Supplementary material for Chapter 2 of the McGraw Hill book: "Object Oriented Software Engineering: Practical Software Development Using UML and Java"

See www.lloseng.com for more information.

Programming Style Guidelines

An important part of developing good software is to make sure programs follow consistent guidelines that make them easy to read. In this book we use the guidelines described below; we recommend that you also follow these guidelines in the programs that you write.

Programming can be seen as the most detailed level of design, so the guidelines described here can be seen as design guidelines. Later on, in Chapter 8, we will discuss design at a higher level. You will then see that some of the principles mentioned here also apply to other aspects of design.

General principle: Remember that programs are for people to read

Although programs are executed by computers, almost all the guidelines in this section are designed to make them easier to read and understand by humans. Simpler programs save money because software developers are more likely to notice defects. Also, when changes are needed, it is easier to change simpler programs. A general principle therefore is:

- If you have two alternative ways of programming something and one alternative makes the code simpler, then choose that simpler alternative.

Corollaries to this are:

- Actively seek simpler alternatives, restructuring the code if necessary.
- Reject 'clever' or 'cool' coding techniques unless they make the code simpler to understand.
- Remember that shorter code is not necessarily better code, but unnecessarily long code is also bad.

The only exception to the above rules occurs when simplification requires a *significant* drop in efficiency. By significant, we mean that the efficiency drop will have a financial impact on end-users (slowing them down or forcing them to buy faster hardware) that will more than counterbalance the benefits of simpler code to the software developers.

Choose good names

We discussed naming of classes earlier in this chapter; the naming of all other elements of a program, such as variables, methods and packages, is equally important. Good names ensure that people can read the code easily.

- Always choose names for variables, classes, packages and methods that are highly descriptive of the purpose and function of the element.
- Do not worry about using very long names if the length is justified because it adds clarity.
- Avoid names less than about six characters, except perhaps for loop counter variables, where `i` and `j` are commonly used.

Comment effectively

Although code should be written as clearly as possible, it cannot always be made completely obvious. Comments are therefore essential to give readers an overview and to help them understand its complexities quickly. The following are some very general commenting guidelines:

- Comment whatever is non-obvious. Unfortunately, it is not always clear what is 'non-obvious. So err on the side of caution: Provide comments if there is any risk that someone reading the code may not completely understand some aspect of it. Remember that the audience for your comments will be other designers and programmers; as you write comments, try to imagine how they will think and what information they may need.
- Avoid writing comments that state the obvious, since they add clutter. For example, following the declaration `int foobar;` there is no need to say in a comment `/* foobar is an int */`.
- The comments in code should normally range between about 20% and 35% of the total length of the code.
- Write comments when you first write the code. In fact it is an excellent idea to write the comments *before* writing the code – writing comments can help you think and design the code effectively.

The following is a list of types of comments you should provide:

- Place a block comment at the top of each class describing the purpose of the class, how it should be used, its authors and its history of modification.
- Each method should also have a comment at its head describing its function and usage.
- Each non-obvious variable should have a comment.
- Loops and conditional statements inside complex algorithms should have comments. In general, readers of complex algorithms should be able to read the comments alone, in order to understand roughly what the algorithm does.
- Comment any changes to the code so that it is easy to see what has changed from one version to the next.
- Follow the specific conventions for commenting classes and methods that allow for documentation to be automatically generated using a program called 'javadoc'. See http://java.sun.com/javadoc for details.

Use a logical ordering within classes

Inside a class, order the elements as follows; within each group, start with the most public ones.

- Class variables
- Instance variables
- Constructors
- The most important public methods
- Methods that are simply used to access variables
- Private methods

We suggest using a horizontal line or some other comment to allow the reader to clearly see the divisions between the above sections.

- Keep related methods together, where this does not conflict with the above.
- If a class has many methods, group them into logical sections with a clear comment separating each section.

Pay attention to detailed layout

There are a number of different detailed stylistic approaches for how to lay out code within methods. The most important rule is:

- Be consistent in your approach to layout: Follow the same style throughout your code and follow the same approach as the other software developers who work in your company or team.

