Coding at the Lowest Level
Coding Patterns for Java Beginners

Joseph Bergin
Pace University
jbergin@pace.edu
http://csis.pace.edu/~bergin

Introduction

Beginners who are learning programming make many mistakes. This is because they have little experience and don't always get good guidance. If they look at a lot of programs we hope they will emulate the style, but they don't always do this, and they don't always see programs that should be emulated in any case. This is an attempt to provide advice to novices learning Java. Not everything here is Java specific, however.

Some of the patterns have been written for this paper. Others have been collected from the literature. The latter have been adapted to Java perhaps, but their originators are referenced.

The Principles

There are a number of reasons for the rules which follow. We will introduce some of them here.

The first is that our programs must be readable by people. Programs are not written just to communicate with the machine. They are read by people far more often than they are by machines. Real programs also tend to live for a long time and really valuable programs are read by many people over a long period of time. The intent of the programs we write is not always obvious, so we need to take care that we make it easy for an unfamiliar reader to learn what we intend in the program.

The second principle is that our programs must be maintainable. It must be easy to modify a program to change what it does. This is because the problem for which the program was originally written will evolve and change over time. This is most true for the most valuable programs, since these are used in businesses whose needs evolve and change over time.

Additionally, programs must be reliable. We need to be able to trust them to do what is right and helpful. We therefore need to use techniques that assure us that we don't introduce errors into the programs we write.

There are other considerations such as overall performance of a system and the cost effectiveness of our solutions. While performance is often over emphasized, nevertheless our programs must perform well enough to do the task they were designed for. Cost effectiveness implies that we don't waste resources, including programmer time.

A Note on the Form

In this paper we use a form called modified Alexandrian form. Christopher Alexander is an architect who was the inspiration for the idea of software patterns through his writings, especially A Pattern Language (Oxford, 1977). The way we present material is a modification of his style. Each pattern begins with a name. This is followed by a short description of the problem that the pattern addresses. This discussion will include a number of "forces" that the users must consider to decide if the pattern is appropriate in their current context.

The section beginning with a bold face **Therefore**... is the introduction to the solution. This is what you need to do to apply the pattern correctly.

This may be followed by additional commentary giving additional information about the pattern and how to use it. This will often include examples of the use of the pattern and what happens if you don't use it. Occasionally examples also appear in the forces section to emphasize the need for the pattern. We might also include references to related work and even contraindications to point to known situations in which the pattern definitely does not apply.

Finally, the patterns are marked with 0-2 asterisks. These stars indicate our confidence that the pattern is universal in some sense and whether or not it captures real truth. Two stars says that the authors think that something fundamental has been captured. Zero stars indicates a complete lack of such faith and in fact an assurance that things could be done better.

The Patterns

There are several different kinds of patterns here. Planning patterns help us think about the programming task as a whole. Stylistic patterns help your programs look nice and make them more readable. Design and Structural ones tell how to structure your code to aid maintainability and safety. Maintainability patterns help us write the code so it can be easily changed as the problem it solves evolves. Safety patterns help us prevent errors. Implementation patterns help us implement other patterns, such as the way in which Method Object helps us implement Composed Method. There are, perhaps, other classifications that will be developed as this paper grows.

Here is an alphabetical list of the patterns.

- Assign Variables Once (Design)
  Immutable Objects
- Be Spacey (Maintenance)
- Brace All (Stylistic)
- Braces Line Up (Stylistic)
- Comment (Only) When Necessary (Maintenance)
  Don't Repeat Yourself
  Express Your Intent
  Javadoc Public Methods
- Complete Interface (Design)
  Let the Client Recover

Separate Levels of Concern

- Composed Method (Implementation)
- Consistent Capitalization (Stylistic)
- Consistent Naming (Maintenance)
- Function For Complex Condition (Structure)
- Indent for Structure (Stylistic)
- Initialize (Safety)
- Intention Revealing Name (Maintenance)
- Local Variables Reassigned Above Their Uses (Design)
- Locals (Only) When Needed (Design)
  Save Your Names
  Don't Recompute
- Logical Interface (Maintenance)
- Method Object (Implementation)
- Name Your Constants (Maintenance)
- One Service Per Class (Structure)
- One Statement Per Line (Stylistic)
- Optimize for Readability (Maintenance)
- Pair Programming (Maintenance)
- Piecemeal Growth (Planning)
- Polymorphism Before Selection (Structure)
- Private Fields (Safety)
- Public Features First (Stylistic)
- Rewrite (Planning)
- Say "this" (Maintenance)
- Say It Once (Structure)
- Short Lines (Maintenance)
- Short Method (Structure)
- Spell It Out (Maintenance)
- Strong Encapsulation (Safety)
- Time on Task (Planning)

Other Java Idioms (http://c2.com/cgi/wiki?JavaIdioms). Some of the idioms here repeat or reinforce what we say here. Others go beyond it.

---

(1)Rewrite (Planning)**

Your programs are a kind of literature. They will be read by many people. If they are

important and valuable programs they will live a long time and be read by many, many people. They will become the basis of a small industry. Literature is never created in one go. Authors rewrite constantly to achieve beauty and clarity of thought and expression.

Therefore, plan on rewriting your program several times. Rewrite it at least once before you show it to anyone.

And read it between rewritings.

Think about how it reads. Think about the names you have used. Do they convey the appropriate concepts. Are your algorithms clear? Correct? Have you the appropriate comments to express your intent? Does it look good on the screen and on the printed page?

Would you be proud to have your name associated with it? To show it to your mother?

It has been said (Dick Gabriel) that all successful large programs grew out of well written and designed small programs.

Note that there is a formal notion of rewriting called refactoring. You can read more about this at http://extremeprogramming.org/rules/refactor.html

---

(2)Piecemeal Growth (Planning)**

Large programs are hard to write correctly, especially in one go. It is easer to modify a good small program into a better and bigger program than it is to write the bigger program from scratch.

Therefore, if you need to write a large program, consider what small program inside it you can get working first.Then expand on that core until you have the whole job done. The first piece can be fairly large or not, but it should be useful by itself. What small part of your overall program is the key to the problem as a whole?

Piecemeal growth is actually the key thing that object-orientation enables. It is much more important than reuse. It is easy to add classes to projects, especially when those classes add new services. So, start with the key services needed in your program and implement these. Even if you first write only the simplest cases of the key services you will have made an important start. More sophisticated versions of the first services you write might be developed through inheritance, or maybe not.

