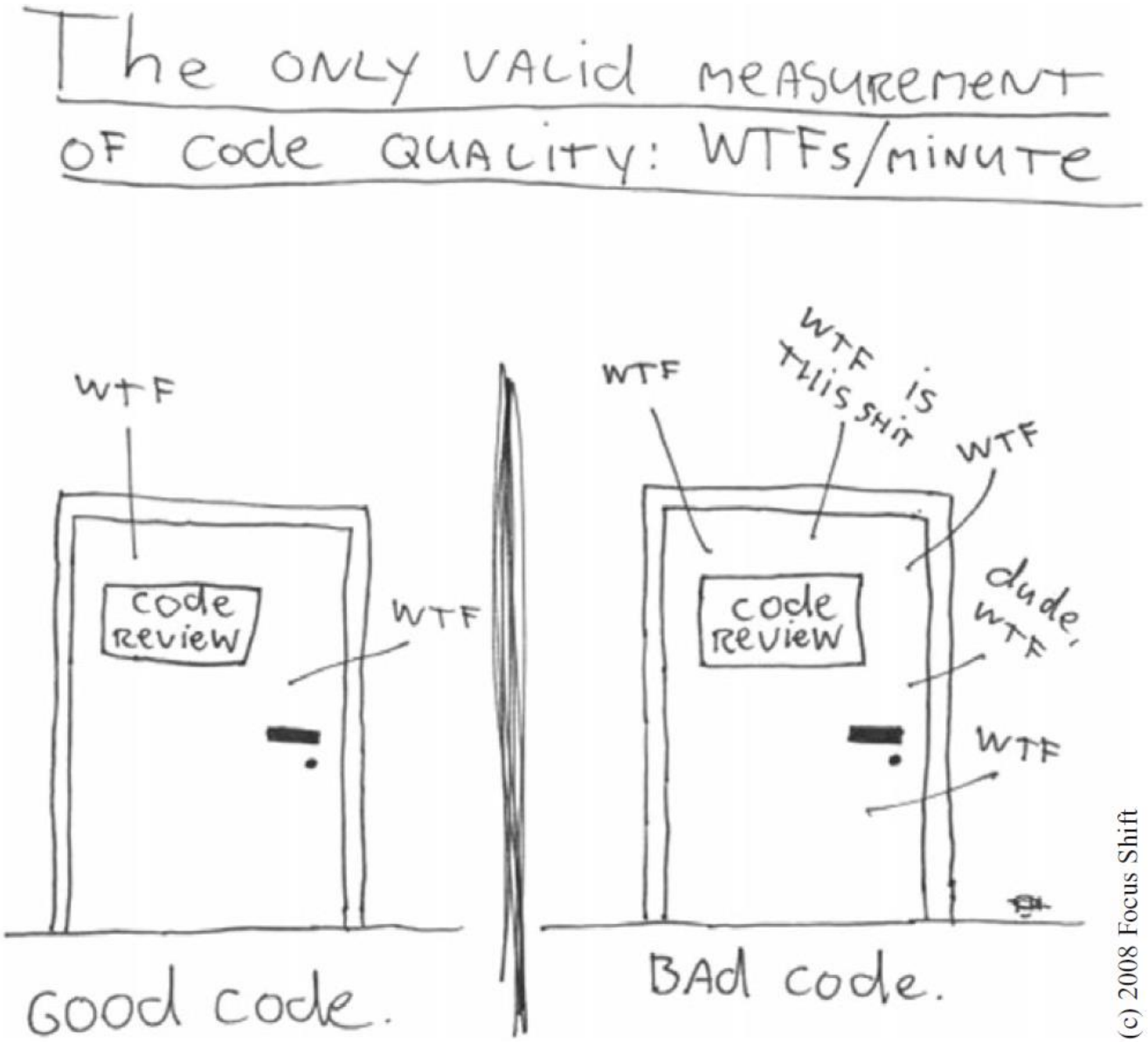
Clean Code

"Clean Code: A Handbook of Agile Software Craftsmanship,"
Robert C. Martin, Prentice Hall, 2008



Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University



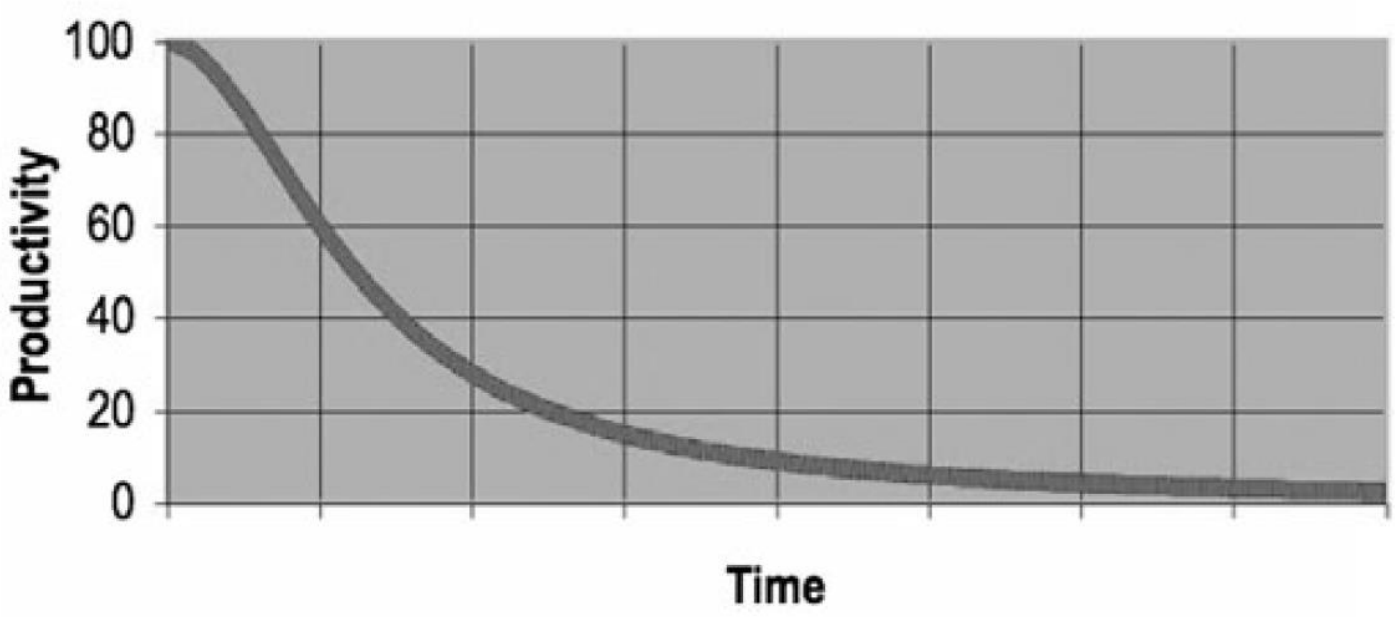
Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. Good. We need better programmers.

Bad Code

- I know of one company that, in the late 80s, wrote a *killer* app. It was very popular, and lots of professionals bought and used it.
  - But then the release cycles began to stretch.
  - Bugs were not repaired from one release to the next.
  - Load times grew and crashes increased.
  - I remember the day I shut the product down in frustration and never used it again. The company went out of business a short time after that.
- Two decades later I met one of the early employees of that company and asked him what had happened. The answer confirmed my fears.
  - They had rushed the product to market and had made a huge mess in the code.
  - As they added more and more features, the code got worse and worse until they simply could not manage it any longer.
  - *It was the bad code that brought the company down.*

