# Naming

In addition to basic naming conventions which all programmers of any language MUST KNOW, including:

* Name variables, functions, classes, etc. meaningfully and easily to understand without the need of comments
* Avoid general names (num, str, arr, etc.) and single-character names (i, j, k, etc.)
* Use searchable names
* Use nouns or noun phrases for structures, classes, objects, modules, packages.
* Use verbs or verb phrases for functions, methods.

## Avoid Hungarian Notation (HN)

In days of old, HN was considered to be helpful in the Windows C API, when everything was an integer handle or a long pointer or a void pointer, or one of several implementations of "string" (with different uses and attributes). The compiler did not check types in those days, so the programmers needed a crutch to help them remember the types by adding a type prefix before each name:

* lAccountNum: variable is a long integer.
* arru8NumberList: variable is an array of unsigned 8-bit integers.
* bReadLine(): function with a byte-value return code.
* strName: variable represents a string containing the name.
* ...

In modern languages, we have much richer type systems, and the compilers remember and enforce the types. What’s more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they’re using.

Java programmers don’t need type encoding. Objects are strongly typed, and editing environments have advanced such that they detect a type error before you can run a compile. So **nowadays, HN and other forms of type encoding are simply impediments**.

They make it harder to change the name or type of a variable, function, or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.

## Avoid Member Prefixes

You also don’t need to prefix member variables with m\_ anymore (eg, m\_name). Your classes and functions should be small enough that you don’t need them. Also you can **use the this keyword (in Java, C#, C++ – eg, this.name) or self keyword (in Python, Objective-C) to call member variables**; this way everything is very clear.

Even better, you should be using an IDE or a text editor that can highlight members to make them distinct.

## Avoid Interface Prefixes

These are sometimes a special case for encodings. For example, say you are building an Abstract Factory for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? IShapeFactory and ShapeFactory?

I prefer to leave interfaces unadorned. The preceding "I", which is so common in today’s legacy wads, is a distraction at best and too much information at worst. I don’t want my users knowing that I’m handing them an interface. I just want them to know that it’s a ShapeFactory.

So if I must encode either the interface or the implementation, I choose the implementation. Calling it ShapeFactoryImp, or even the hideous CShapeFactory, is preferable to encoding the interface.

## When Prefixes?

Imagine you 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 you just saw the state variable being used alone in a method? Would you automatically infer that it was part of an address?

In this case, you can add context by using prefixes: addrFirstName, addrLastName, addrState, and so on. At least readers will understand that these variables are part of a larger structure. Of course, a better solution is to create a class named Address. Then, even the compiler knows that the variables belong to a bigger concept.

## 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. How do you remember which method name goes with which class?

For example, 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 controllers or both not managers? Are they both drivers? The name leads you to expect two objects that have very different type as well as having different classes.

Function names have to stand alone, and they have to be consistent in order for you to pick the correct ones without any additional exploration.

## Use Solution Domain Names

Remember that the people who read your code will be programmers. So if you can, **use computer science terms, algorithm names, pattern names, math terms, and so forth**.

It’s not wise to draw every name from the problem domain because we don’t want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name. For example, the name AccountFactory means a lot to programmers who are familiar with the Factory design pattern.

There are lots of technical things that programmers have to do. Choosing technical names for those things is usually very appropriate.

# Functions

## Functions Should Do One Thing

This (following the Single Responsibility design principle) is by far the most important rule in software engineering. When functions do more than one thing, they are harder to compose, test, and reuse. When you can isolate a function to just one action, they can be refactored easily and your code will be much cleaner.

Bad:

# python

def email\_clients(clients: List[Client]):

    """Filter active clients and send them an email.

    """

    for client in clients:

        if client.active:

            email(client)

Good:

# python

def get\_active\_clients(clients: List[Client]) -> List[Client]:

    """Filter active clients.

    """

    return [client for client in clients if client.active]

def email\_clients(clients: List[Client, ...]) -> None:

    """Send an email to a given list of clients.

    """

    for client in clients:

        email(client)

Even better: Do you see an opportunity for using generators now?

# python

def active\_clients(clients: List[Client]) -> Generator[Client]:

    """Only active clients.

    """

    return (client for client in clients if client.active)

def email\_client(clients: Iterator[Client]) -> None:

    """Send an email to a given list of clients.

    """

    for client in clients:

        email(client)

IMPORTANT NOTE:

"Function should do one thing" doesn’t mean that we should break down the functions to *as small as possible* ones.