It is useful if the first part you write is the most important or most critical part. In large projects it is useful to work in the order of "value to the client", with the most valuable parts done first. This way the client has useful functionality early.

For more on getting started, see Kent Beck's Early Development pattern language on the wiki at: http://c2.com/ppr/early.html. You can learn more about an important new development methodology called Extreme Programming that depends fundamentally on this idea at http://extremeprogramming.org/ and at http://xprogramming.com

---

(3)Time on Task (Planning)**

In the real world, you will need the ability to quickly estimate how long it will take you to perform a given development/programming task. Most software development methodologies depend on this ability. This is a skill that can only be learned with practice.

Therefore, whenever you are given a development project by an instructor estimate how long it will take you to do it before you begin and then measure and record the time as you proceed. Keep a permanent record of the tasks, estimates, and actual times in a notebook that you use just for this purpose. You need the tasks recorded so you can look back when you get a new project to see if it seems to have a similar difficulty to one you worked on previously.

You need to keep a permanent record that you can refer to when estimating in the future. The difference between estimates and actuals should improve over time, but you can use these differences in other ways as well. If you are given a new task that seems about as hard as one you have done before, you can use the old actual time as the estimate of the new project. If your differences stay fixed, you know how optimistic or pessimistic you are in estimating.

Be honest with your estimates and be accurate with your actuals. This is only for your use.

Estimating is hard even with practice. It is easier to estimate small things and add up the estimates than it is to estimate big things. Therefore when you are estimating, break up the task at hand into small pieces that you can build one at a time and estimate these instead of trying to estimate the whole task. Record the individual times and the breakdown into tasks.

At the end of each project try to say why you differ between estimated and actual time and record this in the notebook as well.

---

(4)One Service Per Class (Structural)**

A program is easiest to maintain when it is made up of small, simple parts. The more complex a part gets the harder it is to understand and maintain. Classes can often best be thought of as defining services for the users (clients). The public methods of the class define the services.

Therefore, give each class one job to do; one service that it presents to its clients.

This doesn't mean just one public method. It might require several. For example, a Symbol Table in a compiler needs methods to insert information and to retrieve information. But it has just one job. It serves as a repository for information about the programmer defined identifiers in a program.

---

(5)Polymorphism Before Selection (Structural)**

Good object-oriented programs actually use relatively few if statements and very few switch statements. This is because polymorphism, both that provided by inheritance, and in Java, that provided by interfaces, provides a better, cleaner, and more maintainable way to make decisions in your programs.

Therefore, organize your code so that most decisions are made via the polymorphic nature of the system, not with if and switch statements.

You need to be aware, however, that only methods are polymorphic. While Java will permit you to define a field in a subclass that has the same name as a field in a superclass, doing so is almost always wrong. It results in two variables with the same name and the type of the reference is used to choose which will be used--even by a polymorphic method. This will usually result in very hard to find errors in your programs if you have polymorphic methods that refer to these variables. The fact that Java permits it is probably a flaw in the Java design, actually.

See also the similar advice in http://c2.com/cgi/wiki?InheritDontBranch

See also Selection and Polymorphism. Examples and commentary can be found in the following notes.

[Bergin and Winder] Understanding Object Oriented Programming

[Bergin] Moving Toward Object-Oriented Programming and Patterns

---

(6)Short Method (Structural)**

People have a limited attention span and a limited ability to deal with simultaneous detail. If methods are short a person can read and usually understand them quite easily. If they are short and clear they may not even need to be commented for the intent to be clear.

Therefore, your methods should usually be short. Ten lines is actually quite long.

This is easiest to arrange if each of your methods does only one thing. It may take a few statements to do that thing, but usually one task per method is the right idea.

There are only a few exceptions to the use of this pattern. If the logic of the method is all sequential (no if, while,...) and if it is doing the same thing to a sequence of data (initialization sequences for example), then you may not need or want to apply this pattern.

On the other hand, if your method is long because it has a switch statement with lots of options, you should really see if you can't use polymorphism to make the switch go away altogether. See Polymorphism Before Selection.

---

(7)Composed Method (Implementation)**

You are writing a method and it is getting long. You realize it should be a Short Method. Often it does several things, though they may be related to each other.

Therefore, break up long or complicated methods into parts. The parts are implemented as private, or possibly protected, methods. The original method becomes a sequence of method calls to the parts you have factored out.

The information that a factored part uses become the parameters of the new method.

Each part that you factor out needs to stand alone on its own. You need to be able to give it a short and descriptive name.

Factoring out loops into separate methods is also a very good idea. (This is mentioned in Gabriel's Simply Understood Code also.)

If the factored methods are protected rather than private, you will give yourself opportunity for specialization and hence polymorphism in subclasses. This is worth remembering when you are deciding what to factor. What might change in a more specialized subclass?

[Beck SBPP] and on Wiki at: http://c2.com/cgi/wiki?ComposedMethod

---

(8)Method Object (Implementation)*

You are trying to factor a long method using Composed Method. However, one of the sections that you want to factor out into its own method requires a large number of variables since it uses a lot of information. This would result in a new method with a large number of parameters. If there is a lot of this then the final composed method would be hard to follow.

Therefore, apply the following process to break up the long method.

1. Create a class to represent the section of the original method that you intend to factor out. The class needs a method called compute, with no parameters. The class will have one private field for each piece of information that is needed by the section of code that you are factoring out. It may need an additional parameter for the "this" of the original method if that is used
2. The constructor of the class will have one parameter for each of its fields. That is, the constructor will completely initialize the objects it defines.
3. The section of the original that you factor out becomes the body of the compute method of the new class.
4. The section of the original method now becomes the creation of an instance of this new method object and a call to its compute method.
5. Now you may use Composed Method on the new compute method itself to get a better factoring. These new factorizations become additional (private) methods in the new Method Object class.

The compute method, of course, has access to all of the fields of the object, hence to all of the information needed. The same will be true of the parts if you are able to apply part 4 of the rule.

[Beck SBPP] and on Wiki at: http://c2.com/cgi/wiki?MethodObject

Note that in C++ this is known as Function Object and is fundamental to the Standard Template Library.

---

(9)Say It Once (Structural)*

If you have to repeat the computation of something in your program you are both wasting your time and also building in opportunities for errors. When something that should be said once is said more often, then the opportunity occurs that the multiple expressions will become insonsistent with each other as the program grows and is modified.

