A piece of code should be where you expect to find it—and, if not, you should re-factor to get it there.

Clean code does one thing well

Bad code tries to do too much

Must be tested and contains no duplication

Validate if a method or object is doing more than one thing

Array with values might be better to create a class and map the values there

Do not use accountList unless it is actually a real list

Preferable FactoryImpl than IFactory

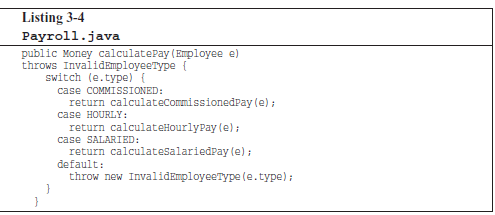
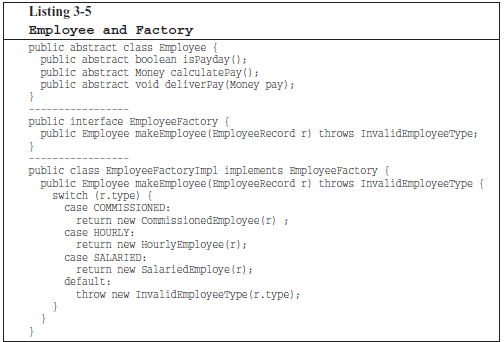
Class names, nouns no verbs

Methods, verbs

Functions should be small

Functions should do only one thing

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

Never more than 3 arguments on the function.

If a function is going to transform its input argument, the transformation should appear as the return value.

Passing a boolean into a function is a truly terrible practice. It does one thing if the flag is true and another if the flag is false

Dyadic functions. you might make the writeField method 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.

When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.

String.format("%s worked %.2f hours.", name, hours)-> public String format(String format, Object... args)

Write(name) -> better writeField(name)

Anything that forces you to check the function signature is equivalent to a double-take. It’s a cognitive break and should be avoided.

In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.

Functions should either do something or answer something, but not both.

public boolean set(String attribute, String value); ->

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

Classes like this are a dependency magnet; many other classes must import and use them. Thus, when the Error enum changes, all those other classes need to be recompiled and redeployed.11 This puts a negative pressure on the Error class.

a function, should have one entry and one exit

when you find yourself in a position where you need to write a comment, think it through and see whether there isn’t some way to turn the tables and express yourself in code.

Instance variables, on the other hand, should be declared at the top of the class. This should not increase the vertical distance of these variables, because in a well-designed class, they are used by many, if not all, of the methods of the class.

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

Objects hide their data and expose operations.

the Law of Demeter says that a method f of a class C should only call the methods of C and object created by f, and object passed as argument to f, object held in and instance var of C.

The method should not invoke methods on objects that are returned by any of the allowed functions

Ex: it calls the getScratchDir() function on the return value of getOptions() and then calls getAbsolutePath() on the return value of getScratchDir().

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

It is usually best to split them up as follows:

Options opts = ctxt.getOptions();

File scratchDir = opts.getScratchDir();

final String outputDir = scratchDir.getAbsolutePath();

the objective of before code was to create a file wit the output dir, so instead:

BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);

This allows ctxt to hide its internals and prevents the current function from having to violate the Law of Demeter by navigating through objects it shouldn’t know about.

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors.

Often a single exception class is fine for a particular area of code. The information sent with the exception can distinguish the errors. Use different classes only if there are times when you want to catch one exception and allow the other one to pass through.

If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead

If

you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.

F.I.R.S.T.8

Clean tests follow five other rules that form the above acronym:

Fast Tests should be fast. They should run quickly. When tests run slow, you won’t want

to run them frequently. If you don’t run them frequently, you won’t find problems early

enough to fix them easily. You won’t feel as free to clean up the code. Eventually the code

will begin to rot.

Independent Tests should not depend on each other. One test should not set up the conditions

for the next test. You should be able to run each test independently and run the tests in

any order you like. When tests depend on each other, then the first one to fail causes a cascade

of downstream failures, making diagnosis difficult and hiding downstream defects.

