Clean Code - A Handbook of Agile Software Craftsmanship

> Notes taken by Mehmet Ozan Güven (10.10.2019 - 09.12.2019)

Content

1. What Clean Code does mean?

- Elegant(zarif) and efficient code
- Make it hard for bugs to hide
- Pleasing(memnuniyet verici) to read
- Clean code does one thing
- Reading the clean code should make you smile the way well-designed car would
- Clean code shows us the problem that is trying to solve
- Code without tests is not clean, not matter how elegant it is, not matter how readable and accessible, if it has not tests, it be unclean
- Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better
- Beautiful code makes the language look like it was made for the problem.
- Bad code **tempts(özendirmek)** the mess to grow.

2. Meaningful Names

1. Use Intention-Revealing Names

- Names should reveal intent.
- Choosing good names takes time but saves more than it takes.

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

- The name d reveals nothing. It does not evoke a sense of elapsed time nor of days
- 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 can make it much easier to understand and change code.

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

- Why is it hard to tell what this code is doing? There are no complex expressions. Spacing and indentation are reasonable and there are only 3 variables and 2 constants. They are just a list of arrays
- The problem isn't the simplicity of the code but the implicity(in a way that is suggested but not communicated directly)
- The code implicitly requires that we know the answers to questions such as:
    - What kinds of things are in theList ?
    - What is the significance of the 0th subscript of an item in theList ?
    - What is the significance of the value 4 ?
    - How would I use the list being returned

2. Avoid Disinformation

- Programmers must avoid leaving false clues that obscure the meaning of code.
- For example let's say we have a class namely `XYZControllerForEfficientHandlingOfStrings`, after a while we created another class namely `XYZControllerForEfficientStorageOfStrings` . The other developers may confuse to use the correct class because of the similarity between these classes

Make Meaningful Distinctions

- Don' use such series: `(a1, a2, .. aN)` . They provide no clue to the author's intention

```
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 we have a `Product` class. If we have another called `ProductInfo` or `ProductData` we have made the names different without making them mean anything different. `Info` and `Data` are indistinct noise words like `a, an, the`
- Distinguish names in such a way that the reader known what the differences offer.

Use Pronounceable Names

- We should make our names pronounceable.
- Bad example: `genymdhms` which means that (generation date, year, month, day, hour, minute and second)

```
class DtaRcrd102{
    private Date genymdhsm;
    private Date modymdhsm;
    private final String psqint "102";
}
// convert to
class Customer{
    private Date generationTimestamp;
    private Date modificationTimestamp;
    private final String recordId = "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?”

Use Searchable Names

- Single-letter names can only be used as local variables inside short methods (personal advice). The length of a name should correspond to the size of its scope. If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name.

Avoid Encodings

- Don't use encoding for names

Member prefixes

- Don't need to prefix member variables with `m_` anymore.
- Our classes and functions should be small enough that we don't need them.

```
public class Part {
    private String m_dsc;

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

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

Interfaces and Implementations

- Prefix `I` is so common, but it gives many information to the users.
- We don't want users knowing that I am handing them an interface.
- For example, instead of `IShapeFactory & ShapeFactory` use `ShapeFactory & ShapeFactoryImpl`

Class Names

- Classes and objects should have noun or noun phrase names like `Customer, WikiPage, Account, AddressParser`. Avoid words like `Manager, Processor, Data, Info` in the name of a class. A class name should not be a verb.

Method Names

- Methods should have verb or verb phrase names like `postPayment, deletePage, save`
- Accessors, mutators, and predicates should be named for their value and prefixed with `get, set, is` according to the javabean standard.

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

- When constructors are overloaded, use static factory methods with names that describe the arguments. For example:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
// is generally better than
Complext fulcrumPoint = new Complex(23.0);
```

- Consider enforcing their use by making the corresponding constructors private.

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. Choose one of them for all.
- Likewise, 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 `ProtocolController` Why are both not `controller`s or both not `manager`s. The name leads us to expect 2 objects that have very different type as well as having different classes.

Don't Pun

- Avoid using the same word for 2 purposes. Using the same term for 2 different ideas is essentially a pun.

Add Meaningful Context

- If we follow the "one word per concept" rule, we could end up with many classes that have, for example an `add` method. As long as the parameter lists and return values of the various `add` methods are semantically equivalent, all i s well.
- However one might decide to use the word `add` for "consistency" when he or she is not in fact adding in the same sense. For example let's say we have many classes where `add` will create a new value by adding or concatenating 2 existing values. Now let's say we are writing a new class that has a method that puts its single parameter into a collection. Should we call this method `add`? It might seem consistent because we have so many other `add` methods, but in this case, the semantics are different, so we should use a name like `insert` or `append` instead

Use Solution Domain Names

- Use Computer Science terms, algorithm names, pattern names and so forth. Because the people who read our code will be programmers.
- Every programmer can understand for example `AccountVisitor` deals with Visitor pattern

Use Problem Domain Names

- When there is no programmer-ease, use the name from the problem domain.

Add meaningful Context

- We said that use well-named classes functions or namespaces. When all ese fails, then prefixing the name may be necessary as a last resort.
- Imagine that we have variables named `firstName, lastName, street, houseNumber, city, state and zipcode`. Taken together it's pretty clear that they form an address.

- But what if we just saw the `state` variable being used alone in a method. Would we automatically infer that it was part of an address?
- We can add context by sing prefixes: `addrFirstname, addrLastName, addrState` and so on. At least readers will understand that these variables are part of a larger structure.
- Consider the method in the below. When we first look at the method, the meanings of the variables are opaque.

```
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);
}
```

- The function is a bit too long and the variables are used throughout. To split the function into smaller pieces we need to create a `GuessStaticticsMessage` class and make the three variables fields of this class. This provides a clear context for the three variables. They are definitively part of the `GuessStaticticsMessage`

```
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";
    }
}
```

Don't Add Gratuitous(Unnecessary) Context

- In an imaginary application called "Gas Station Deluxe", it is a bad idea to prefix every class with `GSD`
- Likewise, say we invented a `MailingAddress` class in `GSD`'s accounting module, and we named it `GSDAccountAddress`. Later, we need a mailing address for our customer contact application. Do we use `GSDAcoountAddress` ?
- Shorter names are generally better than longer ones.
- Add no more context to a name than is necessary

3. Functions

- **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

Here is the example:

```
// code 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);
    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");
    }
        // ... goes on
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
}
```

- Here is the refactored view:

```
// code 2
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();
}
```

- Here is the re-refactored view:

```
// code 3
public static String renderPageWithSetupsAndTeardowns(PageData pageData, boolean
isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

Blocks and Indenting

- The above code implies that the blocks within `if, else, while, for ...` statements and so on should be one line long. Probably **that line should be a function call**.
- Not only does this keep the enclosing function small, but is also adds documentary value because the function called within the block can have a nicely descriptive name.
- **This also implies that functions should be larger 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

- It should be very clear that code-1 is doing lots more than one thing. It's creating buffers, fetching pages, searching for inherited pages, rendering paths.
- On the other hand, code-3 is doing one simple thing. It's including setups and teardowns into test pages.
- **Functions should do one thing. They should do it well. They should do it only**
    - The problem with this statement is that it is hard to know what **one thing** is.
    - Does code-3 do one thing? It's easy to make the case that it's doing three things:
        - Determining whether the page is a test page
        - If so, including setups and teardowns.
        - Rendering the page in HTML
    - Notice that the three steps of the function are one level of abstraction below the stated name of the function. We can describe the function by describing it as a brief paragraph:
        - TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML
- So, another way to know that a function is doing more than "one thing" is if we can extract another function from it with a name that merely(only) a restatement of its implementation.

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 functions are all at the same level of abstraction.
- Code-1 violates this rule. There are concepts in there that are at a very high level of abstraction, such as `getHtml()` others that are at an intermediate level of abstractions such as `String pagePathName = PathParser.render(pagePath)` and still others that are remarkably low level, such as `.append("\n")`
- Mixing levels of abstraction within a function is always confusing.

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.
- We call this **The Stepdown Rule**
- In other words, we want to be able to read the program as though it were a set of **TO** paragraphs, each of which is describing the current level of abstraction and referencing subsequent **TO** paragraphs at the next level down.

