Clean Code

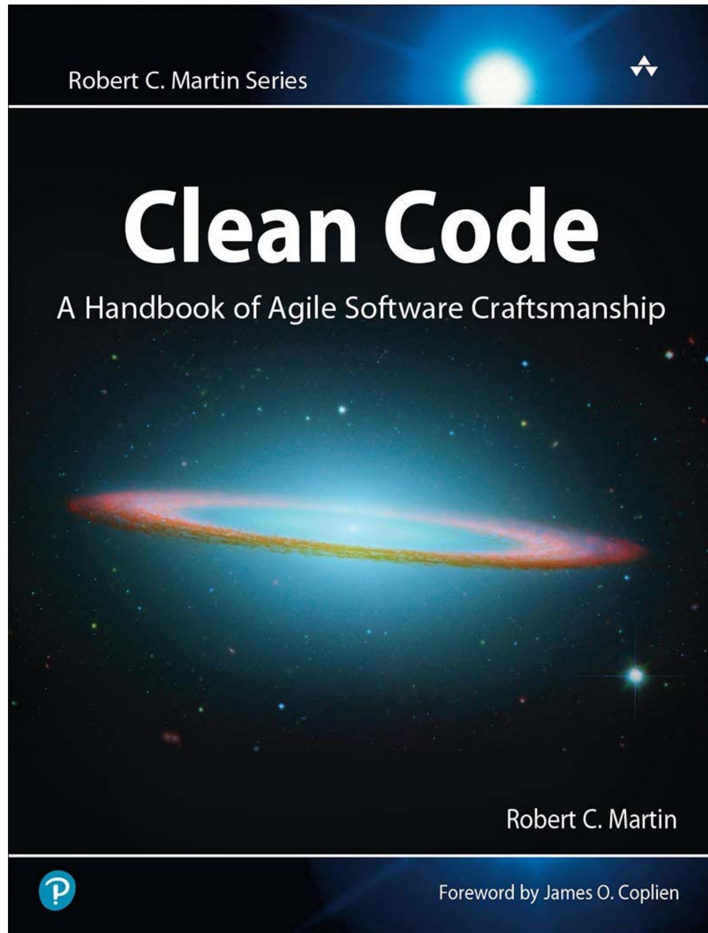
JUNBEOM YOO

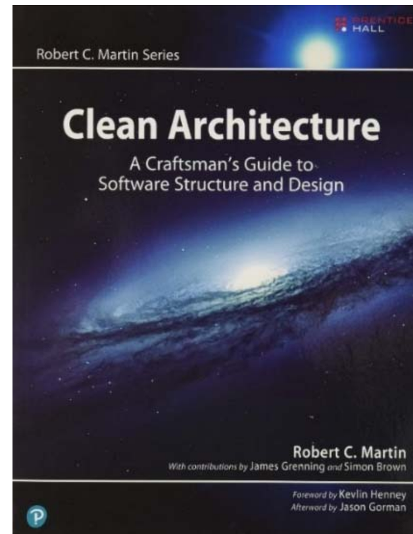
KONKUK University
http://dslab.konkuk.ac.kr

