Notes on programming and debugging

Charles Elkan
elkan@cs.ucsd.edu
January 28, 2009

These notes are based on the early lab assignments for CSE 15L in Winter 2009. Everything here is complementary to other guidelines on debugging using the scientific method. This document simply explains some additional strategies for better programming and for finding and fixing bugs.

When you are working on a piece of software (it can be a self-contained program or part of a large software system), do the following first, in order:

1. Make sure you understand the purpose of the code. For 15L assignments, this purpose is often explained in a usage message.

2. Make sure you have a high-level understanding of the algorithm being implemented to achieve the purpose. This is especially important for Lab 4.

3. Make sure you understand the purpose and initialization of the important variables used in the code.

For each of the items above, if comments are missing, write comments that record your understanding. For example, for the third item, for Lab 4:

    int d = 2; // first day of 2008 is Tuesday == 2

After the steps above, read through the code and think about everything that looks strange. When you are confident you see something wrong, make a change to fix it. At this stage, it is ok if you miss some mistakes, and it is ok if you occasionally make a change that does not fix a bug. The goal is to eliminate as many small
bugs as possible, without needing to apply the full scientific method individually to them.¹

For example, some versions of Lab 3 contain the line

```java
if (arg.charAt(0) == '-' && args.length > 1)
```

It is strange that the first part of the Boolean expression uses `arg` while the second part uses `args`. By noticing this strangeness, and thinking about what the line should be doing, you can immediately change `args` to `arg`, which makes a bug go away and saves you time later.

When you are reading through code, look for situations where the goal is to do something routine, and you already know a correct way to do it. For example, part of the code for Lab 3 reads a file until the end of the file is reached. The original code, which is buggy, looks like this:

```java
do {
    line = in.readLine();
    ...
}
while (line != null);
```

Part of the program for Lab 2 had a similar goal, and was also buggy. By searching the web, we found a well-known standard way for achieving this goal correctly:

```java
String myline;
while ((myline = in.readLine()) != null)
{
    System.out.println(myline);
}
```

¹The advice in the paragraph above contradicts a strict version of the scientific debugging approach. In debugging following the scientific method, you never modify the code first. Instead, first you identify a failure. Second, you form a hypothesis that a certain fragment of code is a defect that caused the failure. Third, you test this hypothesis. You only modify the code fragment last of all, after you have good evidence that the hypothesis is true.

So, when should you use the scientific approach to debugging, and when not? There are several possible answers to this question. One answer is that you should use the strict scientific approach in 15L, because the point of the course is to teach this method. Another answer is that you should use the scientific method when you are faced with a failure that is difficult to understand and difficult to fix. The advice above is primarily for small bugs. A third answer is that you should follow the advice above for code that you write yourself, while you are writing it. For software written by other people, that you are unfamiliar with, following the scientific method is advisable.
A standard way of achieving a common programming task is called a “programming pattern.” Good programmers use patterns over and over again, as much as possible. We can reuse this pattern for Lab 3:

```java
String line;
while ((line = in.readLine()) != null)
{
    
    }
```

Another principle that experienced programmers follow is to implement algorithms at the highest reasonable level of abstraction. A higher level of abstraction means a smaller amount of detail. As a programmer, you want to save your own time and not write code to cope with details, when you don’t have to. Equally important, when you don’t write code to perform low-level operations, you eliminate the possibility of bugs in the low-level code.

As an example of this principle, consider a version of the Lab 3 example from above:

```java
if (arg.charAt(0) == '-' && arg.length == 1)
```

This code is operating at the level of individual characters. It is higher-level to operate at the level of entire strings, when possible. We can rewrite the code fragment as

```java
if (arg.equals("-"))
```

This version is shorter, easier to understand, has less room for errors, and will be easier to fix if does contain an error. It is better in every way.

The principle of operating at the highest reasonable level of abstraction is important for Lab 4. Most of the code you are given for this lab operates on individual characters. You can make it shorter and more understandable by converting it to operate on integers instead.

Good programmers reuse programming patterns, and they also reuse code written by other people, if they know it is reliable. You can assume that methods implemented in standard Java libraries are reliable. When you have a task to perform, think whether it is likely that a library function exists for the task. If it seems plausible that a library function exists, then look for this function. For examples, to operate at the level of integers in Lab 4, you need to convert arguments that are strings into integers. Converting a string into an integer is a very
common programming job. So, it is likely that a Java library function exists for this task. Using a search engine, it is not hard to locate this function. It is called \texttt{Integer.parseInt()}. 

One of the harder aspects of programming is figuring out complicated Boolean logic: doing something if and only if a certain combination of conditions is true. For example, for Lab 3, the goal is to print out a line if and only if it is \textit{not} the same as the previous line, where the meaning of “the same” may or may not ignore upper/lower case differences. The original code, which does not work correctly, is

\begin{verbatim}
if (!line.equals(prevLine)
 || optIgnoreCase && !line.equalsIgnoreCase(prevLine))
{
  out.println(line);
}
\end{verbatim}

There are at least two useful approaches to solving this type of problem. The first approach is “divide and conquer.” Separate the job into parts, and solve each part separately. Of course, you need intuition and experience to identify a good division of a job into separate parts. Here, you can identify two cases based on whether \texttt{optIgnoreCase} is true or false. This gives the programming solution

\begin{verbatim}
if (optIgnoreCase)
{
  if (!line.equalsIgnoreCase(prevLine))
    out.println(line);
}
else {
  if (!line.equals(prevLine))
    out.println(line);
}
\end{verbatim}

Note that the “then” part and the “else” part of the outer “if” command are both surrounded by braces. It is good to use braces even when they are not strictly needed, to make clear which inner commands go together, versus which don’t.

A second approach to solving a complicated logical problem is to reduce the complexity of the Boolean expression. In the example above, the negations (which are indicated by exclamation marks \texttt{!}) are confusing. We can reduce complexity by eliminating the negations and thinking positively: if \texttt{optIgnoreCase} is true, and \texttt{line.equalsIgnoreCase(prevLine)}, then do nothing; similarly, if \texttt{line.equals(prevLine)} then do nothing. This thinking gives the solution

\begin{verbatim}
if (!line.equals(prevLine)
 || optIgnoreCase && line.equalsIgnoreCase(prevLine))
{
  out.println(line);
}
\end{verbatim}
if (line.equals(prevLine)
    || optIgnoreCase && line.equalsIgnoreCase(prevLine))
{
    // do nothing
}
else out.println(line);

Note that there is no command inside the “then” part of the “if” command. Because it is unusual for a “then” part to be empty, it is good practice to include a comment that makes clear that this behavior is intended.