```
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 very difficult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do "one thing"
- Here is the example:

```
package fitnesse.html;
import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;
// code 4
public class SetupTeardownIncluder{
    private PageData pageData;
    private boolean isSuite;
    private WikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;

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

    public static String render(PageData pageDate, 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(); // To include setup and teardown
        return pageData.getHtml();
    }

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

```
    }

    private void includeSetupAndTeardownPages() throws Exception {
        includeSetupPages(); // 1. we include setup
        includePageContent(); // 6. then we include test page content
        includeTeardownPages(); // 8. and then we include teardown page
        updatePageContent(); // 12.
    }

    // 2.
    private void includeSetupPages() throws Exception {
        if (isSuite)
            includeSuiteSetupPage();
        includeSetupPage();
    }

    // 3.
    private void includeSuiteSetupPage() throws Exception {
        include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
    }

    // 4.
    private void includeSetupPage() throws Exception {
        include("SetUp", "-setup");
    }

    // 7.
    private void includePageContent() throws Exception {
        newPageContent.append(pageData.getContent());
    }

    // 8.
    private void includeTeardownPages() throws Exception {
        includeTeardownPage(); // 9.
        if (isSuite)
            includeSuiteTeardownPage(); // 10.
    }

    // 9.
    private void includeTeardownPage() throws Exception {
        include("TearDown", "-teardown");
    }

    // 10.
    private void includeSuiteTeardownPage() throws Exception {
        include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
    }

    // 12.
    private void updatePageContent() throws Exception {
        pageData.setContent(newPageContent.toString());
    }

    // 5. , 11.
    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");
    }
}
```

Switch Statements

- It's hard to make a small switch statement.
- It's also hard to make a `switch` statement that does one thing. By their nature, `switch` statement always do N things.
- Unfortunately we can't always avoid switch statements, but we can make sure that each switch statement is buried in a low-level class and is never repeated. We do this with polymorphism.
- Consider the function below

```
// code 5
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 functions.
    - First, it's large and when new employee types are added, it will grow.
    - Second, it very clearly does more than one thing.
    - Third, it violates the Single Responsibility Principle because there is more than one reason for it to change.

    - Fourth, it violates the Open Closed Principle because it must change whenever new types are added.
    - But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure. For example, we could have:
    - `isPayday(Employee e, Date date) or deliverDay(Employee e, Money pay)` etc.. All of which would have the same deleterious(harmful) structure.
- The solution to this problem is to bury the switch statement in the basement of an ABSTRACT FACTORY and never let anyone see it.
- The factory will use the `switch` statement to create appropriate instances of the derivatives of `Employee` and the various functions, such as `calculatePay, isPayDay and deliverPay` will be dispatched polymorphically through the `Employee` interface.
- General advice for switch statement is that they can be tolerated if they appear only once, are used to create polymorphic objects and are hidden behind an inheritance.

```
// code 6
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);
        }
    }
}
```

Use Descriptive Names

- We changed to example function name from `testableHtml` to `SetupTeardownIncluder.render`. This is a far better name because it better described what the function does.
- It's 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
```

- **Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name.**
- **A long descriptive name is better than a long descriptive comment**
- **Use a naming convention that allows multiple words to be easily read in the function names**
- **and then make use of those multiple words to give the function a name that says what it does**
- **Don't be afraid to spend time choosing a name.**
- **Be consistent in 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 , and includeSetupPage` .

Function Arguments

- The ideal number of arguments for a function is zero
- Next comes one
- Followed closely by two
- Three arguments should be avoided where possible.
- More than three requires very special justification
- Sometimes one input argument is the next bes thing to no arguments. `SetupTeardownIncluder.render(pageDate)` is pretty easy to understand. Clearly we are going to render the data in the `pageData` object.
- Don't follow this structure: (let's say we are going to transform transform string buffer argument to smth useful)

```
void transform-(StringBuffer out);
```

- Follow this:

```
StringBuffer transform(StringBuffer in);
```

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.
- In `code 4` we had no choice because the callers were already passing that flag in the `render(boolean isSuite)`. We should have split the function into two: `renderForSuite()` and `renderforSingleTest()`

Verbs and Keywords

- In the case of a monad(function with one argument), 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

Have No Side Effects

- Side effects are lies. Our function promises to do one thing, but it also does other "hidden" things.
- Consider the example:

```
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();
            if ("ValidPassword".equals(phase)){
                Session.initialize();
                return true;
            }
        }
        return false;
    }
}
```

- It may seem that "this function does one thing". However, it also calls to `Session.initialize()` .
- This side effect creates a temporal coupling. That is, `checkPassword()` can only by called at certain times (in other words, when it is safe to initialize the session). If it is called out of order, session data may be inadvertently lost.
- If we must have a temporal coupling, we 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"

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());
}
```

- In the above, the `delete` function is all about error processing. It is easy to understand and then ignore.
- The `deletePageAndAllReferences` function is all about the processes of fully deleting a `page`. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.

Error Handling is one thing

- Functions should do one thing. Error handling is one thing.
- Thus, a function that handles errors should do nothing else.
- This implies that if the keyword `try` exists in a function, it should be very first word in the function and that there should be nothing after the `catch/finally` blocks.

Conclusion

- 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**
- The art of programming is, and has always been, the art of language design.

4. Comments

- When we find ourselves in a position where we need to write a comment, think it through and see whether there isn't some way to turn the tables and express ourselves in code.

Explain Yourself in Code

- Which one is better ?

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

}
// or this?
if (employee.isEligibleForFullBenefits())
```

Informative Comments

- It is sometimes useful to provide basic information with a comment. For example:

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

- Even if this comment is a good, function could be named it such as `responderBeingTested`
- Here's the another example:

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

Clarification

- It is just helpful to translate the meaning of some obscure argument or return value into smth that's readable.
- For example:

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
    WikiPagePath ba = PathParser.parse("PageB.PageA");
    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
}
```

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

- It is sometimes reasonable to leave "To do" notes in the form of `//TODO` comments.
- `TODO`s are jobs that the programmer thinks should be done, but for some reason can't do at the moment

```
//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;
}
```

Mumbling (Mırıldanma)

- If we decide to write a comment, then spend the time necessary to make sure it is the best comment we can write.
- For example:

```
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
    }
}
```

- What does that comment in the `catch` block mean? Clearly it means smth to the author, but the meaning does not come though all that well.
- Apparently, if we get an `IOException` it means that there was no properties file, and in that case all the defaults are loaded. But who loads all the defaults? Were they loaded before the call to `loadProperties.load` ?
    - Or did `loadProperties.load` catch the exception, load the defaults,and then pass the exception on for us to ignore?
    - Or did `loadProperties.load` load all the defaults before attempting to load the file?
    - Was the author trying to comfort himself about the fact that he was leaving the catch block empty?

Redundant Comments

- This comment probably takes longer to read than the code itself

```
// 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");
    }
}
```

- This comment is not more informative than the code. It does not justify the code, or provide intent or rationale.
- Here is the another redundant comment. These comments serve only to clutter and obscure the code:

```
public abstract class ContainerBase implements Container, Lifecycle, Pipeline,
MBeanRegistration, Serializable{
    /**
    * The processor delay for this component.
    */
    protected int backgroundProcessorDelay = -1;
    /**
    * The lifecycle event support for this component.
    */
    protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);
    /**
    * The container event listeners for this Container.
    */
    protected ArrayList listeners = new ArrayList();
    /**
    * The Loader implementation with which this Container is
    * associated.
    */
    protected Loader loader = null;
    /**
    * The Logger implementation with which this Container is
    * associated.
    */
    protected Log logger = null;
    /**
    * Associated logger name.
    */
    protected String logName = null;
    /**
    * The Manager implementation with which this Container is
    * associated.
    */
    protected Manager manager = null;
    /**
    * The cluster with which this Container is associated.
    */
    protected Cluster cluster = null;
    // ....
```

```
}
```

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.

```
/**
*
* @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 we see comments that are nothing but noise. They restate the obvious and provide no new information.

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

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

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

Don't Use a Comment When We can use a function or 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

- Few practices are as odious(iğrenç) as commenting-out code. Don't do this:

```
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()));
```

- 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.

Inobvious Connection

- The connection between a comment and the code it describes should be obvious.
- Example:

```
/*
* start with an array that is big enough to hold all the pixels
* (plus filter bytes), and an extra 200 bytes for header info
*/
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

- What is a filter byte? Does it relate to the `+1` or to the `*3` or both?
- Is a pixel a byte?
- Why 200?