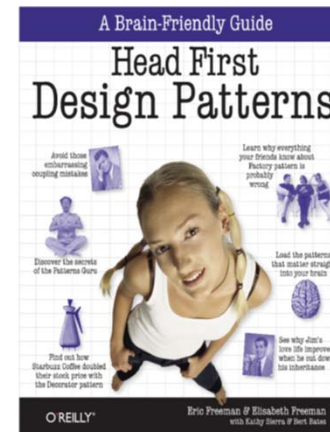
KU KONKUK UNIVERSITY

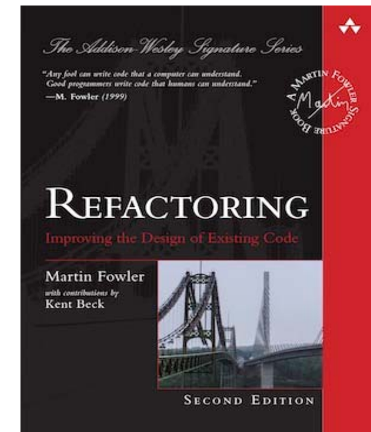
DEPENDABLE SOFTWARE LABORATORY

Text and References





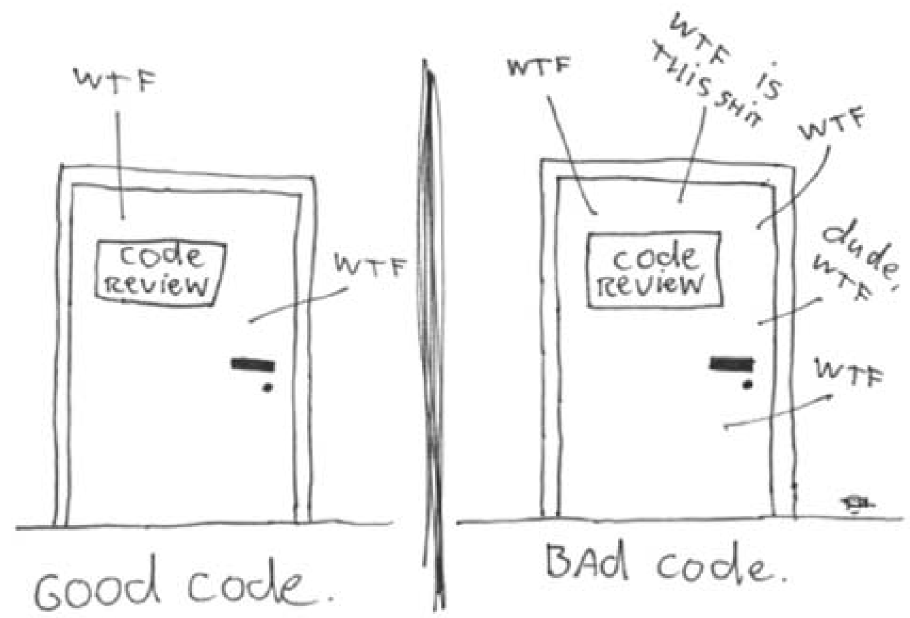




CLEAN CODE

1. Clean Code
2. Meaningful Names
3. Functions
4. Comments
5. Formatting
6. Objects and Data Structures
7. Error Handling
8. Boundaries
9. Unit Tests
10. Classes

- Two parts to learning craftsmanship: ***knowledge*** and ***work***



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

Chapter 2. Meaningful Names

2. Meaningful Names

- **Names** are everywhere in software.
  - We name our *variables*, our *functions*, our *arguments*, *classes*, and *packages*.
  - We name our *source files* and the *directories* that contain them.

- Because we do so much of it, we'd better do it well.
  - Some *simple rules for* ***creating good names***

KU KONKUK UNIVERSITY

2.1 USE INTENTION-REVEALING NAMES

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

- The name of a variable, function, or class, should answer the questions:
  - *What it does?*
  - *Why it exists?*
  - *How it used?*

- If a name requires a ***comment***, then the name does ***not*** reveal its intent.

```
int d; // elapsed time in days
```

  - The name *d* reveals nothing.
    - It does not evoke a sense of elapsed time, nor of days.
    - We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

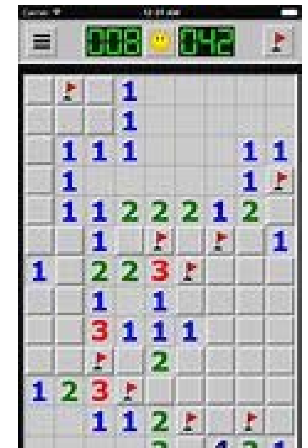
- ***Choosing names that reveal intent*** make it much easier to understand and change code.

- What is the purpose of this code? Why is it ***hard*** to tell what this code is doing?

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

  – The problem is ***not the simplicity*** of the code, but the ***implicity*** of the code.

- The code implicitly requires us to ask questions such as:
  1. What kinds of things are in *theList*?
  2. What is the significance of the zeroth subscript of an item in *theList*?
  3. What is the significance of the value 4?
  4. How would I use the list being returned?