The Total Cost of Owning a Mess



Meaningful Names

Use Intention-Revealing Names

- Choosing good names takes time but saves more than it takes

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Avoid Disinformation

- Programmers must avoid leaving false clues that obscure the meaning of code.
- Do not refer to a grouping of accounts as an `accountList` unless it's actually a `List`. So `accountGroup` or `bunchOfAccounts` or just plain `accounts` would be better.
- Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, `XYZControllerForEfficientStorageOfStrings`?
- A truly awful example of disinformative names would be the use of lower-case `L` or uppercase `O` as variable names

```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```

Make Meaningful Distinctions

- Number-series naming `(a1, a2, .. aN)` is the opposite of intentional naming. This function reads much better when `source` and `destination` are used for the argument names.

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

- Noise words are another meaningless distinction. Imagine that you have a `Product` class. If you have another called `ProductInfo` or `ProductData`, you have made the names different without making them mean anything different.
- Noise words are redundant. How are the programmers supposed to know which of these functions to call?

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Use Pronounceable Names

- If you can't pronounce it, you can't discuss it without sounding like an idiot.

```
class DtaRcrd102 {
private Date genymdhms;
private Date modymdhms;
private final String pszqint = "102";
/* ... */
};
```

- Intelligent conversation is now possible: "Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"

```
class Customer {
private Date generationTimestamp;
private Date modificationTimestamp;;
private final String recordId = "102";
/* ... */
};
```

Use Searchable Names

- Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

```
for (int j=0; j<34; j++) {
s += (t[j]*4)/5;
}
```

- The intentionally named code makes for a longer function, but consider how much easier it will be to find `WORK_DAYS_PER_WEEK` than to find all the places where 5 was used and filter the list down to just the instances with the intended meaning.

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
int realTaskDays = taskEstimate[j] *
realDaysPerIdealDay;
int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
sum += realTaskWeeks;
}
```

Member Prefixes

- People quickly learn to ignore the prefix (or suffix) to see the meaningful part of the name.
- The more we read the code, the less we see the prefixes. Eventually the prefixes become unseen clutter and a marker of older code.

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Don’t Be Cute

- Choose clarity over entertainment value.
- `HolyHandGrenade => DeleteItems`
- Cuteness in code often appears in the form of colloquialisms or slang. For example, don’t use the name `whack()` to mean `kill()`. Don’t tell little culture-dependent jokes like `eatMyShorts()` to mean `abort()`.

Pick One Word per Concept

- Pick one word for one abstract concept and stick with it. For instance, it's confusing to have `fetch`, `retrieve`, and `get` as equivalent methods of different classes.
- It's confusing to have a `controller` and a `manager` and a `driver` in the same code base.
  - What is the essential difference between a `DeviceManager` and a `Protocol-Controller`?
  - Why are both not `controllers` or both not `managers`?
  - Are they both Drivers really? The name leads you to expect two objects that have very different type as well as having different classes.

Functions

Small!

- A function should be no bigger than a screen-full
  - It fits in an "**eye-full**"
- 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. Functions should not be 100 lines long.
  - Functions should hardly ever be 20 lines long.

One Level of Abstraction per Function

- In order to 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.

Before refactoring

Listing 3-1

HtmlUtil.java (FitNesse 20070619)

```
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)
```

After refactoring

Listing 3-3

HtmlUtil.java (re-refactored)

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

Reading Code from Top to Bottom: The Stepdown Rule

- 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 so that we can read the program, descending one level of abstraction at a time as we read down the list of functions.

To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.

To include the setups, we include the suite setup if this is a suite, then we include the regular setup.

To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.

To search the parent. . .

```
public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;


  public static String render(PageData pageData) throws Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
```

Prefer Exceptions to Returning Error Codes

- if you use exceptions instead of returned error codes, then the error processing code can be separated from the happy path code and can be simplified

```
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;
}
```

```
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.

```
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());
}
```

How Do You Write Functions Like This?

- When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists.
  - The names are arbitrary, and there is duplicated code.
- So then I massage and refine that code, splitting out functions, changing names, eliminating duplication.
  - I shrink the methods and reorder them.
- 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.

Comments

Comments Do Not Make Up for Bad Code

- One of the more common motivations for writing comments is bad code.
  - 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!
- Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments.
- Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

Explain Yourself in Code

- It takes only a few seconds of thought to explain most of your intent in code. In many cases it's simply a matter of creating a function that says the same thing as the comment you want to write.

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

Or this?