Example

- Here is the bad code and comment:

```
/**
* 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.
    }
}
```

- Good one

```
/**
* This class Generates prime numbers up to a user specified
* maximum. The algorithm used is the Sieve of Eratosthenes.
* Given an array of integers starting at 2:
* Find the first uncrossed integer, and cross out all its
* multiples. Repeat until there are no more multiples
* in the array.
*/
public class PrimeGenerator{
    private static boolean[] crossedOut;
    private static int[] result;
    public static int[] generatePrimes(int maxValue){
        if (maxValue < 2)
            return new int[0];
        else
        {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }
    private static void uncrossIntegersUpTo(int maxValue){
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
            crossedOut[i] = false;
    }

    private static void crossOutMultiples(){
        int limit = determineIterationLimit();
        for (int i = 2; i <= limit; i++)
        if (notCrossed(i))
            crossOutMultiplesOf(i);
    }

    private static int determineIterationLimit(){
    // Every multiple in the array has a prime factor that
    // is less than or equal to the root of the array size,
    // so we don't have to cross out multiples of numbers
    // larger than that root.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
    }

    private static void crossOutMultiplesOf(int i){
    for (int multiple = 2*i;
    multiple < crossedOut.length;
    multiple += i)
        crossedOut[multiple] = true;
```

```
    }

    private static boolean notCrossed(int i){
        return crossedOut[i] == false;
    }

    private static void putUncrossedIntegersIntoResult(){
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            result[j++] = i;
    }

    private static int numberOfUncrossedIntegers(){
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            count++;
    return count;
    }
}
```

5. Formatting

- Take care your code is nicely formatted
- Choose a set of simple rules that govern the format of your code. Then consistently apply those rules

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 FileInoutStream(getPreferencesFile());
        setPreferences(new Properties(getPreferences()));
        getPreferences().load(is);
    }catch (IOException e){
        try{
            if (is != null)
                is.close();
        }catch{

        }
    }
}
```

- Control variables for loops should be usually declared withing the loop statement.

```
public int countTestCases(){
    int count = 0;
    for (Test each : tests)
        count += each.countTestCases();
    return count;
}
```

- **Instance variables**, should be declared at the top of the class.

Dependent Functions

- If one function calls another, they should be vertically close, and the caller should be above the callee. This gives the program a natural flow.
- Here is the good example:

```
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);
        if (page != null)
            pageData = page.getData();
    }

    private Response notFoundResponse(FitNesseContext context, Request
request)throws Exception {
        return new NotFoundResponder().makeResponse(context, request);
```

```
    }

    private SimpleResponse makePageResponse(FitNesseContext context)throws
Exception {
        pageTitle = PathParser.render(crawler.getFullPath(page));
        String html = makeHtml(context);
        SimpleResponse response = new SimpleResponse();
        response.setMaxAge(0);
        response.setContent(html);
        return response;
    }
```

- We expect the most important concepts to come first and we expect them to be expressed with the least amount of polluting detail. We expect the low-level details to come last.

Horizontal Alignment

- Don't need to do line up all the variable names in a set of declarations. For example:

```
public class FitNesseExpediter implements ResponseSender
{
    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;

public FitNesseExpediter(Sockets,FitNesseContext context) throws Exception
{
    this.context =              context;
    socket =                    s;
    input =                     s.getInputStream();
    output =                    s.getOutputStream();
    requestParsingTimeLimit =   10000;
}
```

- This alignment seems to emphasize the wrong things and lead our eye away from the true indent. We are tempted to read down the list of variables names without looking at their types.
- So you should write like this:

```
public class FitNesseExpediter implements ResponseSender
{
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
```

```
    // ...
public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
{
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
}
```

Uncle Bob Code formatting example

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer() {
        lineWidthHistogram = new LineWidthHistogram();
    }

    public static List<File> findJavaFiles(File parentDirectory) {
        List<File> files = new ArrayList<File>();
        findJavaFiles(parentDirectory, files);
        return files;
    }

    private static void findJavaFiles(File parentDirectory, List<File> files) {
        for (File file : parentDirectory.listFiles()) {
            if (file.getName().endsWith(".java"))
                files.add(file);
            else if (file.isDirectory())
                findJavaFiles(file, files);
            }
    }

    public void analyzeFile(File javaFile) throws Exception {
        BufferedReader br = new BufferedReader(new FileReader(javaFile));
        String line;
        while ((line = br.readLine()) != null)
            measureLine(line);
    }

    private void measureLine(String line) {
        lineCount++;
        int lineSize = line.length();
        totalChars += lineSize;
        lineWidthHistogram.addLine(lineSize, lineCount);
        recordWidestLine(lineSize);
    }

    private void recordWidestLine(int lineSize) {
        if (lineSize > maxLineWidth) {
```

```
            maxLineWidth = lineSize;
            widestLineNumber = lineCount;
        }
    }

    public int getLineCount() {
        return lineCount;
    }

    public int getMaxLineWidth() {
        return maxLineWidth;
    }

    public int getWidestLineNumber() {
        return widestLineNumber;
    }

    public LineWidthHistogram getLineWidthHistogram() {
        return lineWidthHistogram;
    }

    public double getMeanLineWidth() {
        return (double)totalChars/lineCount;
    }

    public int getMedianLineWidth() {
        Integer[] sortedWidths = getSortedWidths();
        int cumulativeLineCount = 0;
        for (int width : sortedWidths) {
            cumulativeLineCount += lineCountForWidth(width);
            if (cumulativeLineCount > lineCount/2)
            return width;
        }
        throw new Error("Cannot get here");
    }

    private int lineCountForWidth(int width) {
        return lineWidthHistogram.getLinesforWidth(width).size();
    }

    private Integer[] getSortedWidths() {
        Set<Integer> widths = lineWidthHistogram.getWidths();
        Integer[] sortedWidths = (widths.toArray(new Integer[0]));
        Arrays.sort(sortedWidths);
        return sortedWidths;
    }
}
```

6. Objects And Data Structure

- Hiding implementation is about abstractions!
- A class does not simply push its variables out through getters and setter. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data (data itself),

without having to know its implementation

```
// sample-1
public interface Vehicle{
    double getFuelTankCapacityInGallons();
    double getGallansOfGasoline();
}
// sample-2
public interface Vehicle{
    double getPercentFuelRemaining();
}
```

- In the first example, one can be sure that these are just accessors of variables. In the second case one have no clue at all about the form of the data.
- In both of the above cases the second option is preferable
- **We do not want to expose the details of our data. Rather we want to express our data in abstract term**

Data/Object Anti-Symmetry

- Objects hide their data behind abstractions and expose functions that operate on that data
- Data structure expose their data and have no meaningful functions
- For better understanding consider the procedural example. `Geomerty` class operate on three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the `Geomerty` class

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

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

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

public class Geomety{
    public final double PI = 3.1415;

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

```
            return PI * c.radious * c.radious;
        }
        throw new NoSuchShapeException();
    }
}
```

- Consider what would happen if a `perimeter()` function were added to `Geometry`. The shape classes would be unaffected. Any other classes that depended upon the shape would also be unaffected
- Now let's consider the object-oriented solution. Here the `area()` method is polymorphic. No `Gemomety` class is necessary. So if we all a new shape, none of the existing functions are affected, but if we add a new function all of the **shapes must be changed**.

```
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;
    }
}

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}
```

- Therefore, fundamental difference between objects and data structures:
    - **Procedural code(code using data structure) makes it easy to add new functions without changing the existing data structure. OO code, on the other hand, makes it easy to add new classes without changing existing functions**
    - On we can say this like this: **Procedural code makes it hard to add new data structure because all the functions must change. OO code makes it hard to add new functions because all the classes must change.**
    - **So, the things that are hard for OO are easy for procedures and visa verse**
- In any complex system:
    - **When we want to add new data types rather than new functions. For these cases objects and OO are most appropriate.**

    - **On the other hand, when we want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate**

The Law of Demeter

- **Law of Demeter**: says a module should not know about the innards of the objects it manipulates
- **Law of Demeter** says that a method `f` of a class `C` should only call the methods of these:
    - C
    - An object created by `f`
    - An object passed as an argument to `f`
    - An object held in an instance variable of `C`
- The method should not invoke methods on objects that are returned by any of the allowed functions. **In other words, talk to friends, not to strangers**
- This example violates Law of Demeter because it calls the `getScratchDir()` function on the return value of `getOptions()` and then calls `getAbsolutePath()` on the return value of `getScratctDir()`:

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

- Let's divide the above example:

```
public void anyMethod(){
    Options opt = ctxt.getOptions();
    File scratchDir = opts.getScratchDir();
    final String outputDir = scratchDir.getAbsoulutePath();
}
```

- Certainly this functions knows that the `ctxt` object contains options which contain a scratch directory, which has an absolute path. That's a lot of knowledge for one function to know. The calling function knows how to navigate through a lot of different objects.
- Whether this is a violation of Demeter depends on whether or not ctxt , Options , and ScratchDir are objects or data structures.
    - If they are objects, then their internal structure should be hidden rather than exposed and so knowledge of their innards is a clear violation of the Law of Demeter.
    - On the other hand, if ctxt, Options and ScratchDir are just data structures with no behavior, then Demeter does not apply.
- The use of accessor functions confuses the issue. If the code had been written as follows, then we probably wouldn't be asking about Demeter violations:

```
final String outputDir = ctxt.options.scracthDir.absolutePath;
```

- What if `ctxt, options and scrathDir` are objects with real behavior. Then because objects are supposed to hide their internal structure, we should not be able to navigate through them. To get the absolute path of the scratch directory:

```
// we can use:
ctxt.getAbsolutePathOfScratchDirectoryOption();
// or
ctxt.getScratchDirectoryOption().getAbsolutePath();
```

- The first option could lead to an explosion of methods in the `ctxt` object.
- The second presumes that `getScratchDirectoryOption()` returns a data structure, not an object. Neither options feels good.
- If `ctxt` is an object, we should be telling it do something, we should not be asking it about its internals. So why did we want the absolute path of the scratch directory? What were we going to do with it? Consider this code from the same module:

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

- So, what if we told the `ctxt` object to do this?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

- That seems like a reasonable thing for an object to do.

7. Exception Handling

Use Exceptions Rather Than Return Codes

- Don't try to set error flag or returned an error code that the caller could check.

Provide Context with Exceptions

- Each exception that we throw should provide enough context to determine the source and location of an error.
- Create informative error messages and pass them along with your exceptions.

Define Exception Classes in Terms of a Caller's Needs

- When we define exception classes in an application, our most important concern should be **how they are caught**
- Here is the bad example:

```
ACMEPort port = new ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
```

```
    logger.log("Device response exception");
} finally {
...
}
```

- This code contains a lot of duplication.
- We can simplify our code considerably by wrapping the API that we are calling and making sure that it returns a common exception type

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
...
}

// LocalPort class is just a simple wrapper that catches and translates
exceptions thrown by the ACMEPort class

public class LocalPort {
    private ACMEPort innerPort;

    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }

    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
}
```

- **One final advantage of wrapping is that we aren't tied to a particular vendor's API design choice. We can define an API that we feel comfortable with**.
- When we need different actions in try and catch block . In these cases use **Special Case Object Pattern**. In this pattern returns object represents by an Interface and 2 or more classes implements this interface. Example:

```
public interface IProduct
{
    string Name { get; set; }
    decimal Price { get; set; }
    string GetProductInformation();
}

public class ProductNotFound : IProduct
```

```
{
    public string Name { get; set; } = string.Empty;
    public decimal Price { get; set; }
    public string GetProductInformation()
    {
        return "Product not found.";
     }
}

public IProduct GetProductById(int productId)
{
    Product product = _productRepository.GetProductById(productId);
    if(product == null)
        return new ProductNotFound();
    return product;
}
```

- The consumer of this method doesn't have to change at all when we return a special case object the difference is the *GetProductInformation* method which will tell the user that the product is missing from the database.

Don't Return Null

- This is a bad code, instead use Special Case Object Pattern.

```
public void registerItem(Item item) {
    if (item != null) {
        ItemRegistry registry = peristentStore.getItemRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getID());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
            }
        }
    }
}
```

Don't Pass Null

- Returning `null` from methods is bad, but passing `null` into methods is worse.
- Unless we are working with an API which expects us to pass null, we should avoid passing `null` in our code whenever possible.

8. Boundaries

- We seldom conrol all the software in our system. Sometimes we buy third-party packages or use open source.
- Other times we depend on teams in our own company to produce components or subsystems.
- See the practices and techniques to keep the boundaries of our software clean

- For example, when you use `Map` interface which is a boundary interface, because it can be any type, do not use it as an argument for other modules and avoid to return it from a function

Learning Tests

- Learning the 3rd party is hard and integrating that party to the our code is hard too.
- Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the 3rd party code. This is called **learning tests**
- With learning test, we can stays up to the with new version of 3rd party code. If update breaks our test cases, then we can find another way or we decide to use another framework. Therefore we don't need to stay in the older versions.

9. Unit Tests

The Three Laws of TDD

- TDD asks us to write tests first, before we write production code. Follow 3 laws:
- **First Law**: You may not write production code until you have written a failing unit test
- **Second Law**: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- **Third Law**: You may not write more production code than is sufficient to pass the currently failing test.
- The tests and the production code are written *together* with the tests jut a few seconds ahead of the production code.
- **Test code is just as important as production code**. It is not a second-class citizen. It requires thought, design and care. It must be kept as clean as production code.

F.I.R.S.T

- Clean tests follow five other rules
    - **Fast**: Tests should be fast. They should run quickly. When tests run slow, we don't want to run them frequently. If we don't run them frequently, we don't find problems early enough to fix them easily.
    - **Independent**: Tests should not depend on each other. One test should not setup the condition for others. Otherwise, one failure will cascade the all tests cases to fail.
    - **Repeatable**: Tests should be repeatable in any environment.
    - **Self-Validating**: The tests should have a boolean output. Either they pass or fail.
    - **Timely**: The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass

- In general, tests are more important, because they preserve and enhance the flexibility, maintainability and re usability of the production code.

10. Classes

Classes Should be Small

- First rule is that they should be small. Second rule of classes is that they should be smaller than that.
- But the question is "How Small?"
  - With functions we measured size by counting physical lines.
  - With classes we use a different measure. We count **responsibilities**
- Here is the `SuperDashboad` class that exposes about 70 public methods. Everyone can agree with "This small is too big":

```
// 10-1
public class SuperDashboard extends JFrame implements MetaDataUser
    public String getCustomizerLanguagePath()
    public void setSystemConfigPath(String systemConfigPath)
    public String getSystemConfigDocument()
    public void setSystemConfigDocument(String systemConfigDocument)
    public boolean getGuruState()
    public boolean getNoviceState()
    public boolean getOpenSourceState()
    public void showObject(MetaObject object)
    public void showProgress(String s)
        public boolean isMetadataDirty()
    public void setIsMetadataDirty(boolean isMetadataDirty)
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public void setMouseSelectState(boolean isMetadataDirty))
//...
```

- What if we have only a few methods in that class?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

- Five methods isn't too much. In this case it is because despite its small number of methods, `SuperDashborad` has too many responsibilities
- **The name of a class should describe what responsibilities it fulfills**. In fact, naming is probably the first way of helping determine class size.
- If we can not derive a concise name for a class then it's likely too large.
- **We should be able to write a brief description of the class in about 25 words, without using the words:** `if, and, or, but`
- The `SuperDahsboard` provides access to the component that last held the focus and it also allows us to track the version and build numbers. `SuperDashboarda` has too many responsibilities

The Single Responsibility Principle

- **The Single Responsibility Principle(SRP) states that a class or module should have one, and only one reason to change.**
- **Classes should have one responsibility one reason to change**
- In the `SuperDashborad` class we should extract the methods about versions into a separate class named `Version` . The `version` class is a construct that has a high potential for reuse in other applications!

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

- Why we encounter classes that do far too many things?
    - Because many people only focus on the getting the software to work. They are not concern about making software clean.
    - Many of us think that we are done once the program works.
    - At the same time, many developers fear that a large number of small, single-purpose classes makes it more difficult to understand the bigger picture. They are concerned that they must navigate from class to class in order to figure out how a larger piece of work gets accomplished.
        - However, a system with many small classes has no more moving parts than a system with a few large classes. So the question is: Do you want your tools organized into toolboxes with many small drawers each containing well-defined and well-labeled components? Or do you want a few drawers that you just toss everything into ?
    - Every sizable system will contain a large amount of logic and complexity. The primary goal in managing such complexity is to organize it so that a developer knows where to look to find things and need only understand the directly affected complexity at any given time.
- In general, **we want our systems to be composed of many small classes, not a few large ones. Each small class encapsulates a single responsibility has a single reason to change and collaborates with a few others to achieve the desired system behaviors**

Cohesion

- Cohesion refers all about how a single class is designed. Class should be designed with a single, well-focused purpose.
- For example: if we want to multiply 2 numbers and creates pop up window to displaying to them. For high cohesion, we create 2 different class(Multiply & Display). For low cohesion, one class contains all things which is bad.

11. Systems

- How to stay clean at higher levels of abstractions?
- **Software systems should separate the startup process, when the application objects are constructed and the dependencies are wried together, from the runtime logic that takes over after startup**
- The **separation of concerns** is one of the oldest and most important design techniques in our craft.
- Consider this example:

```
public Service getService(){
    if (service == null)
        service = new MyServiceImpl()
    return service;
}
```

- This is the Lazy Initialization/Evaluation idiom, we don't incur the overhead of construction unless we actually use the object.
- However, we not have a hard-coded dependency on `MyServiceImpl`. We can' compile without resolving these dependencies, even if we never actually use an object of this type at runtime.
- Testing can be a problem. If `MyServiceImpl` is a heavyweight object, we will need to make sure that an appropriate *mock object* gets assigned to the service field before this method is called during unit testing.
- Worst of all, we do not know whether `MyServiceImpl` is the right object in all cases. Why does the class with this method have to know the global context? Is it even possible for one type to be right for all possible context?
- If we are diligent(willing) about building well-formed and robust systems, we should never let little, convenient idioms lead to modularity breakdown.

Dependency Injection

- A powerful mechanism for separating construction from use is **Dependency Injection(DI)**, the application of **Inversion of Control(IoC)** to dependency management.
- **Inversion of Control moves secondary responsibilities from an object to other objects that are dedicated to the purpose, thereby supporting the Single Responsibility Principle**
- In the context of dependency management, an object should not take responsibility for instantiating dependencies itself. Instead, it should pass this responsibility to another "authoritative" mechanism, thereby inverting the control.

Scaling Up

- It is a myth that we can get systems "right the first time". Instead, we should implement only today's stories, then refactor and expand the system to implement new stories tomorrow. This is the essence of iterative and incremental agility. Test-driven development, refactoring and the clean code they produce make this work at the code level.
- But what about at the system level? Doesn't the system architecture require pre-planning?
    - **Software systems are unique compared to pyhsical systems. Their architectures can grow incrementally, if we maintain the proper separation of concerns.**
    - the ephemeral nature of software systems makes this possible.

Test Drive The System Architecture

- If we can write our application's domain logic using POJOs, decoupled from any architecture concerns at the code level, then it is possible to truly test drive our architecture.

Optimize Decision Making

- Modularity and separation of concerns make decentralized management and decision making possible. In a sufficiently large system, whether it is a city or a software project, no one person can make all the decisions.

12. Emergence

- Follow 4 rules for good design software/architecture:
    1. Runs all the tests
    2. Contains no duplication
    3. Expresses the intent of the programmer
    4. Minimizes the number of classes and methods

Simple Design Rules - Runs All the Tests

- First and foremost, a design must produce a system that acts as intended.
- Systems that aren't testable aren't verifiable. Arguably, a system that cannot be verified should never be deployed.
- Fortunately, making our systems testable pushes us toward a design where our classes are small and single purpose. It's just easier to test classes that conform to the SRP. **So making sure our system is fully testable helps us create better designs**.
- **Writing tests leads to better designs**

Simple Design Rules - Refactoring

- Once we have tests, we are empowered to keep our code and classes clean. We do this by incrementally refactoring the code.
- For each few lines of code we add, we pause and reflect on the new design. Did we just broke it? If so, we clean it up and run our tests to demonstrate that we haven't broken anything. **The fact that we have these tests eliminates the fear that cleaning up the code will break it**.

No Duplication

- Duplication is the primary enemy of a well-designed system. It represents additional work, additional risk, and additional unnecessary complexity.
- Line of code that look exactly alike are duplication
- As we extract commonality at this very tiny level, we start to recognize violations of SRP. So we might move a newly extracted method to another class.

- Someone else may recognize the opportunity to further abstract the new method and reuse it in a different context. This "reuse in the small" can cause the system complexity to shrink dramatically.
- We can use **Template Method** pattern for removing higher-level duplication. For example:

```
public class VacationPolicy {
    public void accrueUSDivisionVacation(){
    // code to calculate vacation based
    // ...
    // code to ensure vacation meets US
    // ...
    // code to apply vaction to payroll
    // ...
    }

    public void accrueEUDivisionVacation()
    // code to calculate vacation based
    // ...
    // code to ensure vacation meets EU
    // ...
    // code to apply vaction to payroll
    // ...
    }
}
```

- The code across `accrueUSDivisionVacation` and `accrueEUDivisionVacation` is largely the same with the exception of calculating legal minimums. That bit of the algorithm changes based on the employee type.
- We can eliminate the obvious duplication by applying the Template method pattern:

```
public abstract class VacationPolicy{
    public void accrueVacation(){
        calculateBaseVacationHours();
        alterForLegalMinimums();
        applyToPayroll();
    }

    private void calculateBaseVacationHours(){
        /* ... */
    }

    protected abstract void alterForLegalMinimums();

    private void applyToPayroll(){
        /* ... */
    }
}

// ---------------------------------
public class USVacationPolicy extends VacationPolicy{
    @Override
    protected void alterForLegalMinimums() {
        // US specific logic
    }
}
```

```
// ---------------------------------
public class EUVacationPolicy extends VacationPolicy {
    @Override
    protected void alterForLegalMinimums() {
        // EU specific logic
    }
}
```

Simple Design Rules - Expressive

- It is critical for us to be able to understand what a system does.
- As systems become more complex, they take more and more time for a developer to understand, and there is an ever greater opportunity for a mis-understanding. The clearer the author can make the code, the less time others will have to spend understanding it. This will reduce defects and shrink the cost of maintenance.
- We can express ourselves by choosing good names.
- We can express ourselves by keeping functions and classes small. Small classes and functions are usually easy to name, easy to write and easy to understand.
- We can express ourselves by using standard nomenclature.
- Well-written unit tests are also expressive. Someone reading our tests should be able to get a quick understanding of what a class is all about.
- But the most important way to be expressive is to try. All too often we get our code working and then move on to the next problem without gibing sufficient thought to making that code easy for the next person to read. Remember, the most likely next person to read the code will be you.

Simple Design Rules - Minimal Classes and Methods

- I an effort to make our classes and methods small, we might create too many tiny classes and methods. So this rule suggests that we also keep our function and class count low.
- High class and method counts are sometimes the result of pointless dogmatism. For example, a coding standard that insist on creating an interface for each and every class.
- Out goal is to keep our overall system small while we are also keeping our functions and classes small**. Remember, however, that this rule is the lowest priority of the 4 rules of Simple Design**.
- So although, it's important to keep class and function count low, it's more important to have tests, eliminate duplication and express yourself.

13. Concurrency

Why Concurrency?

- Concurrency is a decoupling strategy. It helps us decouple "what gets done" from "when it gets done".
- In single-threaded applications what and when are so strongly coupled.

Myths and Misconceptions About Concurrency

Concurrency always improves performance

- Concurrency can sometimes improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situation is trivial.

Design does not change when writing concurrent programs

- In fact, the design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of what from when usually has a huge effect on the structure of the system.

Challenges

- What makes concurrent programming so difficult? Consider the following trivial class