The web site http://java.sun.com/docs/codeconv/ provides a good set of layout principles. The following are some highlights.

- Always indent the contents of nested blocks carefully.
- Try to minimize the number of statements that take more than one line.
- When long statements are necessary, divide them into multiple lines such that the second and subsequent lines begin with an operator and are indented.
- Ensure that no line is longer than 80 characters, so that readers do not have to scroll right, and so that the code always prints correctly.

The following are some stylistic rules that we like to follow, but with which some people may disagree:

- Do not embed 'tab' characters in your code. Use two spaces for indentation. Tab characters can be fast to type; but when code is printed on certain printers, or displayed in certain editors, the width of the indentation resulting from the tab can vary and make the code hard to read.
- We use the following layout style for blocks:

```
if(condition)
{
  // statements
}
```

instead of the following alternative which some people, including Sun, prefer:

```
if(condition) {
  // statements
}
```

The first style ensures that the open and close braces always line up at the same level of indentation, at the cost of having a few extra lines of code. About half of the books we looked at do it the first way and the other half do it the second way. The most important point, however, is to be consistent in all the code you write.

Avoid duplication

It is a big problem to have the same or very similar code in two or more places. It increases the total volume of code and means that if you change the code in one place, then you might forget to change the code in the other places.

- Avoid *cloning* more than about one line of code. Cloning means deliberately copying code to use somewhere else. If you feel tempted to do this, you should normally create a separate method that has the common code, and call it from the original location and any other needed locations.
- If you find several substantially similar lines of code in several places, then normally you should write a single method to contain the code, and call it wherever necessary.
- If the duplication exists in two separate classes, then consider creating a common superclass (although stick to the rules discussed earlier that determine what constitutes a good generalization – such as the isa rule).

Adhere to object oriented principles

- Take full advantage of polymorphism and inheritance as well as abstract classes and methods.
- Ensure the 'isa' rule is religiously applied.
- Ensure that anything that is true in a superclass is also true in its subclasses.
- Avoid over-use of class variables or class methods. Wherever possible try to create designs that use instance variables and instance methods instead.
- Create several small classes, rather than one big, complex class.
- Keep the number of instance variables small. If this number exceeds 10, then consider splitting the class into separate classes – e.g. a superclass and a subclass.

Prefer private as opposed to public

Favouring privacy improves encapsulation, by ensuring that only programmers working inside a class (or inside a package) can use all of its facilities. This allows changes to be more easily made since one can be confident that 'outsiders' are not relying on too many details.

- Make variables and methods as private as possible. In other words, prefer **`private`** to **`protected`**; **`protected`** to the default package access, and package access to **`public`**.
- Unless there is a compelling reason to the contrary, declare instance variables to be **`private`**, and provide methods to access the private variables if necessary.

Restrict user interface statements to classes specifically designed for this

This guideline is one that we will re-visit in much more detail in later chapters. However, it is so important, yet so often remains unknown to beginners, that it needs mentioning now. Most of the classes in the system should do not interact with the user in any way (neither using windows nor using **`system.in`** and **`system.out`** objects). Most classes should simply store data in their instance variables, and provide methods for manipulating those instance variables. When you need to get information from the user, you should set up a separate set of classes to do this.

Other ways to simplify code

- In general, ensure that there is only one place from which a method returns to its caller; this should be the last statement. This rule can be violated if adhering to it would add extra indentation or several extra statements.
- Avoid too many levels of nesting (i.e. indentation), where possible. You should think carefully if the nesting level exceeds five – the code then becomes quite hard to understand.
- Divide up long methods into shorter ones. If a method exceeds about 20 lines, see if you can take part of the method and make it into a separate method that can be called by the original method.
- Split complex conditions. For example, imagine you had the following:

```
if(a==5 &&(b > 40 || c) && (d > a+2 || e==5))
```

A statement like this might be easier to read if the parts of the condition were placed on separate lines like this:

```
if(a==5
  && (b > 40 || c)
  && (d > a+2 || e==5))
```