Breaking down big blocks of code having multiple functionilties into multiple smaller functions is definitely good. **But overdoing it is clearly very bad**. For example, do you really need a separate function called addTwoNums()? I don’t think so!

In addition, deciding whether to break a function down **depends on the coupling**. You can have a bunch of small ones which are tighly coupled together => good thing. But having smallA() calls smallB() which calls smallC() is highly coupled => bad (you might have to jump around just to understand a little picture of your code).

All about the balance :)

## Functions Should Only Be One Level of Abstraction

When you have more than one level of abstraction, your function is usually doing too much. Splitting up functions leads to reusability and easier testing.

Bad:

# python

def parse\_better\_js\_alternative(code: str) -> None:

    regexes = [

        # ...

    ]

    statements = regexes.split()

    tokens = []

    for regex in regexes:

        for statement in statements:

            # ...

    ast = []

    for token in tokens:

        # Lex...

    for node in ast:

        # Parse...

Good:

# python

REGEXES = (

   # ...

)

def parse\_better\_js\_alternative(code: str) -> None:

    tokens = tokenize(code)

    syntax\_tree = parse(tokens)

    for node in syntax\_tree:

        # Parse...

def tokenize(code: str) -> list:

    statements = code.split()

    tokens = []

    for regex in REGEXES:

        for statement in statements:

           # Append the statement to tokens...

    return tokens

def parse(tokens: list) -> list:

    syntax\_tree = []

    for token in tokens:

        # Append the parsed token to the syntax tree...

return syntax\_tree

## Avoid Side Effect

**A function produces a side effect if it does anything other than take a value in and return another value or list of values**. For example, a side effect could be writing to a file in inappropriate methods, or modifying some global variables.

You do need to have side effects in a program on occasion. In these cases, you should **centralize and indicate where you are incorporating side effects**. For example, when writing to a file, make sure you don't have several functions and classes that can write to that file - rather, have one (and only one) service that does it.

**The main point is to avoid common pitfalls like sharing state between objects without any structure, using mutable data types that can be written to by anything**, or using an instance of a class, and not centralizing where your side effects occur.

For example, the following function uses a standard algorithm to match a userName to a password. It returns true if they match; otherwise false. But it also has a side effect. Can you spot it?

// Java

public class UserValidator

{

    private Cryptographer cryptographer;

    public boolean checkPassword(String userName, String password)

    {

        User user = UserGateway.findByName(userName);

        if (user != User.NULL)

        {

            String codedPhrase = user.getPhraseEncodedByPassword();

            String phrase = cryptographer.decrypt(codedPhrase, password);

            if ("Valid Password".equals(phrase))

            {

                Session.initialize();

                return true;

            }

        }

        return false;

    }

}

The side effect is the call to Session.initialize(). The checkPassword() checks the password, and its name does not imply that it initializes the session. So a call to Session.initialize() would run the risk of erasing the existing session data when he or she decides to check the validity of the user.

This side effect creates a temporal coupling. That is, checkPassword() can only be 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. Temporal couplings are confusing, especially when hidden as a side effect. If you must have a temporal coupling, you should make it clear in the name of the function. In this case we might rename the function checkPasswordAndInitializeSession(), though that certainly violates "Do one thing."

## Command Query Separation

Functions should either do something or answer something, but not both. **Either a function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion**.

For example:

public boolean set(String name, String value) {

    // if attribute 'name' doesn't exist, return false

    // set 'value' to 'name', then return true

}

// In main():

if (set("username", "unclebob")) {

    ...

}

The above set function leads to a confusing if statement. What does it mean? Is it asking whether the name attribute was previously set to unclebob? Or is it asking whether the name attribute was successfully set to unclebob? It’s hard to infer the meaning from the call because it’s not clear whether the word "set" is a verb or an adjective.

We could try to resolve this by renaming the function to setAndCheckIfExists, but that doesn’t much help the readability of the if statement. The real solution is to separate the command from the query so that the ambiguity cannot occur.

if (attributeExists("username")) {

    setAttribute("username", "unclebob");

    ...

}

## Avoid Flags as Function Parameters

**Flags tell your user that this function does more than one thing** – one thing if the flag is true and another if the flag is false. But remember that functions should do one thing. So, split your functions if they are following different code paths based on a boolean.