```
public class X{
    private int lastIdUser;

    public int getNextId(){
        return ++lastIdUsed;
    }
}
```

- Let's say we create an instance of `X`, set the `lastIdUsed` field to 42 and then share the instance between 2 threads. Now suppose that both of those thread call method `getNextd()` there are 3 possible outcomes:
    - Thread one gets the value 43, thread two gets the value 44, lastIdUsed is 44.
    - Thread one gets the value 44, thread two gets the value 43, lastIdUsed is 44.
    - Thread one gets the value 43, thread two gets the value 43, lastIdUsed is 43
- This happens because there are many possible paths that the 2 threads can take through that one line of Java code.

Concurrency Defense Principles

Single Responsibility Principle

- Keep your concurrency-related code separate from other code

Limit the Scope of Data

- Threads that modifying the same field of a shared object can interfere with each other, causing unexpected behavior. One solution is to use the `synchronized` keyword to protect a critical section in the code that uses the shared object.
- Take data encapsulation to heart, severely limit the access of any data that may be shared.

Use Copies of Data

- A good way to avoid shared data is to avoid sharing the data in the first place.
- In some situations it is possible to copy objects and treat them as read-only. In other cases it might be possible to copy objects, collect results from multiple threads in these copies and then merge the results in a single thread.

- You might be concerned about the cost of all extra object creation. It is worth experimenting to find out if this is in fact a problem.

Thread Should Be as Independent as Possible

- Consider writing threaded code such that each thread exists in its own world, sharing no data with any other thread
- Attempt to partition data into independent subsets than can be operated on by independent threads, possibly in different processors.

Know your library

- Review the classes available to you. In the case of Java, become familiar with `java.util,concurrent, java.util.concurrent.atomic, java.util.concurrent.locks`

Know your Execution Models

- **Most concurrent problems you will likely encounter will be some variation of these 3 problems. Study these algorithms and write solutions using them on your own.**

Producer-Consumer

- One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it.
- The queue between the producers and consumers is a bound resource. This means:
    - Producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume.
- Coordination between the producers and consumers via the queue involves producers and consumers signaling each other. The producers write to the queue and signal that the queue is no longer empty. Consumers read from the queue and signal that the queue is no longer full
- The problem is both potentially wait to be notified when they can continue

Readers-Writers

- When you have a shared resource that primarily serves as a source of information for readers, but which is occasionally updated by writers.
- Coordinating readers so they do not read something when a writer is updating and vice versa.
- Writers tend to block many readers for a long period of time, thus causing throughput issues.
- The problem is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput and avoiding starvation.

Dining Philosophers

- Imagine a number of philosophers sitting around a circular table.
    - A fork is placed to the left of each philosopher.
    - There is a big bowl of spaghetti in the center of the table.
    - The philosophers spend their time thinking unless they get hungry. Once hungry they pick up he forks on either side of them and eat
    - A philosopher cannot eat unless he is holding two forks.
    - If the philosopher to his right of left is already using one of the forks he needs, he must wait until that philosopher finishes eating and puts the forks back down.

    - Once a philosopher eats, he puts his forks back down on the table and waits until he is hungry again.
- Replace philosophers with threads and forks with resources and this problem is similar to many enterprise applications in which processes compete for resources.
- Unless carefully designed, systems that compete in this way can experience deadlock, livelock, throughput and efficiency degradation

Beware Dependencies Between Synchronized Methods

- Avoid using more than one method on a shared object

Keep Synchronized Sections Small

- Keep your synchronized sections as small as possible.
- The `synchronized` keyword introduces a lock. All sections of code guarded by the same lock are guaranteed to have only one thread executing through them at any given time.
- Locks are expensive because they create delays and add overhead. So we want to design our code with as few critical sections as possible.
- Some naive programmers try to achieve this by making their critical sections very large. However extending synchronization beyond the minimal critical section increases contention and degrades performance.

Writing Correct Shut-down code is hard

- Think about shut-down early and get it working early. It is going to take longer than you expect. Review existing algorithms because this is probably harder than you think.

Testing Threaded Code

- Write tests that have the potential to expose problems and then run them frequently, with different programatic configurations and system configurations and load. If tests ever fail, track down the failure. Don't ignore a failure just because the tests pass on a subsequent run.
- Do not ignore system failures as one-offs
- Do not try to solve non-threading bugs and threading bugs at the same time. Make sure your code works outside of threads.
- Make your thread-based code especially pluggable so that you can run it in various configurations.
- Multithreaded code behaves differently in different environments. You should run your tests in every potential deployment environment.

14. Smells and Heuristic

Comments

C1: Inappropriate Information

- In general, meta-data such as authors, last-modified-date, SPR number and so on should not appear in comments.
- Comments should be reserved for technical notes about the code and design

C2: Obsolete Comment

- A comment that has gotten old, irrelevant and incorrect is obsolete.
- If you find an obsolete comment, it is best to update it or get rid of it as quickly as possible.

C3: Redundant Comment

- Don't comment like this:

```
i++; // increment i, (really? I thought that it was something else !!!)
```

C4: Poorly Written Comment

- Think twice when you write a comment. Choose words carefully
- Use correct grammar and punctuation.
- Be brief

C5: Commented-Out Code

- Don't do this:

```
public String findIBANNumber(String orderName){
    // boolean isIBANNumberCorrect = checkIBANCorrectWithOrderName(orderName);
    // String region = findRegionFromOrderName(orderName);
    // return findIBANNumber(orderName);
    return ibanService.findIBANNumber(orderName);
}
```

- Now:
    - Who knows how old this comment is?
    - Who knows whether or not it's meaningfull
    - No one will delete it because everyone assumes someone else needs it or has plans for it.
- When you see commented out code, **delete it**!. Don't worry, the source code control system still remembers it. If anyone really needs it, he or she can go back and check out a previous version.

Environment

E1: Build Requires More Than One Step

- You should be able to check out the system with one simple command and then issue one other simple command to build it

```
svn get mySystem
cd mySystem
ant all
```

E2: Tests Require More Than One Step

- You should be able to run all the unit tests with just one command.
- Being able to run all the tests is so fundamental and so important that it should be quick, easy and obvious to do.

Functions

F1: Too Many Arguments

- Functions should have a small number of arguments.
- No argument is best, followed by one, two and three.
- More than three is very questionable and should be avoided

F2: Output Arguments

- Output arguments are counterintuitive.
- If your function must change the state of something, have it change the state of the object it is called on

F3: Flag Arguments

- Boolean arguments loudly declare that the function does more than one thing.
- They are confusing and should be eliminated.

F4: Dead Function

- Methods that are never called should be discarded.
- Keeping dead code around is wasteful.

General

G1: Multiple Languages in One Source File

- The ideal is for a source file to contain one and only one language.
- We should take pains to minimize both the number and extend of extra languages in our source files

G2: Obvious Behavior Is Unimplemented

- Any function or class should implement the behaviors that another programmer could reasonably expect.
- For example, consider a function that translated the name of a day to an `enum` that represents the day.

```
Day day = DayDate.StringToDay(String dayName);
```

- We would expect the String "Monday" to be translated to `Day.MONDAY`

G3: Incorrect Behavior at The Boundaries

- Don't rely on your intuition. Look for every boundary condition and write a test for it.

G4: Overridden Safeties

- Turning off certain compiler warnings (or all warnings) may help you get the build to succeed, but at the risk of endless debugging sessions.

G5: Duplication

- **One of the most important rules in the book**
- Don't Repeat Yourself(DRY principle)
- Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright. By folding the duplication into such an abstraction, you increase the vocabulary of the language of your design. Other programmers can use the abstract facilities you create. Coding becomes faster and less error phone because you have raised the abstraction level.
- Most obvious form of duplication is when you have clumps of identical code that look like some programmers pasting the same code over and over again. These should be replaced with simple methods.
- A more subtle form is the `switch/case` or `if/else` chain that appears again and again in various modules, always testing for the same set of conditions. These should be replaced with polymorphism.
- Still more subtle are the modules that have similar algorithms, but that don't share similar lines of code. This is still duplication and should be addressed by using the **TEMPLATE METHOD** or **STRATEGY** pattern.

G6: Code at Wrong Level of Abstraction

- Isolating abstractions is one of the hardest things that software developers do, and there is no quick fix when you get it wrong
- We want all the lower level concepts to be in the derivatives and all the higher level concepts to be in the base class.
- For example, constants, variables or utility functions that correspond only to the detailed implementation should not be present in the base class.
- Consider the following code:

```
public interface Stack{
    Object pop() throws EmptyException;
    void push(Object o) throws FullException;
    double percentFull();
}
class EmptyException extends Exception {}
class FullException extends Exception {}
```

- The `percentFull` function is at the wrong level of abstraction. No one simply could not know how full they are. So the function would be better placed in a derivative interface such as `BoundedStack`.
- Perhaps you are thinking that the implementation could just return zero, if the stack were boundless. The problem with that is that no stack is truly boundless. You cannot really prevent an `OutOfmemoryException` by checking for