```
if (employee.isEligibleForFullBenefits())
```

GOOD COMMENTS

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.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

Informative Comments

- It is sometimes useful to provide basic information with a comment

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

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

Explanation of Intent

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

```
//This is our best attempt to get a race condition
//by creating large number of threads.
for (int i = 0; i < 25000; i++) {
  WidgetBuilderThread widgetBuilderThread =
    new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
  Thread thread = new Thread(widgetBuilderThread);
  thread.start();
}
assertEquals(false, failFlag.get());
```

Clarification

- When its part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful.

```
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
assertTrue(b.compareTo(a) == 1);     // b > a
assertTrue(ab.compareTo(aa) == 1);   // ab > aa
assertTrue(bb.compareTo(ba) == 1);   // bb > ba
```

Warning of Consequences

- Sometimes it is useful to warn other programmers about certain consequences

```
// Don't run unless you
// have some time to kill.
public void _testWithReallyBigFile()
{
  writeLinesToFile(10000000);
  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```

TODO Comments

- TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment. It might be a reminder to delete a deprecated feature or a plea for someone else to look at a problem.

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
   return null;
}
```

BAD COMMENTS

Mumbling

- Plopping in a comment just because you feel you should or because the process requires it, is a hack.
- If you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write.

```
public void loadProperties()
{
   try
   {
    String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
    FileInputStream propertiesStream = new FileInputStream(propertiesPath);
    loadedProperties.load(propertiesStream);
   }
   catch(IOException e)
   {
     // No properties files means all defaults are loaded
   }
}
```

Redundant Comments

- The comment probably takes longer to read than the code itself.

Listing 4-1

waitForClose

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

Mandated Comments

- It is just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment.

Listing 4-3

```
/**
 *
 * @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.duration = duration;
  cdList.add(cd);
}
```

Noise Comments

- Sometimes you see comments that are nothing but noise. They restate the obvious and provide no new information.

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

No, *really?* Or how about this:

```
/** The day of the month. */
    private int dayOfMonth;
```

And then there's this paragon of redundancy:

```
/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

Don't Use a Comment When You Can Use a Function or a Variable

Consider the following stretch of code:

```
// does the module from the global list <mod> 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))
```

Commented-Out Code

- Others who see that commented-out code won't have the courage to delete it. They'll think it is there for a reason and is too important to delete.

```
    InputStreamResponse response = new InputStreamResponse();
    response.setBody(formatter.getResultStream(), formatter.getByteCount());
//  InputStream resultsStream = formatter.getResultStream();
//  StreamReader reader = new StreamReader(resultsStream);
//  response.setContent(reader.read(formatter.getByteCount()));
```

Nonlocal Information

- If you must write a comment, then make sure it describes the code it appears near. Don't offer systemwide information in the context of a local comment

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

Formatting

Formatting

- None are over 500 lines and most of those files are less than 200 lines.