- Assume that we're working in <u>a mine sweeper game</u>.
  - The board is a list of cells called *theList*. Let's rename that to *gameBoard*.
  - Each cell on the board is represented by a simple array.
  - The zeroth subscript is the location of a status value and a status value of 4 means "*flagged*."

- Just by giving these concepts names, we can improve the code considerably:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

- With these simple name changes, it gets easier to understand what's going on.

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

2.2 AVOID DISINFORMATION

- Programmers must ***avoid leaving false clues*** that obscure the meaning of code.
  - We should avoid words whose entrenched meanings vary from our intended meaning.
    - *hp*, *aix*, and *sco* would be poor variable names.
  - Do not refer to a grouping of accounts as an *accountList* unless it's actually a *List*.
  - Beware of using names which vary in small ways.
    - *XYZControllerForEfficientHandlingOfStrings* vs. *XYZControllerForEfficientStorageOfStrings*
  - Spelling similar concepts similarly is information.

- A truly awful example
  - The use of lower-case L or uppercase O as variable names, especially in combination.

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

2.3 MAKE MEANINGFUL DISTINCTIONS

- Problems happen *when programmers write code only to satisfy a compiler*.
  - It is not sufficient to add number series or ***noise words***, even though the compiler is satisfied.

- If names must be different, then they should also mean something different.
  - Noise words are another meaningless distinction.
  - *Noise words are redundant*.

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

  - For example, can you tell the difference?

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

2.4 USE PRONOUNCEABLE NAMES

- Make your names ***pronounceable***.

- A company I know has *genymdhms* (generation date, year, month, day, hour, minute, and second) so they walked around saying "gen why emm dee aich emm ess".
  - I have an annoying habit of pronouncing everything as written, so I started saying "gen-yah-muddahims."

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

vs.

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

2.5 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.
  - *MAX_CLASSES_PER_STUDENT* vs. the number 7
  - The name *e* is a poor choice for any variable.

- Single-letter names can *ONLY be used as local variables inside short methods*.

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

vs.

```
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;    realTaskDays
}
```

2.6 AVOID ENCODINGS

- ***Encoding type or scope information into names*** simply adds an extra burden of deciphering.
  - **Fortran** forced encodings by making the first letter a code for the type.
  - **Modern languages** have much richer type systems, and the compilers remember and enforce the types.

- You don't need to prefix member variables with *m_* anymore.

```
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;
  }
}
```

2.7 AVOID MENTAL MAPPING

- Readers shouldn't have to mentally ***translate your names into other names they already know***.
  - This problem generally arises from a choice to use neither *problem domain terms (2.14)* nor *solution domain terms (2.13)*.

- Many problems arise with single-letter variable names.
  - A loop counter may be named *i* or *j* or *k*, only if its scope is very small and no other names can conflict with it.
  - There can be no worse reason for using the name *c* than because *a* and *b* were already taken.

KU KONKUK UNIVERSITY

2.8 CLASS NAMES

- ***Classes*** and ***objects*** should have ***noun*** or ***noun phrase*** names.
  - Such as *Customer*, *WikiPage*, *Account*, and *AddressParser*.

- Avoid obscure and common words.
  - Such as *Manager*, *Processor*, *Data*, or *Info*.

- A class name should not be a verb.

2.9 METHOD NAMES

- ***Methods*** should have ***verb*** or ***verb phrase*** names.
  - Such as *postPayment*, *deletePage*, or *save*.

- Accessors, mutators, and predicates should be named for their value and prefixed with *get* or *set*.

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

2.10 DON'T BE CUTE

- If names are ***too clever***, they will be memorable only to people who share the author's sense of humor.

- Cuteness in code often appears in the form of *colloquialisms* or *slang*.
  - Don't use the name *whack()* to mean *kill()*.
  - Don't tell little culture-dependent jokes like *eatMyShorts()* to mean *abort()*.

2.11 PICK ONE WORD PER CONCEPT

- ***Pick one word for one abstract concept.***
  - For instance, it's confusing to have *fetch*, *retrieve*, and *get* as equivalent methods of different classes.