Therefore, say each thing in your program just once.

To do this well, your program will be made up of lots of little pieces. Probably your classes will be small and certainly your methods will also be small.

If you find yourself rewriting some code that looks familiar, see if it already appears in your program. If it does, see if this code can become a method of some appropriate class, or the foundation for a new class.

Object-orientation and especially polymorphism helps with this. In procedural programming conditions normally have to be repeatedly tested to determine what to do. As the program grows these tests need to be re-written. In well written object-oriented programs, polymorphism is used instead of these tests and the different behaviors are built into different objects, each adhering to this rule.

I think this is due to Beck. See the Wiki: http://c2.com/cgi/wiki?OnceAndOnlyOnce

Note that this principle applies to computations that are necessarily the same. It does not mean that every use of x++, for example, has to be collected together, of course. Things that are only accidentally the same should be separate. Things that need to be the same should be said once and only once.

---

(10)Function For Complex Condition (Structural)*

If you are writing a selection or other structured statement that uses a boolean condition, the easy understanding of the meaning of the condition is critical for understanding what follows. Complex booleans with many parts connected with && and || can be hard to grasp quickly.

Therefore, any complex condition should be written as a separate method, expressing a positive condition. Then call the method in the if or while statement instead of writing out the complex condition.

If the method name is well chosen the intent will be clear without comments. By the way, methods that return booleans are often called "predicates."

For example:

```
if(reactorError() || (transmissionError() && heatRising()))
{       stutdownReactor();
}
else
{       fullPower();
}
```

If we write a function to capture the condition, this becomes

```
boolean shutdownRequired()
{       return reactorError() || (transmissionError() && heatRising());
}
...
if(shutdownRequired ())
{       stutdownReactor();
}
else
{       fullPower();
}
```

[Astrachan and Wallingford: Loops]

---

(11)Complete Interface (Design)**

When you write a class, its public methods define the services that this class provides to its clients. If clients find that there is some aspect of this service that they don't have access to, then they will be frustrated and will look elsewhere for solutions.

Therefore, provide a set of public methods in your class that permits the client software to perform all legal transformations. Make sure that the clients will have access to all needed aspects of this service (but nothing more).

For example, suppose you are writing a Stack class (not really needed in Java, which has a complete one in java.util). You know about the push and pop methods of course, as these define a class. A top (or peek) method that retrieves the top element in the stack without altering the stack contents is also very useful. However, critical to the correct use of a stack is the isEmpty predicate by which the client can know if a pop is legal. Without this the processing using the stack will be awkward at best, with the client always needing to keep track of something so that a pop isn't issued in error. Likewise if the stack contents are bounded (such as with an array implementation) you also need an isFull predicate.

You should also be aware of the needs of every class if it is to act smoothly within the Java system. There is a Java Canonical Class Form that describes what needs to be implemented in ANY Java class. See http://csis.pace.edu/~bergin/patterns/CanonicalJava.html

A consequence of this rule is that you also need to consider what to do when the client does issue an instruction that cannot be carried out. Usually the worst thing you can do is to write a message to System.out. There are many reasons for this. First is that you limit where your class can be used to code in which writing to output is possible and appropriate. A pacemaker probably needs a stack in its software, but doesn't have a printer or screen attached so that you can see these messages. What would the pacemaker wearer do if there was such a device and a message showed up on it that said "Stack Underflow"? More importantly, simply writing to System.out doesn't let the client that sent the improper message recover from the error.

Part of the complete interface of any class involves the ability of the client to recover from errors. Writing to System.out does not permit this to happen. In the case of trying to pop an empty stack in Java the correct response is to throw an Exception (a new NoSuchElementException, perhaps). Then, if the client is using an algorithm in which this situation cannot occur, there is no cost. But if the client is at risk of an "underflow", then he or she can anticipate this by catching this exception in a try block and take appropriate action.

The key message is that the Stack code itself can't anticipate what will be the correct response to an error in every case, so must make it possible for the client to make this determination.

This also requires that separate levels of concern (here normal processing vs. error processing) be handled with different mechanisms, perhaps. Here exceptions are recommended for error processing.

---

(12)Locals (Only) When Needed (Design)*

Good programs don't recompute the same thing repeatedly for two reasons. The least important today is that it is wasteful of the computer resource. More important is that if you compute the same thing in two places and the problem changes you may need to find all of the places in which you computed some item and make consistent changes. This is notoriously error prone. You can use a local variable within a method to capture the result of a computation and use it several times within that method.

On the other hand, readable programs are not overly wordy. If it takes you five lines of code to say what can better be said in 1 or 2 lines, your reader will not apprieciate you. If you use a local variable to save the results of every sub expression of a complex expression you will take a lot of statements to say what, perhaps, can better be said in a single statement.

Therefore, use local variables when they are used to avoid duplicate computation and otherwise avoid their use when the intent is clear without them.

For example,

```
FileReader fr = new FileReader("greatamericannovel.txt");
BufferedReader novel = new BufferedReader(fr);
```

is probably better written as follows unless the variable fr needs to be reused, in which case it needs a better name.

```
BufferedReader novel = new BufferedReader(new FileReader("greatamericannovel.txt"));
```

In Java, if you need to do a cast and use the "casted" result several times, capture the cast in a variable and then use the new variable.

In Java it is usually better if you declare your locals at the place at which you first use them and can initialize them, rather than collected together at the beginning of your method. This minimizes the searching your reader will need to do to find the types of your variables.

Kent Beck gives advice here on the Wiki: Caching Temporary Variable (http://c2.com/ppr/temps.html)

---

(13)Assign Variables Once (Design)*

Your programs will usually have several variables. At any given point in the program several may be in play at once. This makes it hard to understand what is going on at times since you don't know what the current values of your variables are. Your computations depend on the values, of course. If you execute the same statement several times (in a loop, for example) a

variable may have a different value each time. This can make it hard to reason about the correctness of your programs. Especially when there are lots of variables that "vary."

Constants, on the other hand are easy to reason about since they always have the value they were defined with. Our programs can become much easier to reason about and hence understand if we have to consider a minimum of data that actually varies.

Therefore, once you have given a variable a value, do not change it if at all possible. The value should preferably be assigned when the variable is declared.