Bad:

# python

from pathlib import Path

def create\_file(name: str, temp: bool) -> None:

    if temp:

        Path('./temp/' + name).touch()

    else:

        Path(name).touch()

Good:

# python

from pathlib import Path

def create\_file(name: str) -> None:

    Path(name).touch()

def create\_temp\_file(name: str) -> None:

    Path('./temp/' + name).touch()

## Use Default Arguments Instead of Conditionals

**Default arguments are very readable for optional parameters** to a method.

# python

def create\_micro\_brewery(name):

    name = "Hipster Brew Co." if name is None else name

    ...

Good:

# python

def create\_micro\_brewery(name: str = "Hipster Brew Co."):

    ...

## Avoid Output Arguments

When you are reading the story told by the module, StringBuffer includeSetupPage() is easier to understand than void includeSetupPageInto(StringBuffer newPageContent).

Output arguments are harder to understand than input arguments. When we read a function, we are used to the idea of information going in to the function through arguments and out through the return value. We don’t usually expect information to be going out through the arguments. So output arguments should be avoided when possible.

## How Many Arguments for a Function?

The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), then two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification – and then shouldn’t be used anyway.

In most cases, the answer to the question "how many arguments for a function?" is gotten by answering "*should pass a variable as an argument to a method or making it an instance variable?*". There are some rules of thumb you can follow for deciding whether to use an instance variable:

- Instance variables are typically considered to be attributes of a class. Think of these as adjectives of the object that will be created from your class. **If your instance data can be used to help describe the object, it's a good choice for instance data**.

- Local variables are used within the scope of methods to help them complete their work. Usually, a method should have a purpose of getting some data, returning some data, and/or proccessing/running an algorithm on some data. Sometimes, it helps to think of local variables as ways of helping a method get from beginning to end.

- Mainly this depends on the lifetime of the data you store in the variable. If the data is only used during a computation, pass it as a parameter. **If the data is bound to the lifetime of the object, use an instance variable**. **That mostly also means that instance variables are only necessary when the data will be used across calls**.

Note: When your list of instance variables gets too long, maybe it's a good point to think about refactoring some parts of the class into a new class.

Cons of function arguments:

- Readers would have to interpret an argument each time they see it. Arguments are at a different level of abstraction than function names and force you to know a detail that isn’t particularly important at some cases.

- Arguments are even harder from a testing point of view. Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly.

### Common Monadic Forms

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in boolean fileExists("MyFile"). Or you may be operating on that argument, transforming it into something else and returning it, as in InputStream fileOpen("MyFile"). These two uses are what readers expect when they see a function.

A less common yet very useful form for a single argument function, is an *event*. In this form there is an input argument but no output argument. The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system, for example, void passwordAttemptFailedNtimes(int attempts).

Try to avoid any monadic functions that don’t follow these forms, for example, void includeSetupPageInto(StringBuffer newPageContent).

### Dyadic Functions

A function with two arguments is harder to understand than a monadic function. For example, writeField(name) is easier to understand than writeField(output-Stream, name). Though the meaning of both is clear, the first glides past the eye, easily depositing its meaning.

There are times, of course, where two arguments are appropriate. For example, Point p = new Point(0,0) is perfectly reasonable. Cartesian points naturally take two arguments. Indeed, we’d be very surprised to see new Point(0).

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

Dyads aren’t evil, and you will certainly have to write them. However, you should be aware that they come at a cost and should take advantage of what mechanims may be available to you to convert them into monads. For example, you might make the writeField() a member of OutputStream so that you can say OutputStream.writeField(name). Or you might make the outputStream a member variable of the current class so that you don’t have to pass it. Or you might extract a new class like FieldWriter that takes the outputStream in its constructor and has a write method.

### Triadic Functions

Functions that take three arguments are significantly harder to understand than dyads. The issues of ordering, pausing, and ignoring are more than doubled. I suggest you think very carefully before creating a triad.

For example, consider the common overload of assertEquals that takes three arguments: assertEquals(message, expected, actual). How many times have you read the message and thought it was the expected? I have stumbled and paused over that particular triad many times. In fact, every time I see it, I do a double-take and then learn to ignore the message.

## Argument Objects Are Good

When a function seems to need more than two or three arguments, it is likely that some of those arguments should be wrapped into a class of their own. For example, the difference between the two following declarations:

Circle makeCircle(double x, double y, double radius);

Circle makeCircle(Point center, double radius);

Reducing the number of arguments by creating objects out of them may seem like cheating, but it’s not. When groups of variables are passed together, the way x and y are in the example above, they are likely part of a concept that deserves a name of its own.