- It's confusing to have a *controller*, a *manager* and a *driver* in the same code base.
  - What is the essential difference between a *DeviceManager* and a *ProtocolController*? Why are both not controllers or both not managers? Are they both Drivers really?

- ***Consistent lexicon*** is a great boon to the programmers who must use your code.

2.12 DON'T PUN

- Avoid using *the same word* for *two purposes*.

- Using the same term for two different ideas is essentially a **pun**.
  - *add* vs. *insert* vs. *append*

2.13 USE SOLUTION DOMAIN NAMES

- Remember that the people who read your code will be ***programmers***.

- Go ahead and ***use computer science (CS) terms***.
  - Such as algorithm names, pattern names, math terms, and so forth.

- It is not wise to draw every name from the ***problem domain*** because
  - We don't want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name.

2.14 USE PROBLEM DOMAIN NAMES

- When there is no "programmer-eese" for what you're doing, use the name from the ***problem domain***.
  - At least the programmer who maintains your code can ask a domain expert what it means.

- "***Separating solution and problem domain concepts***" is part of the job of a good programmer and designer.

2.15 ADD MEANINGFUL CONTEXT

- You need to ***place names in context for your reader*** **by enclosing them in** *well-named classes, functions, or namespaces*.
  - When all else fails, then prefixing the name may be the last resort.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```

vs.

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

2.16 DON'T ADD GRATUITOUS CONTEXT

- In an imaginary application called "*Gas Station Deluxe*," it is a bad idea to prefix every class with GSD.
  - Frankly, you are working against your tools.
  - Because whenever you type G and press the completion key, then you are rewarded with a mile-long list of every class in the system.

- ***Shorter names*** are generally better than longer ones, so long as they are clear.

- *Add no more context to a name than is necessary.*

FINAL WORDS

- The hardest thing about ***choosing good names*** is that it requires ***good descriptive skills*** and a ***shared cultural background***.
  - This is a ***teaching issue*** rather than a technical, business, or management issue.
  - As a result, many people in this field don't learn to do it very well.

- People are afraid of *renaming* things for fear that some other developers will object.
  - We find that they will be grateful when names change for the better.
    - You will probably end up surprising someone when you rename, just like you might with any other code improvement.

- Follow some of these rules and see whether you don't ***improve the readability*** of your code.

Chapter 3. Functions

3. Functions

- ***Functions*** are the first line of organization in any program.
- *Writing functions well* is the topic of this chapter.

- See how much you can understand it in the next 3 minutes.

```
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");
    }
  }
  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();
}
```

Listing 3-1 HtmlUtil.java (FitNesse 20070619)

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

- However, with just a few simple ***method extractions***, some ***renaming***, and a little ***restructuring***, we can capture the intent of the function ***in the 9 lines***.

```
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();
}
```

Listing 3-2 HtmlUtil.java (refactored)

3.1 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***.
  - Lines should not be **150** characters long.
  - Functions should not be **100** lines long.
    - Functions should hardly ever be **20** lines long.

- *How short should your function be?*
  - They should usually be shortened to the below:

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

    - The blocks within *if* statements, *else* statements, *while* statements, and so on should be one line long.
  - Functions should **not** be large enough to hold nested structures.

3.2 DO ONE THING

- The following advice has appeared for *30 years or more*.
  - "***FUNCTIONS SHOULD DO ONE THING.***
    THEY SHOULD DO IT WELL.
    THEY SHOULD DO IT ONLY. "
- The problem is that it is hard to know **what "*one thing*" is**.
  - If a function does *only those steps that are one level below* the stated name of the function, then the function is doing one thing.
  - A function is doing more than "one thing", *if you can extract another function from it* with a name that is not merely a restatement of its implementation.