A corollary to this rule is that **Immutable Objects** are a good thing. An Immutable is an object whose class will not permit it to be changed after it is constructed. This means the class must provide a constructor that provides values for all fields of the object and further, the class defines only private fields and no mutator methods for the fields. Once you create such an object and give it a "value," it always has that same value. Such objects are easier to reason about than objects that can change their state. An immutable object is like a structured constant.

Immutable objects are especially valuable when the program uses concurrency, since two threads can safely share references to an immutable without synchronization.

This pattern is from Gabriel's Simply Understood Code. On the Wiki, see http://c2.com/cgi/wiki?ImmutableValue

Of course, not every value can be immutable. Loop counters, for example only work because they CAN change. Likewise many of our objects are useful only because they encapsulate mutable state. An Employee class in which you could never change the currentSalary would not be very useful in most applications. However, not every class needs to be mutable.

---

(14)Local Variables Reassigned Above Their Uses (Design)*

Sometimes you must give a new value to a variable. In the presence of loops, the new value could be given either above or below its use. However, most people like to read programs from the top down.

Therefore, if you must change the value of a variable, arrange the code so that the change appears above any subsequent use.

This pattern is from Gabriel's Simply Understood Code.

---

(15)Private Fields (Safety)**

If your class fields are not private you won't be able to guarantee the maintenance of class invariants. This is a major cause of problems in large projects.

Therefore, all of your fields should be private. Provide protected accessor functions when necessary.

Remember that in Java, protected opens visibility (beyond the default package visibility), it does not restrict it. Subclasses within the same package will be able to see default visibility fields and methods. Protected opens visibility to those subclasses defined outside the current package. Sometimes it is appropriate for all code to be inside the package. In which case you don't need or want anything (except Main) to be public. Other times (library code) you want to provide services for other packages. Then you will need public and possibly protected methods. But your fields should always be private.

---

(16)Initialize (Safety)*

Every variable needs to have a value before you can use it. Java is very good about requiring this. Java even initializes some things for us automatically. Java also permits fields of classes to be initialized as part of their declarations, making constructors unnecessary for many classes. You want to think about what value each of your variables should begin with.

Therefore, when you declare a field or local variable, initialize it immediately.

Java permits quite sophisticated code to be executed as part of variable initialization. For example you can send messages to various objects as part of initializations. Take advantage of this and initialize as part of declarations, rather than in your constructors.

For example, the following are equivalent.

```
class Container
{               public Container()
                {               size = 0;
                }
                ...
                private long size;
}
```

We prefer the next form, however, since it initializes in the declaration. Here we don't even need the constructor.

```
class Container
{
                ...
                private long size = 0;
}
```

---

(17)Strong Encapsulation (Safety)**

If you reveal, through your accessor methods, the implementation details of your class, a client can take improper advantage of the knowledge, perhaps making it impossible to maintain invariants and certainly making a change in implementation difficult and costly.

Therefore, don't reveal the details of your implementation, even indirectly.

Generally this means that the accessors of your class should not return references to the variables used to implement the class.

The most common example in which revealing implementation appears is when the service

provided by the class requires that (perhaps among other things) it serve as a container for values. The clients need to access the things that are contained. You may choose to use a Vector as the container mechanism. Then, to provide clients access to the values, you have an accessor that returns the vector. This is poor practice for two reasons. Most important is that you lose control over what is in the Vector as any client can access it and modify it. Also, you lose control over changing the implementation to something else, like a Hashtable once clients take advantage of the fact that it is known to be a vector. Instead, you should return an Iterator or Enumeration over the Vector. Note that a Java 2 Iterator may actually not be safe to return, since it implements a remove method. Thus a client getting an iterator into an encapsulated Vector, could remove elements from it while iterating over it. This has the potential to invalidate your class invariants. In Java 2, however, you can return an immutable version of any container. This will give access to the contents but will prevent modification.

A general solution in situations like the above, is to define a Java interface for the thing returned (Enumeration is an interface, by the way). Then have the accessor return type be this interface. You are then free to implement with any class that implements this interface and your clients are free to take advantage of all of the methods defined in the interface, but nothing more.

For more on this, and an even stronger recommendation. See the Law of Demeter. http://c2.com/cgi/wiki?LawOfDemeter. What we have done here is not the Law of Demeter, but a consequence of it.

Notice that the same thing can happen if you pass an object into a constructor and the constructor saves a reference to that object in a field. Remember that the code that called the constructor has a reference to that object and can still modify it. If this is a problem, you should clone the object passed in and save the result instead. You can find a discussion of thse issues on Wiki at: http://c2.com/cgi/wiki?ReturnNewObjectsFromAccessorMethods

On the other hand, if the essence of a class is to define a container, then almost certainly it should be able to return the objects that it contains using some method. Stacks for example return their elements when popped. They don't return clones of the elements.

---

(18)Optimize For Readability (Maintainability)**

Also Known As: **Optimize Later** or even **Don't Optimize**.

Programs need to be correct before they can be fast. If they are to be correct, they usually need to be readable. If they are not clear to the reader, the reader will not be able to verify correctness. The more readable you can make your programs through naming, structure, correct use of comments and even formatting, the more maintainable they will be.

Therefore, make your code as clear and correct as possible, not as fast as possible.

Nearly everything you do to make a program fast, other than the choice of a good algorithm, will also tend to obfuscate it. A program that is faster than it needs to be has wasted other precious resources (programmer time, clarity,...).

Occasionally you will find, when nearing the end of a project, that it isn't fast enough. In this case get a good program profiler that will tell you where it is spending its time and optimize

only where the bottlenecks are and only as needed. Comment liberally in this case, perhaps leaving the original code in place (as comments) so that the next programmer on the project can see the intent.

Sometimes the best way to get better performance out of a program is to buy more or better hardware. This will make all programs run faster, not just the one being developed now. This won't help you if you are using an exponential running time algorithm when a faster one is available, but it can easily halve the running time of most algorithms.

Also see: http://c2.com/cgi/wiki?OptimizeLater

I first saw this in a book on Fortran by Henry Ledgard. It is supposedly due to Dijkstra ("Optimization is the root of all evil.")

Also note that modern compilers for languages like Java are much better at optimization than you are. Often a clear program, simply written, can be a better base for compiler optimization than one that tries to be clever and fast. This is because of both the speed of modern machines and the sophistication of modern compilers. A compiler can make tens of millions of decisions about how to optimize your program in just a few seconds. This is more than you can do yourself. Java has been called an excellent language for optimization, though current compilers for Java can be greatly improved.