## Argument Lists Are Good

Sometimes we want to pass a variable number of arguments into a function. For example, the String.format("%s worked %.2f hours.", name, hours). If variable arguments are all treated identically, then they are equivalent to a single argument of type List. By that reasoning, String.format is actually dyadic. Indeed, its declaration is clearly dyadic – public String format(String format, Object... args).

So all the same rules apply. Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that.

void monad(Integer... args);

void dyad(String name, Integer... args);

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

## Prefer Exceptions to Returning Error Codes

Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of if statements. This leads to deeply nested structures. When you return an error code, you create the problem that the caller must deal with the error immediately.

// java

if (deletePage(page) == E\_OK) {

    if (registry.deleteReference(page.name) == E\_OK) {

        if (configKeys.deleteKey(page.name.makeKey()) == E\_OK) {

            logger.log("page deleted");

        }

        else {

            logger.log("configKey not deleted");

        }

    }

    else {

        logger.log("deleteReference from registry failed");

    }

}

else {

    logger.log("delete failed");

    return E\_ERROR;

}

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

try {

    deletePage(page);

    registry.deleteReference(page.name);

    configKeys.deleteKey(page.name.makeKey());

}

catch (Exception e) {

    logger.log(e.getMessage());

}

### Extract Try/Catch Blocks

The try/catch blocks are, without a doubt, a great tool for error handling. But they’re ugly in their own right because they mix error processing with normal processing. So it is better to **extract the bodies of the try/catch blocks out into functions of their own**.

// java

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 code, 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.

**Functions should do one thing, and error handing is one thing**. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that **if the keyword try exists in a function, it should be the very first word in the function and that there should be nothing after the catch/finally blocks**.

## How Do You Write Functions Like This?

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code.

So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.

In the end, I wind up with functions that follow the rules I’ve laid down in this chapter. I don’t write them that way to start. I don’t think anyone could.

# Comments

## Explain Yourself in Code, Not in Comments

There are certainly times when code makes a poor vehicle for explanation. And as a result, we choose comments to make our code easier to understand. But this is really not a good choice.

Which would you rather see? This:

// Check to see if the employee is eligible for full benefits

if ( (employee.flags & HOURLY\_FLAG) && (employee.age > 65) ) { ... }

Or this?

if (employee.isEligibleForFullBenefits()) { ... }

One of the most common motivations for **writing comments is bad code**. We write a module and we know it is confusing and disorganized. We know it’s a mess. So we say to ourselves, “Ooh, I’d better comment that!” No! *You’d better clean it*!

## Comments Waste Time

Code changes and evolves over time. Chunks of it move from here to there. Those chunks bifurcate and reproduce and come together again to form chimeras. Unfortunately the **comments don’t always follow code changes – can’t always follow them**. And all too often the comments get separated from the code they describe and become orphaned blurbs of everdecreasing accuracy.

Due to the lack of time, we cannot always fix these comments. Not to mention that **writing a bunch of comments is really time-consuming**.

## When Comments Are Good?

### Legal Comments

Sometimes our corporate coding standards force us to write certain comments for legal reasons. For example, copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file.

// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.

// Released under the terms of the GNU General Public License version 2 or later.

### Informative Comments

There are many cases where we find it helpful to provide basic information with a comment. For example:

// format matched kk:mm:ss EEE, MMM dd, yyyy

Pattern timeMatcher = Pattern.compile("\\d\*:\\d\*:\\d\* \\w\*, \\w\* \\d\*, \\d\*");

### Explanation of Intent

Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. In the following case, we see an interesting decision documented by a comment. When comparing two objects, the author decided that he wanted to sort objects of his class higher than objects of any other.

// java

public int compareTo(Object o)

{

    if (o instanceof WikiPagePath) {

        WikiPagePath p = (WikiPagePath)o;

        String compressedName = StringUtil.join(names, "");

        String compressedArgumentName = StringUtil.join(p.names, "");

        return compressedName.compareTo(compressedArgumentName);

    }

    return 1; // we are greater because we are the right type.

}

Here’s an even better example. You might not agree with the programmer’s solution to the problem, but at least you know what he was trying to do.