- ***Sections within functions*** is an obvious symptom of doing more than one thing.
  - Notice that the *generatePrimes* function is divided into sections such as ***declarations***, ***initializations***, and ***sieve***.
  - *Functions that do one thing cannot be reasonably divided into sections.*

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria.  The first man to calculate the
 * circumference of the Earth.  Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple.  Given an array of integers
 * starting at 2.  Cross out all multiples of 2.  Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat untilyou have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
import java.util.*;

public class GeneratePrimes
{
  /**
   * @param maxValue is the generation limit.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      boolean[] f = new boolean[s];
      int i;

      // initialize array to true.
      for (i = 0; i < s; i++)
        f[i] = true;
```

```
      // get rid of known non-primes
      f[0] = f[1] = false;

      // sieve
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // if i is uncrossed, cross its multiples.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // multiple is not prime
        }
      }

      // how many primes are there?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // bump count.
      }

      int[] primes = new int[count];

      // move the primes into the result
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i])             // if prime
          primes[j++] = i;
      }

      return primes;  // return the primes
    }
    else // maxValue < 2
      return new int[0]; // return null array if bad input.
  }
}
```

3.3 ONE LEVEL OF ABSTRACTION PER FUNCTION

- The **statements** within our function should be *all at the same level of abstraction*.

- For example, Listing 3-1 violates this rule.
  - At a very high level of abstraction, such as
    - *getHtml();*
  - At an intermediate level of abstraction, such as:
    - *String pagePathName = PathParser.render(pagePath);*
  - Remarkably at a low level, such as:
    - *.append("\n")*

- ***Mixing levels of abstraction within a function*** is always confusing.
  - Once details are mixed with essential concepts, more and more details tend to accrete within the function.

```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");
    }
  }
  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();
}
```

Listing 3-1 HtmlUtil.java (FitNesse 20070619)

- ***The Stepdown Rule*** : *Reading code from top to bottom*
  - Read the code like a *top-down narrative*.
    - *Every function are 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.
  - It looks like a set of *TO* paragraphs :
    - *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…*

- It is the key to ***keeping functions short*** and ***making sure they do "one thing."***

3.4 SWITCH STATEMENTS

- It's hard to make a ***small switch statement***.
  - By their nature, switch statements always do N things.

- But we can make sure that each switch statement is *buried in a low-level class* and is never repeated *with polymorphism*. *→ But I don't agree that.*

- The code shows just one of the operations that might depend on the type of *Employee*.