The standard java interpreter has a profiling option (-prof) that you can use to find out where the program spends its time.

---

(19)Pair Programming (Maintainability)**

Programming is hard. It is easy to make mistakes and to overlook details. You want your programs to be of very high quality. Two minds are better than one. Teamwork is highly valued in the workplace, and it needs to be practiced.

Therefore, work in pairs using the following technique. Two programmers sit at the same screen/keyboard. One has control of the keyboard at any given time, but this control shifts back and forth. The one with the keyboard is typing in changes/updates to the existing code. The other is watching and thinking about whether it works, what tests you would need to execute to verify that it works, looking for small and large errors, and suggesting alternatives. If you think you have a better way to do something suggest it or ask for the keyboard to enter it yourself. Both members of a pair need to be totally engaged in the task and in each others thought processes.

The above focus on the actual programming part. You can also work in pairs to break a large project down into small parts. The parts should be small enough that each can be done by the pair at the keyboard in an hour or less. This includes time to test what you have done as well as write it. The parts can be written down on 3 by 5 cards and you can work from these at the screen. You can make notes on these about difficulties you encounter. You can also use these to record time estimates and time actuals. (See Time on Task).

There is evidence that two programmers working together get more quality work done in a given time period than two programmers working alone and then integrating their work. The important word here is quality.

Note to students: Only use this if your instructor approves. Many places this will be considered dishonesty. When in doubt, ask. Also, be fair in your use of the technique. Don't dominate the keyboard. Don't fade into the background and let your partner carry the load.

Note to instructors: You would be wise to enable this in your courses. Your grading may need to be different in order to make it so, but there are fairly simple techniques for this. For example, you can pair students differently on different projects and even though you give the students in a pair the same grade on each assignment, the better students, doing better work generally, will still rise to the top. Team work generally is highly valued in the workplace. This can also help the weaker students, as they have an easy way to get their questions answered. The pair exercise is almost a tutorial in some cases.

---

(20)Logical Interface (Maintainability)**

A program is most adaptable if it is can easily be thought of and read in the terms of the problem it solves, rather than the solution it provides. When you write a class you do so to solve some problem, and you use some internal implementation to represent the solution. If the only way to understand the class is to think of it as manipulating the implementation you will too often need to get mired in detail of the state changes. If, however, you can consider it in terms of the services it provides to clients and in the terms of the original problem, it is more likely that you can take an overview that obscures the detail so that you can get the "big picture".

The interface of a class (its public methods) gives the best overview of the class and what it does. This is the set of services it provides to its clients.

Therefore, the interface of your class should define its logical services, not its implementation. If your public methods consist (largely) of accessors and mutators for individual fields, you have probably not adhered to this rule.

In particular, if you have "designed" a class by first deciding what fields it needs and then providing accessors/mutators for these, you do not have the basis for a maintainable program. The client must then think in terms of your implementation, not in terms of the logic of the application. Worse, if the implementation changes, you will be tempted to change the mutators/accessors also, since you have developed this mind set about design. Then the client code will also need to change. This will perhaps result in further changes that will propagate throughout the code. This is a maintenance nightmare.

In Java, the interface of a class can actually be a Java interface. If you design this way, you will be less likely to fall into the implementation first trap.

Sometimes the best way to proceed is to examine how a typical client will need to interact with the class you are developing. You might even write some test code that the client might incorporate. This can give you a guide to a logical interface. The client won't want to know about the internals of your class and will want to express things in its own terms. However, don't try to anticipate all client uses or you may build something that is too specialized. In extreme programming such test code, written before the class is developed, is a fundamental part of the methodology. See http://extremeprogramming.org for example.

---

(21)Intention Revealing Name (Maintainability)**

It is helpful if our programs read naturally and don't need excessive documentation to understand them. One of the things that the programmer has most under his or her control is the names used to denote things. If names are correctly and carefully chosen our programs will read quite naturally and be almost like poetry.

Therefore, all of the names in your programs (variables and parameters, classes, methods) should denote the purpose of the item they refer to.

In creating a name, especially a method name, think how the name will look when used. Often long names, catenating many words are best, but this can be over done.

Flag (boolean) variables should be named for the condition they represent. Variables should be named for the objects they represent. In Java, the name does not usually include the type of the variable, though it might. And **NEVER** name a flag *flag*.

```
Stack myStack = new Stack(); // Poor name. How is myStack used?
Stack expressionStack = new Stack(); // Better. It will be used to hold expressions.
```

Also see the Consistent Naming pattern, below.

Special rule for visibility. The more visible a thing is the more care needs to be taken with naming it. For example, package and class names have wide visibility. They need to be carefully named. The names can be quite long. Local variables in (short) methods are visible over only a few lines of code. These names are not as important, but not unimportant. A simple indexing variable over an array of Vector might be just "i" if the loop used is short. Beyond that, think about the meaning of the thing the name refers to and choose accordingly. Even local names visible over a few lines of code can help the reader understand the intent of the code if the names are well chosen.

Special rule for parameters. Always take special care in naming parameters of your methods. The user of your class will depend on these to help understand what the method does. This is especially important when the method has more than one parameter of the same type. If you have several methods in your program that take the same kinds of parameters, then develop a style for which will come first and stick to your style. For example, if several methods manipulate a bank account that is passed in as a parameter, you might decide to always put this parameter first.

Special rule for accessors (getters) and mutators (setters). Accessors are methods that retrieve information from an object. Mutators send information to an object that they somehow retain for later use. The simplest case is when an accessor gives us the value of a field, for example. A poor style has developed lateley that we don't like to see. That is naming accessors by preceeding the thing to be accessed with the word "get.". For example, to access foo, some will define a method getFoo(). Likewise mutators are often prefixed with "set." This is a terrible style and leads to stilted reading of programs. You don't walk up to a new acquaintance and ask "John, what is your get last name?" or "Mary, what is your get income?" A function returning information should be named with a noun describing the information, not a verb phrase as here. See Consistent Naming.

Special rule for Java Beans. Unfortunately, Java Beans suggests that you prefix your accessors with get and your mutators with set. If you are writing Beans, you may want to do this. However, not everything needs to be a Bean, and in fact few programs need to be built

with a Bean (Component) architecture, though those that should be are very important. Also, you may use a BeanInfo class to define the accessors and mutators of a bean even when they have more suitable names than get... and set... Learn about BeanInfo if you are writing beans.