```
stack.percenFull() < 50.0
```

G7: Base Classes Depending on Their Derivatives

- In general, base classes should know nothing about their derivatives.

G8: Too Much Information

- Well-defined modules have very small interfaces that allow you to do a lot with a little.
- Poorly defined modules have wide and deep interfaces that force you to use many different gestures to get simple things done. A well-defined interface does not offer very many functions to depend upon, so coupling is low. A poorly defined interface provides lots of functions that you must call, so coupling is high
- **Hide your data. Hide your utility functions. Hide your constants and your temporaries.**
- **Don't create classes with lots of method or lots of instance variables.**
- **Don't create lots of protected variables and functions for your subclasses**
- Concentrate on keeping interfaces very tight and very small.

G9: Dead Code

- Dead code is code that isn't executed. For example: if statement that checks for a condition that can't happen or try/catch block that never throws.
- The problem with dead code is that they will become older and older for each new version of the system. Because this dead part will never be updated with new system.
- Delete dead code from the systeö

G10: Vertical Separation

- Variables and function should be defined close to where they are used.
- Local variables should be declared just above their first usage and should have limited scope.

- We don't want local variables declared hundreds of lines distant from their usages.
- Private functions should be defined just below their first usage. Private functions belong to the scope of the whole class, but we'd still like to limit the vertical distance between the invocations and definitions.

G11: Inconsistency

- If you do something a certain way, do all similar things in the same way.
- For example, if particular function you use a variable named `response` to hold an `HttpServletResponse`, then use the same variable name consistently in other functions that use `HttpServletResponse`

G12: Clutter(Untidy)

- Of what use is a default constructor with no implementation? All it serves to do is clutter up the code with meaningless artifacts. Same for variables that aren't used, functions that aren't called etc..
- Keep your source files clean, well organized, and free of clutter.

G13: Artificial Coupling

- Things that don't depend upon each other should not be artificially coupled.
- For example, `enums` should not be contained within more specific classes because this forces the whole application to know about these more specific classes.

G14: Feature Envy

- The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes.
- **Feature envy** is a term used to describe a situation in which one object gets at the fields of another object in order to perform some sort of computation or make a decision, rather than asking the object to do the computation itself. It is a kind of : "An object A want to know about details of another object B.".
- However we want to eliminate Feature envy, because it exposes the internals of one class to another.

G15: Obscured Intent

- Be expressive as possible.
- For example (don't do this):

```
public int m_otCalc(){
    return iThsWkd * iThsRte +
        (int) Math.round(0.5 * iTheRte *
                Math.max(0, iThsWdk - 400)
             );
}
```

G16: Misplaced Responsibility

- One of the most important decisions a software developer can make is where to put code.
- The principle of least surprise comes into play here. **Code should be placed where a reader would naturally expect it to be.**

G17: Inappropriate Static

- In general, you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you'll want it to behave polymorphically.
- For example:
  - `Math.max(double a, double b)` is a good static method. It does not operate on a single instance, indeed it would be silly to have to say `new Math().max(a,b)` or even `a.max(b)`. All the data not from owning object. More to point, there is almost no chance that we'd want `Math.max` to be polymorphic.
  - Sometimes, we write static functions that should not be static. For example: `HourlyPayCalculator.calculatePay(employee, overtimeRate)`. This seem like a reasonable static function. However there is reasonable chance that we'll want this function to be polymorphic. We may wish to implement several different algorithms for calculating hourly pay, for example, `OvertimeHourlyPayCalculator` and `StraightTimeHourlyCalculator`. So in this case the function should not be static.

G18: Use Explanatory Variables

- One of the more powerful ways to make a program readable is to break the calculations up into intermediate values that are held in variables with meaningful names.
- For example:

```
Matcher matcher = headerPattern.matcher(line);
if (match.find()){
    String key = match.group(1);
    String value = match.group(2);
    headers.put(key.toLowerCase, value);
}
```

- The simple use of explanatory variables makes it clear that the first matched group is the key, and the second matched group is the value.

G19: Function Names Should Say What They Do

- For example:

```
Date newDate = date.add(5);
```

- Would you expect this to add five days to the date? Or is it weeks, or hours? You can't tell from the call what the function does.

- If the function adds five days to the date and changes the date, then it should be called `addDaysTo` or `increaseByDays`.
- If on the other hand, the function returns a new data that is five days later but does not change the date instance, it should be called `daysLater` or `daysSince`
- To sum up: If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

G20: Understand the Algorithm

- Try to understand "how the code works"
- Best way to gain this knowledge and understanding is to refactor the function into something that is so clean and expressive that it is obvious how it works.

G21: Make Logical Dependencies Physical

- If one module depends upon another, that dependency should be physical, not just logical.
- The dependent module should not make assumptions (in other words, logical dependencies) about the module it depends upon. Rather it should explicitly ask that module for all the information it depends upon.
- For example, imagine that you are writing a function that prints a plain text report of hours worked by employees. One class named `HourlyReporter` gathers all the data into a convenient form and then passes it to `HourlyReportFormatter` to print it.

```
public class HourlyReporter {
    private HourlyReportFormatter formatter;
    private List<LineItem> page;
    private final int PAGE_SIZE = 55;

    public HourlyReporter(HourlyReportFormatter formatter) {
        this.formatter = formatter;
        page = new ArrayList<LineItem>();
    }

    public void generateReport(List<HourlyEmployee> employees) {
        for (HourlyEmployee e : employees) {
            addLineItemToPage(e);
            if (page.size() == PAGE_SIZE)
                printAndClearItemList();
        }

        if (page.size() > 0)
            printAndClearItemList();
     }

    private void printAndClearItemList() {
        formatter.format(page);
        page.clear();
    }

    private void addLineItemToPage(HourlyEmployee e) {
        LineItem item = new LineItem();
```

```
            item.name = e.getName();
            item.hours = e.getTenthsWorked() / 10;
            item.tenths = e.getTenthsWorked() % 10;
            page.add(item);
        }
    }
    ////////////////////
    public class LineItem {
        public String name;
        public int hours;
        public int tenths;
     }
```

- This code has a logical dependency that has not been physicalized. It is the constant `PAGE_SIZE` .
    - Why should be `HourlyReporter` know the size of the page?
    - Page size should be the responsibility of the `HourlyReportFormatter`
- The fact that represents a misplaced responsibility that causes `HourlyReporter` to assume that it knows what the page size ought to be. Such an assumption is a logical dependency. `HouryReporter` depends on the fact that `HourlyReportFormatter` can deal with page sizes of 55. If some implementation of `HourlyReportFormatter` could not deal with such sizes, then there would be an error.
- We can physicalize this dependency by creating a new method in `HourlyReportFormatter` named `getMaxPageSize()` . `HourlyReporter` will then call that function rather than using the `PAGE_SIZE` constant.

G23: Prefer Polymorphism to If/Else or Switch/Case

- Don't forget this point: "switch statements are probably appropriate in the parts of the system where adding new functions is more likely than adding new types." However the cases where functions are more volatile than types are relatively rare. Therefore you should prefer polymorphism rather than if/else or switch/case

G24: Follow Standard Conventions

- Should follow a coding standard based on common industry norms.

G25: Replace Magic Numbers with Named Constants

- For example, the number 86,400 should be hidden behind the constant `SECONDS_PER_DAY` .

G26: Be Precise

- When you make a decision in your code, make sure you make it precisely. Know why you have made it and how you will deal with any exceptions. Don't be lazy about the precision of your decisions.
- For example:

    - If you decide to call a function that might return null, make sure you check for `null`.
    - If there is possibility of concurrent update, make sure you implemented some kind of locking mechanism.

G27: Structure over Convention

- No one is forced to implement the switch / case statement the same way each time; but the base classes do enforce that concrete classes have all abstract methods implemented.

G28: Encapsulate Conditionals

- Boolean logic is hard enough to understand without having to see it in the context of an `if` or `while` statement. Extract functions that explain the intent of the conditional.
- For example, this style is better

```
if (shouldBeDeleted(timer))
```

- than this style:

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29: Avoid Negative Conditionals

- Negatives are just a bit harder to understand than positives. So when possible conditionals should be expressed as positives.