```
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 InvalidEmployeeType(e.type);
    }
  }
```

- There are **several problems** with this function.
  - When new employee types are added, it will grow.
  - It very clearly does more than one thing.
    - There are an unlimited number of other functions that will have the same structure.
  - It violates the *Single Responsibility Principle (SRP)* because there is more than one reason for it to change.
  - It violates the *Open Closed Principle (OCP)* because it must change whenever new types are added.

- My general rule for switch statements is
  - They 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.

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
-----------------
public interface EmployeeFactory {
  public 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 SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

Factory Method

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
-----------------
public interface EmployeeFactory {
  public 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 SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

Provides an abstract interface for creating one product.

EmployeeFactory

PizzaStore

createPizza()

PizzaStore is implemented as a Factory Method because we want to be able to create a product that varies by region. With the Factory Method, each region gets its own concrete factory that knows how to make pizzas that are appropriate for the area.

Each subclass decides which concrete class to instantiate.

EmployeeFactoryImpl

New York Store

+ makeEmployee( )

NYPizzaStore

createPizza()

ChicagoPizzaStore

createPizza()

Chicago Store

The Factory Method

The Factory Method

The NYPizzaStore subclass only instantiates NY style pizzas.

This is the product of the PizzaStore. Clients only rely on this abstract type.

The ChicagoPizzaStore subclass instantiates only Chicago style pizzas.

Employee

+ isPayday( )

+ calculatePay( )

+ deliverPay( )

Pizza

CommissionedEmployee

HourlyEmployee

SalariedEmployee

NYStyleCheesePizza

NYStylePepperoniPizza

NYStyleClamPizza

NYStyleVeggiePizza

Subclasses are instantiated by the Factory Methods.

New York

Chicago

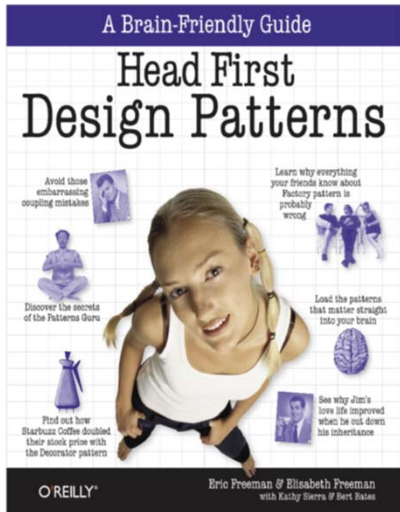
ChicagoStyleCheesePizza

ChicagoStylePepperoniPizza

ChicagoStyleClamPizza

ChicagoStyleVeggiePizza

The createPizza() method is parameterized by pizza type, so we can return many types of pizza products.



3.5 USE DESCRIPTIVE NAMES

- **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 smaller* and more focused a function is, the easier it is to choose a descriptive name.

- **Choosing descriptive names** will clarify the design of the module in your mind and help you to improve it.
  - A long descriptive name is better than a short enigmatic name.
  - A long descriptive name is better than a long descriptive comment.

- ***Be consistent in your names.***
  - Use the same phrases, nouns, and verbs in the function names you choose for your modules.
  - For example,
    - For he names *includeSetupAndTeardownPages*, *includeSetupPages*, *includeSuiteSetupPage*, and *includeSetupPage,* then you'd ask yourself: "What happened to *includeTeardownPages*, *includeSuiteTeardownPage*, and *includeTeardownPage*?"

3.6 FUNCTION ARGUMENTS

- The 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 shouldn't be used anyway.

- Arguments are hard.
  - Our readers would have had to *interpret the argument* each time they saw it.
  - *Testing* every combination of appropriate values can be daunting.
  - *Output arguments are harder to understand than input arguments.*
    - We don't usually expect information to be going out through the arguments.

- ***One input argument** is the next best thing to **no arguments**.*

3.6.1 Common Monadic Forms

- There are **two very common cases** to pass **a single argument** into a function.
  1. ***Asking*** a question about that argument, as in *boolean fileExists("MyFile")*.
  2. Operating on that argument, ***transforming*** it into something else and returning it.
     - For example, *InputStream fileOpen("MyFile")* transforms a file name *String* into an *InputStream* return value.

- A somewhat less common is an ***event***.
  - There is *an input argument but no output argument*.
    - *void passwordAttemptFailedNtimes(int attempts)*
  - It should be very clear to the reader that this is an event.

- Try to avoid any monadic functions that don't follow these (three) forms.

KU KONKUK UNIVERSITY

3.6.2 Flag Arguments

- ***Flag arguments*** are ugly.
  - *Passing a boolean into a function* is a truly terrible practice.
  - It immediately 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!

- For example, the method call *render(true)* is just plain confusing to a poor reader.

```
private String render(boolean isSuite) throws Exception {
  this.isSuite = isSuite;
  if (isTestPage())
    includeSetupAndTeardownPages();
  return pageData.getHtml();
}
```

  - We should have split the function into two: *renderForSuite()* and *renderForSingleTest()*.

KU KONKUK UNIVERSITY

3.6.3 Dyadic Functions

- A 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)*.
- Even obvious dyadic functions like *assertEquals(expected, actual)* are problematic.
  - How many times have you put the *actual* where the *expected* should be?
  - The two arguments have no natural ordering. But the *expected*, *actual* ordering is a convention that requires practice to learn.

- However, you should be aware that dyads comes at a cost and should take advantage of other mechanisms available to you to convert them into monads.

3.6.4 Triads

- Functions that take **three arguments** are *significantly harder to understand* than dyads.
  - The issues of ordering, pausing, and ignoring are more than doubled.

- I suggest you think very carefully before creating a triad.

- For example, 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*?

3.6.5 Argument Objects

- When 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,

```
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.
  - *When groups of variables are passed together, they are likely part of a concept that deserves a name of its own*.

KU KONKUK UNIVERSITY

3.6.6 Argument Lists

- Sometimes we want to pass *a variable number of arguments* into a function.