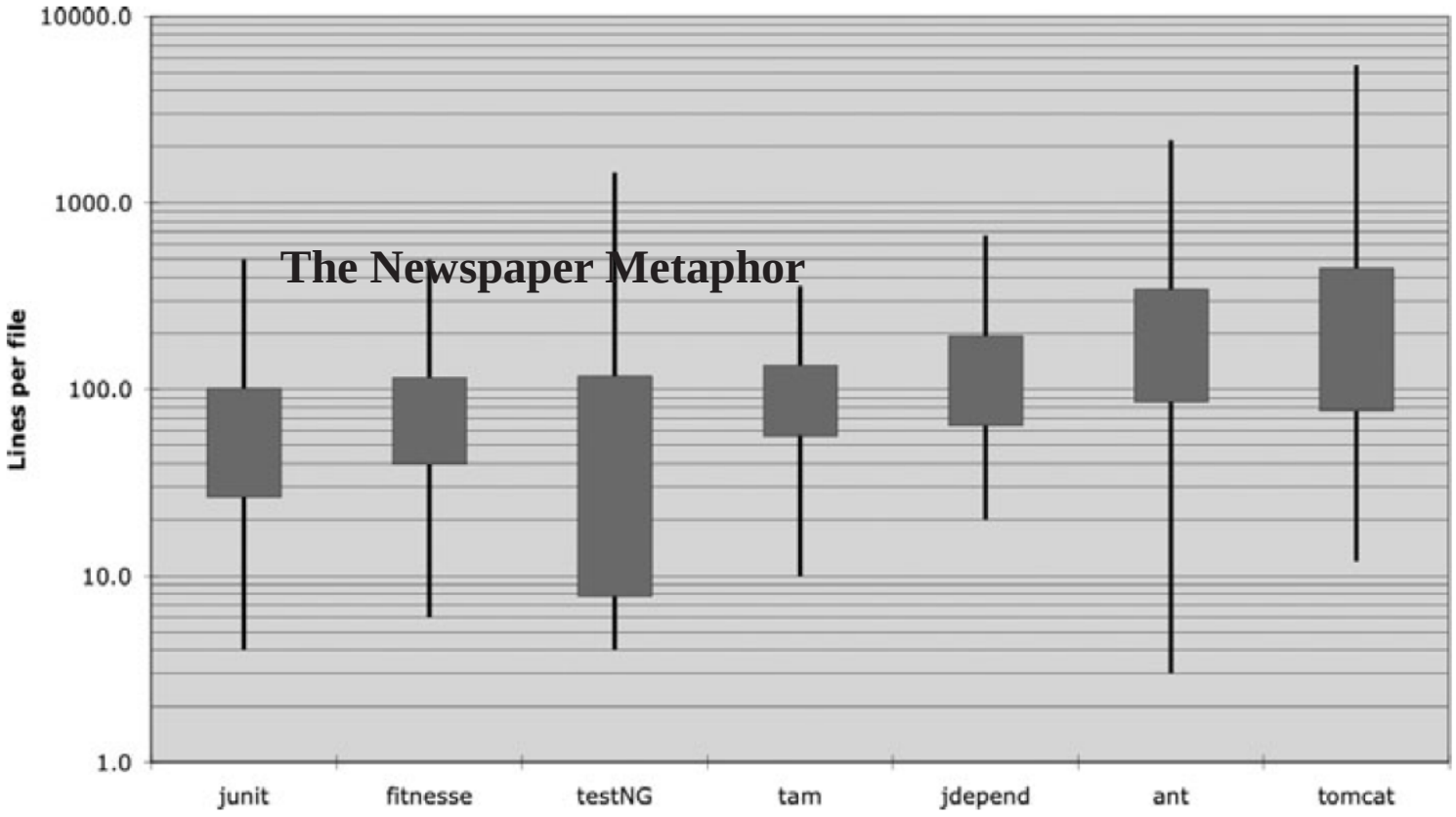


Figure 5-1

File length distributions LOG scale (box height = sigma)

The Newspaper Metaphor

- We would like a source file to be like a newspaper article.
  - The name should be simple but explanatory.
  - The topmost parts of the source file should provide the high-level concepts and algorithms.
  - Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.
- A newspaper is composed of many articles; most are very small. Some are a bit larger.
  - If the newspaper were just one long story containing a disorganized agglomeration of facts, dates, and names, then we simply would not read it.

Vertical Openness Between Concepts

- Each line represents an expression or a clause, and each group of lines represents a complete thought.
- Those thoughts should be separated from each other with blank lines.

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Vertical Density

- If openness separates concepts, then vertical density implies close association. It fits in an "**eye-full**"

Listing 5-3

```
public class ReporterConfig {

  /**
   * The class name of the reporter listener
   */
  private String m_className;

  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();

  publi
    m_p
  }
```

Listing 5-4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Vertical Distance

- Have you ever chased your tail through a class, hopping from one function to the next, scrolling up and down the source file, trying to divine how the functions relate and operate?