```
if (buffer.shouldCompact())
```

is preferable to

```
if (!buffer.shouldNotCompact())
```

G30: Functions Should Do One Thing

- It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than one thing and should be converted into many smaller functions, each of which does one thing.
- For example:

```
public void pay(){
    for (Employee e : employees){
        if (e.isPayday()){
            Money pay = e.calculatePay();
            e.deliverPay(pay);
        }
    }
}
```

- This bit of code does 3 things. It loops over all the employees, checks to see whether each employee ought to be paid, and then pays the employee. This code would be better written as:

```
public void pay(){
    for (Employee e : employees)
        payIfNecessary(e);
}

private void payIfNecessary(Employee e){
    if (e.isPayday())
        calculateAndDeliverPay(e);
}

private void calculateAndDeliverType(Employee e){
    Money pay = e.calculatePay();
    e.deliverPay(pay);
}
```

G31: Hidden Temporal Couplings

- Temporal couplings are often necessary, but you should not hide the coupling.
- Consider the following:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;

    public void dive(String reason) {
        saturateGradient();
        reticulateSplines();
        diveForMoog(reason);
    }
//...
}
```

- The order of 3 functions is important. Someone should call `reticulateSplines()` before `saturaGradient()` .
- A better solution is:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;
    public void dive(String reason) {
        Gradient gradient = saturateGradient();
        List<Spline> splines = reticulateSplines(gradient);
        diveForMoog(splines, reason);
    }
//...
}
```

- This exposes the temporal coupling. Each function produces a result that the next function needs, so there is no reasonable way to call them out of order.

G32: Don't Be Arbitrary

- Have a reason for the way you structure your code.
- If a structure appears arbitrary, others will feel empowered to change it. If a structure appears consistently throughout the system, others will use it and preserve the convention.

G33: Encapsulate Boundary Conditions

- Boundary conditions are hard to keep track of. Put the processing for them in one place.
- For example:

```
if (level + 1 < tags.length){
    parts = new Parse(body, level + 1, offset+endTag);
    body = null;
}
```

- `level+1` appears twice. This is a boundary condition that should be encapsulated within a variable named something like `nextLevel`

```
int nextLevel = level + 1;
if (nextLevel < tags.length){
    parts = new Parse(body, nextLevel, offset+endTag);
    body = null;
}
```

G34: Functions Should Descend Only One Level of Abstraction

- All statements of a method should belong to the same level of abstraction. If there is a statement which belongs to a lower level of abstraction, it should go to a private method which comprises statements on this level. Doing so will result in smaller methods.

G35: Keep Configurable Data at High Levels

- If you have a constant such as default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function.
- Expose it as an argument to that low-level function called from the high-level function.
- For example:

```
public static void main(String[] args) throws Exception{
    Arguments arguments = parseCommandLine(args);
}

//////////////////////

public class Arguments{
    public static final String DEFAULT_PATH = ".";
    public static final String DEFAULT_ROOT = "FitNesseRoot";
    public static final int DEFAULT_PORT = 80;
    public static final int DEFAULT_VERSION_DAYS = 14;
    // ...
}
```

- The command-line arguments are parsed in the very first executable line.
- The default values of those arguments are specified at the top of the `Argument` class.
- Someone don't have to go looking in low levels of the system for statement like this one:

```
if (arguments.port == 0) // use 80 by default
```

- The configuration constants reside at a very high level and are easy to change. They get passed down to the rest of the application. The lower levels of the application do not own the values of these constants.

G36: Avoid Transitive Navigation

- In general, we don't want a single module to know much about its collaborators. More specifically, if `A` collaborates with `B` and `B` collaborates with `C`, we don't want modules that use `A` to know about `C`. (For example, we don't want `a.getB().getC().doSomething();`)
- If many modules used some form of the statement `a.getB().getC()` then it would be difficult to change the design and architecture to put a `Q` between `B` and `C`. You'd have to find every instance of `a.getB().getC()` and convert it to `a.getB().getQ().getC()`
- Rather we want our immediate collaborators to offer all the services we need.

JAVA

J1: Avoid Long Import Lists by Using Wildcards

- If you use 2 or more classes from a package, then import the whole package with `import package.*;`

J2: Don't Inherit Constant

- A programmer puts some constants in an interface and then gains access to those constants by inheriting that interface. Don't do that.
- Example:

```
public class HourlyEmployee extends Employee{
    private int tenthsWorked;
    private double hourlyRate;

    public Money calculatePay(){
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    }
}
```

- Where did the contansts `TENTHS_PER_WEEK` and `OVERTIME_RATE` come from?

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

- It is not in the Employee class

```
public interface PayrollConstants {
    public static final int TENTHS_PER_WEEK = 400;
    public static final double OVERTIME_RATE = 1.5;
}
```

- This is not good practice. Use a static import instead:

```
import static PayrollConstants.*;
public class HourlyEmployee extends Employee{
    private int tenthsWorked;
    private double hourlyRate;

    public Money calculatePay(){
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    }
}
```

J3: Constants versus Enums

- Use enums!
- Don't keep using the old trick of `public static final int`s . The meaning of ints can get lost.

Names

N1: Choose Descriptive Names

- Don't be too quick to choose a name.
- Make sure the name is descriptive.
- This is not just a "feel-good" recommendation. Names in software are 90 percent of what make software readable.

N2: Choose Names at the Appropriate Level of Abstraction

- Don't pick names that communicate implementation, choose names the reflect the level of abstraction of the class or function you are working in.
- Consider the Modem interface:

```
public interface Modem{
    boolean dial(String phoneNumber);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedPhoneNumber();
}
```

- At first this look fine. The functions all seem appropriate. Indeed, for many applications they are. But now consider an application in which some modems aren't connected by dial-ling. Rather they are connected permanently by hard wiring(think of it like Ethernet). Perhaps some are connected by sending a port number to a switch over a USB connection. A better naming strategy for this scenario might be:

```
public interface Modem{
    boolean connect(String connectionLocator);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedLocator();
}
```

- Now the names don't make any commitments about phone numbers. They can still be used for phone numbers, or they could be used for any other kind of connection strategy.

N3: Use Standard Terminology Where Possible

- Names are easier to understand if they are based on existing convention or usage.
- Follow your team name standard as possible.

N4: Unambiguous Names

- Choose names that make the working of a function or variable unambiguous. For example:

```
private String doRename() throws Exception{
    if (refactorReferences)
        renameReferences();
    renamePage();
    //...
}
```

- The name of the function does not say what the function does except in broad and vague terms. And also there is functional call named `renamePage()` in the function `doRename()`. Who can say the differences between `renamePage()` and `doRename()` ?
- A better name for that function is `renamePageAndOptionallyAllReferences`. This may seem long and it is, but it's only called from one place in the module.

N5: Use Long Names for Long Scopes

- The length of a name should be related to the length of the scope.
- You can use very short variable names for tiny scopes, but for big scope you should use longer names.

N6: Avoid Encodings

- Names should not be encoded with type or scope information.
- Prefixes such as `m_` or `f` are useless
- Also project and/or subsystem encodings such as `vis_` (for visual imaging system) are distracting and redundant.

N7: Names Should Describe Side-Effects

- Names should describe everything that a function, variable, or class is or does. Don't hide side effects with a name. For example:

```
public ObjectOutputStream getOos() throws IOException{
    if (m_oos == null){
        m_oss = new ObjectOutputStream(m_socket.getOutputStream());
    }
    return m_oss;
}
```

- This function does a bit more than get an "oss"; it creates the oss if it hasn't been created already. Thus, a better name might be `createOrReturnOos`

Tests

T1: Insufficient Tests

- How many tests should be in a test suite?
- Unfortunately, the metric many programmers use "That seems like enough".
- A test suite should test everything that could possibly break.

T2: Use a Coverage Tool

- Coverage tools reports gaps in your testing strategy.
- They make it easy to find modules, classes, and functions that are insufficiently tested.

T3: Don't Skip Trivial Tests

- They are easy to write.

T4: An Ignored Test Is a Question about an Amubiguity

T5: Test Boundary Conditions

- Take special care to test boundary conditions.

T6: Exhaustively Test Near Bugs

- When you find a bug in a function, it is wise to do an exhaustive test of that function. You'll probably find that the bug was not alone.

T7: Patterns of Failure Are Revealing

- Sometimes you can diagnose a problem by finding patterns in the way the test cases fail. This is another argument for making the test cases as complete as possible. Complete test cases, ordered in a reasonable way.

T8: Test Coverage Patterns Can be Revealing

T9: Tests Should be Fast

Conclusion

- You don't become a software craftsman by learning a list of heuristics or reading this book. Professionalism and craftsmanship come from values that drive disciplines.