- For example,

```
String.format("%s worked %.2f hours.", name, hours);
```

  - If the variable arguments are all treated identically, as they are in the example above, then they are equivalent to a single argument of type ***List***.

  - By that reasoning, *String.format* is actually dyadic.

```
public String format(String format, Object... args)
```

- Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

3.6.7 Verbs and Keywords

- 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.

- In the case of a monad, the function and argument should form a very nice ***verb/noun*** *pair*.
  - For example, *write(name)* is very evocative. Whatever this "*name*" thing is, it is being "*written*."
  - An even better name might be *writeField(name)*, which tells us that the "*name*" thing is a "field."

- With the ***keyword* form** of a function name, we encode the names of the arguments into the function name.
  - For example, *assertEquals* might be better written as *assertExpectedEqualsActual(expected, actual)*.
    - This strongly mitigates the problem of having to remember the ordering of the arguments.

3.7 HAVE NO SIDE EFFECTs

- ***Side effects*** are lies.
  - Your function promises to do one thing, but it also does other **hidden things**.
    - Sometimes it will make unexpected changes to the variables of its own class.
    - Sometimes it will make them to the parameters passed into the function or to system globals.
  - Often result in *strange temporal couplings* and *order dependencies*.

- For example,
  - This function uses a standard algorithm to match a *userName* to a *password*.
  - But it also has a side effect. It calls to *Session.initialize()*.

```
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;
  }
}
```

3.7.1 Output Arguments

- Arguments are most naturally interpreted as inputs to a function.
  - For example, is *s* an input or an output?

```
appendFooter(s);
```

    - Does this function append *s* as the footer to something? Or does it append some footer to *s*?
  - What about the signature of the function?

```
public void appendFooter(StringBuffer report)
```

- In the days before object-oriented programming, it was sometimes necessary to have output arguments.
  - However, in OO languages, ***this*** is intended to act as an output argument.
  - In other words, it would be better for *appendFooter* to be invoked as ***report.appendFooter()***;
- **In general, output arguments should be avoided.**
  - If your function must change the state of something, *let it change the state of its owning object*.

3.8 COMMAND QUERY SPEARATION

- Functions should either ***do something*** or ***answer something***, but **not both**.
  - Command vs. Query : doing both often leads to confusion.
    - Either your function should change the state of an object, or it should return some information about that object.

- For example,

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

  - What does it mean?
    - Asking whether the "*username*" attribute was previously set to "*unclebob*"? → *Query*
    - Asking whether the "*username*" attribute was successfully set to "*unclebob*"? → *Command & Query*

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

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

3.9 PREFER EXCETIONS TO RETURNING ERROR CODES

- ***Returning error codes*** *from command functions* is 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)*

```
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;
}
```

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

3.9.1 Extract Try/Catch Blocks

- **But *Try/catch* blocks are ugly in their own right.**
  - They confuse the structure of the code.
  - They **mix** *error processing* with *normal processing*.

- It is better to extract the bodies of the *try/catch* blocks out into functions of their own.
  - The *delete* function is all about error processing.
  - The *deletePageAndAllReferences* function is all about the processes of fully deleting a page.

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

```
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());
}
```

3.9.2 Error Handling Is One Thing

- **Functions** should do one thing and **error handling** is one thing.
- *A function that handles errors should do nothing else.*
  - If the keyword *try* exists in a function, it should be the very first word in the function and that there should be nothing after the *catch/finally* 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());
}
```

3.9.3 The Error.java Dependency Magnet

- Returning error codes usually implies that there is *some class* or *enum* in which all the ***error codes are defined***.