// java

public void testConcurrentAddWidgets() throws Exception

{

    WidgetBuilder widgetBuilder = new WidgetBuilder(new Class[]{BoldWidget.class});

    String text = "'''bold text'''";

    ParentWidget parent = new BoldWidget(new MockWidgetRoot(), "'''bold text'''");

    AtomicBoolean failFlag = new AtomicBoolean();

    failFlag.set(false);

    // This is our best attempt to get a race condition by creating large number of threads.

    for (int i = 0; i < 25000; i++) {

        WidgetBuilderThread widgetBuilderThread = new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);

        Thread thread = new Thread(widgetBuilderThread);

        thread.start();

    }

    assertEquals(false, failFlag.get());

}

### Warning of Consequences

Sometimes it is useful to warn other programmers about certain consequences. For example, here is a comment that explains why a particular test case is turned off:

// java

// 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);

}

Nowadays, we’d turn off the test case by using the @ignore attribute with an appropriate explanatory string – @ignore("Takes too long to run"). But back in the days before JUnit 4, putting an underscore in front of the method name was a common convention. The comment, while flippant, makes the point pretty well.

Here’s another more poignant example:

// java

public static SimpleDateFormat makeStandardHttpDateFormat()

{

    // SimpleDateFormat is not thread safe, so we need to create each instance independently.

    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");

    df.setTimeZone(TimeZone.getTimeZone("GMT"));

    return df;

}

You might complain that there are better ways to solve this problem. I might agree with you. But the comment, as given here, is perfectly reasonable. It will prevent some overly eager programmer from using a static initializer in the name of efficiency.

### TODO Comments

It is reasonable to leave "todo" notes in the form of // TODO comments to notify somethings that should be done, but for some reasons can’t do at the moment. In the following case, the TODO comment explains why the function has a degenerate implementation and what that function’s future should be.

// java

// 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;

}

### Docs in Public APIs

If you are writing a public API, then you should certainly write good docs (javadocs, python docstring, etc.) for it. But keep in mind that Javadocs can be just as misleading, nonlocal, and dishonest as any other kind of comment.

## When Comments Are Bad?

### Mandated Comments

It is silly to have a rule that says that every function must have a javadoc, or every variable must have a comment. Comments like in the following example just clutter up the code without giving any valueable information:

// java

/\*\*

\* Add CD to the list

\* @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);

}

### Change Log Comments

Sometimes people add a comment to the start of a module every time they edit it. These comments accumulate as a kind of logs. For example:

/\*

\* Changes (from 11-Oct-2001)

\* --------------------------

\* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate

\* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);

\* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);

\* 13-Mar-2003 : Implemented Serializable (DG);

\* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);

\* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);

\*/

Long ago, there was a good reason to create and maintain these log entries at the start of every module. We didn’t have source code control systems that did it for us. Nowadays, however, these long change logs are just more clutter to obfuscate the module. They should be removed.

### Noise Comments

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

/\*\*

\* Default constructor

\*/

~myClass() {

}

Javadocs can also be noisy. What purpose do the following Javadocs (from a well-known open-source library) serve? Answer: nothing. They are just redundant noisy comments written out of some misplaced desire to provide documentation. Read carefully and you can also even see the cut-paste error:

/\*\* The name. \*/

private String name;

/\*\* The version. \*/

private String version;

/\*\* The licenceName. \*/

private String licenceName;

/\*\* The version. \*/

private String info;

# Switch Statements (need edit)

It’s hard to make a small switch statement. Even a switch statement with only two cases is

larger than I’d like a single block or function to be. It’s also hard to make a switch state

ment that does one thing. By their nature, switch statements always do N things. Unfortu

nately 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, of course, with

polymorphism.

Consider Listing 3-4. It shows just one of the operations that might depend on the

type of employee.

public Money calculatePay(Employee e) throws InvalidEmployeeType

{

    switch (e.type)

    {

    case COMMISSIONED:

        return calculateCommissionedPay(e);

    case HOURLY:

        return calculateHourlyPay(e);

    case SALARIED:

        return calculateSalariedPay(e);

    default:

        throw new InvalidEmployeeType(e.type);

    }

}

There are several problems with this function. 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 Principle7 (SRP) because there is more than one

reason for it to change. Fourth, it violates the Open Closed Principle8 (OCP) 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

deliverPay(Employee e, Money pay),

or a host of others. All of which would have the same deleterious structure.

The solution to this problem (see Listing 3-5) is to bury the switch statement in the

basement of an ABSTRACT FACTORY,9 and never let anyone see it. The factory will use the

switch statement to create appropriate instances of the derivatives of Employee, and the var

ious functions, such as calculatePay, isPayday, and deliverPay, will be dispatched poly

morphically through the Employee interface.

My general rule for switch statements is that they can be tolerated if they appear

only once, are used to create polymorphic objects, and are hidden behind an inheritance relationship so that the rest of the system can’t see them [G23]. Of course every circum

stance is unique, and there are times when I violate one or more parts of that rule.

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);

        }

    }

}