Special rule for the results of casts. Another special circumstance arises in Java. You sometimes have a variable of a given class. You somehow check and find that the variable actually refers to a value from a subclass so you can safely cast the variable. What do you call the resulting cast value? One convention is to choose a good name indicating the intent and then use a prefix of "checked" to indicate that the new reference resulted from a checked cast. For example, in the following, we cast the item retrieved from the Stack to an Expression knowing that we put only Expression objects into the stack. In the first fragment we have used the "throwaway" name temp which is not very good, event though it is temporary.The word temp is too generic.

```
private void generateCode(Stack expressionStack)
{       Enumeration expressions = expressionStack.elements();
        while (expressions.hasMoreElements())
        {               Expression temp = (Expression) expressions.nextElement();
                        temp.typeCheck();
                        temp.generateCode();
        }
}
```

Contrast the above fragment with the equivalent one which follows.

```
private void generateCode(Stack expressionStack)
{       Enumeration expressions = expressionStack.elements();
        while (expressions.hasMoreElements())
        {               Expression checkedExpression =
                                (Expression) expressions.nextElement();
                        checkedExpression.typeCheck();
                        checkedExpression.generateCode();
        }
}
```

Flags (boolean variables) should never be named flag. Variables should never be named variable. Methods should never be named method.

This pattern comes from Kent Beck [SBPP]

---

(22)Consistent Naming (Maintainability)*

What we name things in our programs greatly affects their readability. If a program is not easily readable, it can be painful to debug and modify. The language structure of the programming language we use as well as the language structure of the natural language (English,...) we use should be our guide to making programs readable. When you name something it is useful to consider how it will be used and how it will appear when used and sound when read aloud.

Therefore, use a consistent style for naming the things in your programs. The suggested standard is:

- Classes should have singular nouns for names: Stack.
- void methods should have verbs (or verb phrases)for names describing what they do:

openFiles().

- boolean methods and variables should have verb phrases using the verb "to be" and variations as the initial word: isFinished().
- Other non-void methods should have nouns (or noun phrases) for names describing what they return: sizeOfFigure().
- non-boolean variables should have noun (or noun phrases) for names describing what they represent: age.

A consequence of this is that accessor methods will not be named get..., etc.

---

(23)Name Your Constants (Maintainability)**

42 may be the answer to the question "What is the meaning of life, the universe, and everything?" but it can also be the number of departments in your company, the number of ..., whatever.

If the literal value appears throughout out your program and appears in different places with different meaning, it will be difficult to change when conditions change. Maybe 42 isn't a good example, but values like 1 and 10 are used for a variety of purposes even in the same program. If one of the uses requires that the value be changed, but the other uses do not, it may be very hard to determine which values need to be updated. Even if a constant is used only once in your program and can't be confused with another value that might be equal to it, you would still like a point of definition giving the intent.

Therefore, name any important constant values in your program and thereafter refer to those values only by name.

```
public static final int lifeTheUniverseAndEverything = 42;
public static final int departmentsInNYC = 42;
public static final double pi = 3.14159; // Better value in java.lang.Math.PI.
public static final int USStates = 50;
public static final int ageOfMaturity = 50;
public static final int legalAgeOfMaturity = 18;
...
double [] departmentBudget = new double[departmentsInNYC];
```

Now they can be easily changed easily, independently, and in one place and have an effect throughout a long program.

A common stylistic suggestion is to completely capitalize all constants. If we use this we would get the following instead.

```
public static final int LIFETHEUNIVERSEANDEVERYTHING = 42;
public static final int DEPARTMENTSINNYC = 42;
public static final double PI = 3.14159; // Better value in java.lang.Math.PI.
public static final int USSTATES = 50;
public static final int AGEOFMATURITY = 50;
public static final int LEGALAGEOFMATURITY = 18;
```

I PERSONALLY DON'T CARE FOR THIS STYLE. Oops. I personally don't care for this style. I guess it reminds me too much of the punch card machines of my youth. Others insist on this style to differentiate constants from variables. Of course, the above could be improved

with underscores:

```
public static final int AGE_OF_MATURITY = 50;
public static final int LEGAL_AGE_OF_MATURITY = 18
```

A good place to name constants relating to a single concept is in an interface for that concept. See http://c2.com/cgi/wiki?InterfacesForDefiningConstants

---

(24)Comment (Only) When Necessary (Maintainability)*

Program language notations (code) is often too small scale and fine grained to clearly indicate the intent of the programmer. Programming languages therefore provide a mechanism with which the programmer can insert natural language comments into the code to document this intent. Comments can be helpful or they can get in the way.

If your comments simply restate what the code says they are worthless. If your comments say something different from what the code says, they are less than worthless. If the code and the comments disagree they are probably both wrong (proverb). And remember that the code will change. Your comments can, however, say what the intended meaning of the code is supposed to be. They can help a reader adopt the correct mind set for examining the code and can also help a reader skimming the code understand it overall.

Therefore, make your comments indicate the intent of your programs.

The kinds of comments that are generally always useful are pre- and post-conditions on methods and invariants on classes. However, there are also important technical replacements for such comments. Both assertions and Java Exceptions can provide executable guarantees that your pre- and post-conditions are met.

For example, if you build a Stack class, and it has a pop method. Then a pre-condition for this method is that the stack is non-empty. This means that the client that uses the Stack must assure that the stack is non-empty before calling the method. On the other hand, the push operation has a post-condition that the stack is non-empty. This means that when the push operation terminates it guarantees that the stack is not empty.

For example, if you build a linked list class, its invariant is that every list is always properly terminated, either with null or with an equivalent *null-object.* A property is invariant if no public method will ever terminate with the invariant false, assuming the method was called under conditions in which its preconditions were true.

The following comment is worthless.

```
i++; // Increment i
```

The following is better.

```
i++; // Increment the card counter.
```

The following needs no comment (See Intention Revealing Name) and is therefore best.

```
cardCounter++;
```

This last example indicates that often the names you choose in your program are the best comments of all.

Usually comments on every line of a Java (or similar language) program are not a good idea.

They may be needed for languages like Assembler, however.

A comment at the beginning of a method or class detailing the intent of the code and any special needs is often useful. Likewise, if the code implements a well known algorithm (quick sort) then it is good to name the algorithm. Even better would be to have the name of the method capture all needed information.