- Concepts that are closely related should be kept vertically close to each other.

- We want to avoid forcing our readers to hop around through our source files and classes.

Variable Declarations

- Variables should be declared as close to their usage as possible.
- Because our functions are very short, local variables should appear at the top of each function

```
private static void readPreferences() {
   InputStream is= null;
   try {
      is= new FileInputStream(getPreferencesFile());
      setPreferences(new Properties(getPreferences()));
      getPreferences().load(is);
   } catch (IOException e) {
      try {
         if (is != null)
            is.close();
      } catch (IOException e1) {
      }
   }
}
```

Dependent Functions

- If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible.

Listing 5-5
WikiPageResponder.java

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;

    return pageName;
  }

  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
```

Conceptual Affinity

- Affinity might be caused because a group of functions perform a similar operation.

```
public class Assert {
   static public void assertTrue(String message, boolean condition) {
      if (!condition)
         fail(message);
   }

   static public void assertTrue(boolean condition) {
      assertTrue(null, condition);
   }

   static public void assertFalse(String message, boolean condition) {
      assertTrue(message, !condition);
   }

   static public void assertFalse(boolean condition) {
      assertFalse(null, condition);
   }
...
```

Horizontal Openness and Density

- We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related
- Another use for white space is to accentuate the precedence of operators

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

Horizontal Alignment

- In the list of declarations you are tempted to read down the list of variable names without looking at their types.

```
private   Socket          socket;
private   InputStream     input;
private   OutputStream    output;
private   Request         request;
private   Response        response;
private   FitNesseContext context;
protected long            requestParsingTimeLimit;
private   long            requestProgress;
private   long            requestParsingDeadline;
private   boolean         hasError;
```

- Prefer unaligned declarations and assignments

```
private Socket socket;
private InputStream input;
private OutputStream output;
private Request request;
```

Indentation

- Without indentation, programs would be virtually unreadable by humans

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

```
public FitNesseServer(FitNesseContext context) {
  this.context = context;
}

public void serve(Socket s) {
  serve(s, 10000);
}

public void serve(Socket s, long requestTimeout) {
  try {
    FitNesseExpediter sender = new FitNesseExpediter(s, context);
    sender.setRequestParsingTimeLimit(requestTimeout);
    sender.start();
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

Breaking Indentation

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return ""; }
}
```

Prefer to expand and indent the scopes instead, like this

```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }

  public String render() throws Exception {
    return "";
  }
}
```

Objects and Data Structures

Data Abstraction

- A class does not simply push its variables out through getters and setters.
- Rather it exposes abstract interfaces that allow its users to manipulate the *essence* of the data, without having to know its implementation.

Listing 6-3

Concrete Vehicle

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

Listing 6-4

Abstract Vehicle

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

Data/Object Anti-Symmetry

- *Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures.*
- *OO code, on the other hand, makes it easy to add new classes without changing existing functions.*

- Mature programmers know that the idea that everything is an object *is a myth*. Sometimes you really *do* want simple data structures with procedures operating on them.

Data Structure

Listing 6-5

Procedural Shape

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
```

Object

Listing 6-6

Polymorphic Shapes

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

Hybrids

- This confusion sometimes leads to unfortunate hybrid structures that are half object and half data structure.
- Such hybrids make it hard to add new functions but also make it hard to add new data structures.
- They are the worst of both worlds.

Data Transfer Objects

- The quintessential form of a data structure is a class with public variables and no functions. This is sometimes called a data transfer object, or DTO.
- Somewhat more common is the "bean" form shown in Listing 6-7. Beans have private variables manipulated by getters and setters.
  - The quasi-encapsulation of beans seems to make some OO purists feel better but usually provides no other benefit

Listing 6-7

address.java

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }
```

Train Wrecks

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

- This kind of code is often called a *train wreck* because it look like a bunch of coupled train cars.
- Chains of calls like this are generally considered to be sloppy style and should be avoided. It is usually best to split them up as follows:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```