Repeatable Tests should be repeatable in any environment. You should be able to run the

tests in the production environment, in the QA environment, and on your laptop while

riding home on the train without a network. If your tests aren’t repeatable in any environment,

then you’ll always have an excuse for why they fail. You’ll also find yourself unable

to run the tests when the environment isn’t available.

Self-Validating The tests should have a boolean output. Either they pass or fail. You

should not have to read through a log file to tell whether the tests pass. You should not have

to manually compare two different text files to see whether the tests pass. If the tests aren’t

self-validating, then failure can become subjective and running the tests can require a long

manual evaluation.

Classes should have one responsibility—one reason to change.

Trying to identify responsibilities (reasons to change) often helps us recognize and create better abstractions in our code.

Unfortunately, this also means that our classes lose cohesion because they accumulate more and more instance variables that exist solely to allow a few functions to share them. But wait! If there are a few functions that want to share certain variables, doesn’t that make them a class in their own right? Of course it does. When classes lose cohesion, split them!

According to Kent, a design is “simple” if it follows these rules:

• Runs all the tests

• Contains no duplication

• Expresses the intent of the programmer

• Minimizes the number of classes and methods

For each few lines of code we add, we pause and reflect on the new design. Did we just degrade it? If so, we clean it up and run our tests to demonstrate that we haven’t broken anything.

Character.isLetter(char)

It is usually a bad idea to pass a flag as an argument to a function, especially when that flag simply selects the format of the output

Comments should say things that the code cannot say for itself.

you should be able to run all the unit tests with just one command.

Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on.

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

The ideal is for a source file to contain one, and only one, language

That duplication could probably become a subroutine or perhaps another class outright.

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 STRATEGY5 pattern.

It is important to create abstractions that separate higher level general concepts from lower level detailed concepts. Sometimes we do this by creating abstract classes to hold the higher level concepts and derivatives to hold the lower level concepts. When we do this, we need to make sure that the separation is complete. 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 pertain only to the detailed implementation should not be present in the base class. The base class should know nothing about them.

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.

Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better.

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 a small vertical scope. We don’t want local variables declared hundreds of lines distant from their usages.

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

Of course, selectors need not be boolean. They can be enums, integers, or any other type of argument that is used to select the behavior of the function. In general it is better to have many functions than to pass some code into a function to select the behavior

HourlyPayCalculator.calculatePay(employee, overtimeRate).

Again, this seems like a reasonable static function. It doesn’t operate on any particular object and gets all it’s data from it’s arguments. However, there is a reasonable chance that we’ll want this function to be polymorphic. We may wish to implement several different algorithms for calculating hourly pay

Prefer Polymorphism to If/Else or Switch/Case

The constants are hidden at the top of the inheritance hierarchy.

Ick! Don’t use inheritance as a way to cheat the scoping rules of the language. Use a static import instead.

import static PayrollConstants.\*;

good code, run all tests, no duplication, express ideas clearly, minimize number of classes, methods, etc

class name nounds method verbs

For example, you might make the writeField method 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.

Functions one entry one exit

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

Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation.

ctx.getScratchDirectoryOption().getAbsolutePath()

TO

BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);

Do not traverse over objects, getting access to its internals, tell the object to do something instead.

If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead

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

Test an api creating tests!

TDD generate a failing test, then work on the code to make it work then again

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 cannot derive a concise name for a class, then it’s likely too large. The more ambiguous the class name, the more likely it has too many responsibilities

Classes should have only one reason to change (SingleResponsability)

Trying to identify responsibilities (reasons to change) often helps us recognize and create better abstractions in our code.

Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. In general the more variables a method manipulates the more cohesive that method is to its class

If we promoted those four variables to instance variables of the class, then we could extract the code without passing any variables at all. It would be easy to break the function up into small pieces.

Unfortunately, this also means that our classes lose cohesion because they accumulate more and more instance variables that exist solely to allow a few functions to share them.

But wait! If there are a few functions that want to share certain variables, doesn’t that make them a class in their own right? Of course it does. When classes lose cohesion, split them!

Needs will change, therefore code will change. We learned in OO 101 that there are concrete classes, which contain implementation details (code), and abstract classes, which represent concepts only. A client class depending upon concrete details is at risk when those details change. We can introduce interfaces and abstract classes to help isolate the impact of those details.