One very useful way to comment a class and its public methods is to use Javadoc. These comments can be extracted into web pages (HTML) that can give the users of your class all the important information they need to utilize it. Here is a Javadoc example.

The name of this pattern indicates that it actually has two parts. Comment when you need to say something that you can say more clearly than the code. Don't comment if you can make the code clear enough by itself to be easily understadable without the comments.

---

(25)Spell It Out (Maintainability)**

If you use abbreviations in naming things you will likely spend a lot of time trying to remember whether you abbreviated or not the next time you need to type the name. You will find yourself scrolling to find the original declaration. If abbreviations are arcane or uncommon in any way your readers won't easily understand the intent of the thing named. The easiest thing to remember is the full name of a concept, not some particular shortcut for its name.

Therefore don't abbreviate your method and variable names. Use full words in the natural language in which you write.

Use abbreviations only if they are in common use in normal speech. The more visible a name is, the more important this rule. Class names and public method names have wide visibility.

---

(26)Short Lines (Maintainability)*

You want to be able to read your programs both on the screen and on paper. If you use long lines you may not see everything on one line without scrolling and some printers will truncate your lines when you print. Others will wrap the lines to the next lines destroying indentation in most cases

Therefore, keep your line length to 80 characters or less. This will accommodate even very old printers.

Some editors can be set to enforce this and most will give you some indication of the length of the line, usually on a status line.

---

(27)Be Spacey (Maintainability)*

You want your indentation to be preserved no matter how someone reads your programs. However, many printers and editors interpret tabs differently and most deal differently with

font faces that are not monospaced.

Therefore, use spaces for indentation, rather than tabs, and use a monospaced font in your documents when you have a choice.

Many editors will insert spaces when you hit the tab key. Look for this in the options.

---

(28)Say "this."(Maintainability)

You want to be able to tell, when reading code, where the various values were defined. Local variables of the current method have a different use and meaning than fields of the current object.

Therefore, when you refer to a field of the current object, prefix the reference with "this."

So, instead of referring to the size field of the current object as just *size*, use *this.size* instead.

Some would also say that you should do the same with all instance method messages of the current class as well. Then every message is explicitly sent to a particular object, perhaps "this." However, in Java, a message that is not preceeded by the name of any object can only be directed to "this" so it is perhaps less important for messages than for fields.

---

(29)Brace All (Stylistic)**

You are writing a selection or other structure and notice that some of the actions consist of single statements. The language doesn't require that you write braces or other grouping symbols in this situation.

However, you recognize that programs change as the problems that they solve change. In real programming this is a very frequent occurrence. If you have a single statement in an action, chances are that later it may need more statements.

Therefore, completely brace all statement parts in all structures when you first write the program.

```
if (measuredHeat() > subBoilThreshold) { shutDownGenerator();    }
```

can be modified more easily and with less possibility for error than the logically equivalent

```
if (measuredHeat() > subBoilThreshold) shutDownGenerator();
```

[Bergin: Selection]

---

(30)Braces Line Up (Stylistic)**

You are writing a structured statement that requires braces or other grouping symbols. You want your code to be as readable as possible.

When structures are nested, the indentation structure is often hard to follow. It is especially hard when the inner structures end and the outer structure resumes. The eye cannot always

easily see what goes with what level of the overall structure.

```
if(reactorOK())
{       if(transmissionOk()
        {       fullPower();
        }
        else
        {       reducedPower();
        }
}
else
{       shutDown();
}
```

The braces of a structure give its real intent, independent of how it is indented.

Therefore, when writing brace symbols or other grouping symbols such as parentheses, if the opening and closing symbol don't both fit on the same line, then make them line up exactly vertically.

```
if (measuredHeat() > subBoilThreshold) { shutDownGenerator(); }
```

Note that when the opening brace begins a line, you can put a full statement on that line as well, so that you don't waste vertical real estate. Only the closing brace is on a line by itself if you need more than one line.

Note that this pattern is at odds with the Java Standard Form in which you will find most published programs. This form hides the opening brace on the line on which the structure starts. This more typical style would have the opening brace at the end of the line on which it opens and the closing brace under the keyword that indicates the structure.

```
if(reactorOK()){
        if(transmissionOk(){
                fullPower();
        }
        else{
                reducedPower();
        }
}
else{
        shutDown();
}
```

We find the standard style more difficult to read and error prone.

The official Java coding standard can be found on the web at:
http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[Bergin: Selection]

---

(31)Indent for Structure (Stylistic)**

You are writing a structured statement using these (or other) patterns. You want to write readable code. In particular you want to indicate to your reader what the individual parts of your structure are.

The eye is good at grouping things. It is probably better at this than the mind is.

Therefore, the parts of a structure should be indented from the keywords and punctuation symbols that define its structure. All of the statements at the same level of the structure should be indented exactly the same amount.

Don't indent too much or you waste horizontal real estate. Don't indent too little, or the eye won't see the structure. See the code fragments above for examples of the use of this pattern. Also compare the following.

Too much indentation. Losing real estate fast. If the lines are long then you won't get to see the right ends of the lines properly.

```
if(reactorOK()) // Too much. Losing real estate fast.
{                               if(transmissionOk()
                                {                               fullPower();
                                }
                                else
                                {                               reducedPower();
                                }
}
else
{                               shutDown();
}
```

Too little indentation. The eye can't line up the structure if the sections are long.

```
if(reactorOK()) // Too little. Eye can't line up if sections are long.
{ if(transmissionOk()
  { fullPower();
  }
  else
  { reducedPower();
  }
}
else
{ shutDown();
}
```

Just Right.

```
if(reactorOK()) // Just right.
{       if(transmissionOk()
        {               fullPower();
        }
        else
        {               reducedPower();
        }
}
else
{       shutDown();
}
```

It is important that within a structured statement that all of the statements are indented the same amount. Consider the following two samples.

```
                        private void putForks()
        {               pickBeeper();
        pickBeeper();
                        turnLeft();
move();
```

```
                putBeeper();
        turnAround();
                move();
                move();
putBeeper();
        turnAround();
        move();
                                turnRight();
        showState("Think ");
                }
```

Contrast the above fragment with the equivalent one which follows.

```
private void putForks()
{       pickBeeper();
        pickBeeper();
        turnLeft();
        move();
        putBeeper();
        turnAround();
        move();
        move();
        putBeeper();
        turnAround();
        move();
        turnRight();
        showState("Think ");
}
```

The second sample indents consistently. The first is harder to read. (This example comes from "Dining Robots" in Karel J. Robot.)