```
public enum Error {
   OK,
   INVALID,
   NO_SUCH,
   LOCKED,
   OUT_OF_RESOURCES,
   WAITING_FOR_EVENT;
}
```

- Classes like this are a ***dependency magnet***; many other classes must import and use them.
  - *When the Error enum changes, all those other classes need to be recompiled and redeployed*.
  - Programmers don't want to add new errors because then they have to rebuild and redeploy everything. So, they reuse old error codes instead of adding new ones.

- When you use ***exceptions***, then new exceptions are *derivatives* of the exception class.
  - They can be added without forcing any recompilation or redeployment
  - This is an example of the **Open Closed Principle (OCP)**.

3.10 DON'T REPEAT YOURSELF

- ***Duplication*** may be the root of all evil in software.

- Many principles and practices have been created for the purpose of controlling or eliminating it. For example,
  - All of Codd's database normal forms serve to eliminate duplication in data.
  - Consider also how object-oriented programming serves to concentrate code into base classes that would otherwise be redundant.
  - Structured programming, Aspect Oriented Programming, Component Oriented Programming, are all, in part, strategies for eliminating duplication.

- Since the invention of the ***subroutine***, innovations in software development have been an ongoing attempt to eliminate duplication from our source code.

- Listing 3-1 vs. Listing 3-7

```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");
    }
  }
  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();
}
```

Listing 3-1 HtmlUtil.java (FitNesse 20070619)

```java
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

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;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }
```

1

```java
  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }

  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();
  }

  private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
  }

  private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
  }

  private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());
  }

  private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
      includeSuiteTeardownPage();
  }

  private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
  }

  private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
  }
```

2

```java
  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
  }

  private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName = getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }

  private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
  }

  private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
      .append("\n!include ")
      .append(arg)
      .append(" .")
      .append(pagePathName)
      .append("\n");
  }
}
```

3

Listing 3-7 SetupTeardownIncluder.java

1

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

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;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }
```

2

```
  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception
    includeSetupPages(); (1)
    includePageContent(); (2)
    includeTeardownPages(); (3)
    updatePageContent(); (4)
  }

  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();                    (1)
  }

  private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
  }
      (1-1)
  private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
  }

  private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());      (2)
  }

  private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
      includeSuiteTeardownPage();                 (3)
  }

  private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
  }
      (3-1)
  private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
  }
```


Listing 3-7 SetupTeardownIncluder.java

2

```
private boolean isTestPage() throws Exception {
  return pageData.hasAttribute("Test");
}

private void includeSetupAndTeardownPages() throws Exception
  includeSetupPages(); (1)
  includePageContent(); (2)
  includeTeardownPages(); (3)
  updatePageContent(); (4)
}

private void includeSetupPages() throws Exception {
  if (isSuite)
    includeSuiteSetupPage();
  includeSetupPage();                      (1)
}

private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}    (1-1)

private void includeSetupPage() throws Exception {
  include("SetUp", "-setup");
}

private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());       (2)
}

private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();                       (3)
}

private void includeTeardownPage() throws Exception {
  include("TearDown", "-teardown");
}    (3-1)

private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}
```

3

```
  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());        (4)
  }

  private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName = getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }                                                  (1-1) (3-1)

  private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
  }

  private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
      .append("\n!include ")
      .append(arg)
      .append(" .")
      .append(pagePathName)
      .append("\n");
  }
}
```

3.11 STRUCTURED PROGRAMMING

- Some programmers follow ***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 in a function, no *break* or *continue* statements in a loop, and never, ever, any *goto* statements.

- It is only in larger functions that such rules provide significant benefit.
  - Those rules serve *little benefit* when functions are *very small.*

- *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.
  - As *goto* only makes sense in large functions, it should be avoided.

HOW DO YOU WRITE FUNCTIONS LIKE THIS?

- ***Writing software is like any other kind of writing.***
- When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well.
  - The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.
- 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.
  - But I also have a suite of *unit tests* that cover every one of those clumsy lines of code.
- 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.

CONCLUSION

- *The art of programming is the art of language design*.
  - Every system is built from a domain-specific language designed by the programmers to describe that system.
  - Functions are the verbs of that language, and classes are the nouns.
- 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 language that can be used to tell that story.
- This chapter has been about the mechanics of ***writing functions well***.
  - If you follow the rules herein, your functions will be *short*, *well named*, and *nicely organized*.
- But never forget that your real goal is to *tell the story of the system*, and that the functions you write *need to fit cleanly together into a clear and precise language* to help you with that telling.