[Bergin: Selection]

---

(32)Consistent Capitalization (Stylistic)**

Your programs should give both semantic (meaning) and structural information to the readers. Semantic information is conveyed by the names we use, but structural information is given in many ways. Often it is useful to know if a name seen in a program fragment is a variable, constant, class, or ...

Therefore, capitalize consistently to show the kind of thing a name refers to.

In Java, the conventional style is as follows:

- Always capitalize the first letter of any class or interface name: Stack.
- Never capitalize the first letter of any method or variable name: push(...).
- Constant names are completely capitalized: java.lang.Math.PI;
- Package names are written in all small case letters: java.lang.
- If a name is the catenation of several words, the second and subsequent subwords are capitalized: mustCopyToRunstack(...). Underscores are not preferred to separate words. Thus, mustCopyToRunstack is better than must_copy_to_runstack

Constant names are discussed in Name Your Constants.

---

(33)Public Features First (Stylistic)*

When you write a class, the most important use of the class description itself will be reading by other programmers who wish to use your class. They can only use the public features. The private (and protected) features are more specialized and are part of the implementation. It is awkward to have to read through things you don't need and can't use to find the things of use to you.

Therefore, list the public parts of your class first, and the private parts last.

Note that in Java, you don't need to define something before you use it, so there is no need to ever break this rule.

If you Javadoc your public classes and methods, then this becomes much less important. Here is a Javadoc example.

---

(34)One Statement Per Line(Stylistic)*

When you write a method, you want your readers to be able to easily follow your logic. Sometimes they will need to stop and think about it for a second. They need to be able to continue easily and gracefully. In some sense, programs are more like poetry than like prose. It is useful if there can be only one idea on a give line

Therefore, write your programs so that there is only one statement on each line.

A statement is supposed to represent a single idea. Contrast the following equivalent fragments for readability.

```
for(int i = 0; i < string.length(); ++i) { if( i % 5 != 0) { dest[i] += src[i]; } }
```

and

```
for(int i = 0; i < string.length(); ++i)
{       if( i % 5 != 0)
        {               dest[i] += src[i];
        }
}
```

Other Papers and Sources

Wiki Wiki Web http://c2.com/cgi/wiki The Wiki is a completely interactive web site at which readers can update the pages. It currently (Mid 2000) contains about 10000 dynamically generated web pages. It changes constantly. Spend some time reading before you start to edit though. It is a very valuable resource that should be treated with respect. It was created by

Ward Cunningham, who also maintains it.

Selection This paper [Bergin] has additional advice specific to if and switch statements. http://csis.pace.edu/%7Ebergin/patterns/Patternsv4.html

Polymorphism [Bergin] How to use polymorphism and how to look for opportunities to use it. http://csis.pace.edu/~bergin/patterns/polymorphism.html

Iteration/Looping [Astrachan and Wallingford] More advice on loops.http://www.cs.duke.edu/~ola/patterns/plopd/loops.html

Recursion: Wallingford's Roundabout pattern language is done in Scheme, but its ideas are also relevant to the Java programmer: http://www.cs.uni.edu/~wallingf/patterns/recursion.html

More Advice from Ward Cunningham can be found on Wiki at http://c2.com/cgi/wiki?MethodCommenting

Beck has more to say on formatting your code. This was written for the Smalltalk programmer, but is still relevant. http://c2.com/ppr/formatting.html

Kent has still more to say about how to think about starting on your project. http://c2.com/ppr/early.html. These ideas have been developed into a programming methodology called Extreme Programming. More on this at: http://c2.com/cgi/wiki?ExtremeProgramming. Extreme programming is itself a pattern language.

Books

[SBBP] Smalltalk Best Practice Patterns, Beck, Prentice Hall, 1997

[PLOPD 1] Pattern Languages of Program Design, Coplien and Schmidt (editors), Addison Wesley 1995

[PLOPD 2] Pattern Languages of Program Design 2, Vlissides, Coplien and Kerth (editors), Addison Wesley 1996

[PLOPD 3] Pattern Languages of Program Design 3, Martin, Riehle, and Buschmann(editors), Addison Wesley 1998

[PLOPD 4] Pattern Languages of Program Design 4, Harrison, Foote, and Rohenert(editors), Addison Wesley 2000

Advice similar to that contained herein may be found in the following book. Note that the advice is not identical, however.

Essential Java Style: Patterns for Implementation, Langr, Prentice Hall, 2000

A book that is somewhat different and focuses on structural issues is:

Refactoring: Improving the Design of Existing Code, Fowler, Addison Wesley, 2000

Acknowledgements and Contributors

Kent Beck and Richard P. Gabriel did a lot of work over a long period of time that is reflected here. Thanks to them especially. In particular is Kent's *Smalltalk Best Practice Patterns* and Richard P. Gabriel's Simply Understood Code (http://c2.com/cgi/wiki?SimplyUnderstoodCode).

The direct contributors to this page include:

Peter Andreae: Victoria University, New Zealand
Kent Beck: Three Rivers Institute
Byron Weber Becker: University of Waterloo, Canada
Larry Bliss, Pace University doctoral student
Kim Bruce, Williams College, MA
Max Hailperin: Gustavus Adolphus College, MN
James Heliotis: Rochester Institute of Technology, NY
Dorothy Nixon: Queens College, NY
Lynn Andrea Stein: Massachusetts Institute of Technology
Eugene Wallingford: University of Northern Iowa

Note, however, that the contributors don't necessarily agree with all that is said here.

The EuroPLoP 2001 shepherd for this paper is Manfred Lange of HP-Germany. He has greatly helped improve the presentation and helped me focus on the structure of this as well as made other helpful suggestions. In addition, he suggested One Statement Per Line. The paper is much nicer for his efforts. Thank you.

The main source of inspiration for this paper is the Elementary Patterns Working Group project. You can find the home page of this project at: http://www.cs.uni.edu/~wallingf/patterns/elementary/

If you are reading this on paper, the online version, which may be more recent, may be found at: http://csis.pace.edu/~bergin/patterns/codingpatterns.html

Last Updated: April 30, 2001 2:03 PM