ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 8 — January 22, 2013

Patrick Lam *version 1*

Debugging Strategy

We can use the scientific method as a strategy for debugging. Here's an adaptation of the scientific method, from Zeller's book:

1. Observe a failure (i.e. as described in the problem description).
2. Invent a *hypothesis* as to the failure cause that is consistent with the observations.
3. Use the hypothesis to make *predictions*.
4. Test the hypothesis by *experiments* and further *observations*:
   - If the experiment satisfies the predictions, refine the hypothesis.
   - If the experiment does not satisfy the predictions, create an alternate hypothesis.
5. Repeat steps 3 and 4 until the hypothesis can no longer be refined.

Once you have a cause for the failure, then you can also fix the failure by modifying the program.

The more explicit you are about the various products (what's the failure? what's the hypothesis?), the easier you'll find it to fix the problem.

Bug Localization. A key part of the hypothesis underlying a bug is the location of the bug in the code. This is especially true for larger software, and should be the first step in your debugging strategy.

You should attempt to localize the bug to within a subsystem or a set of modules. You may have to do this iteratively, continuing to localize the bug to narrower and narrower sets of modules. Once you know *where* the bug is, you'll find it to be easier to refine your hypothesis about *what* the bug may be.

Debugging Tactics

You can combine the general strategy with the tactics I'll present here to debug programs. These tactics are applicable to many languages, but I'll tell you about Java implementations. There are three general kinds of tactics that you can use to debug your code.

- *Code review*: I find this to be the most effective technique. Stare at your code and think hard about what it's supposed to be doing, compared to what it's actually doing.

- *Code instrumentation*: Put `print` statements (or equivalent) into your code. Create hypotheses about what the program is doing, insert code to get more observable outputs, and verify your hypotheses by running the program.
- *Single-step execution*: You can also step through your code in a debugger line-by-line and manually inspect the program state. I find this to be not-so-effective, because it's too low-level.

Tactics for bug localization. To formulate a hypothesis about the location or identity of a bug, you must collect sufficient diagnostic information using the above tactics: you might supply different inputs, run the instrumented program, set breakpoints, and examine internal state. Note that doctors do this—they send a patient for tests so that there is sufficient diagnostic info to formulate a hypothesis about the underlying cause.

Code instrumentation

Beyond `Log.d()` for getting information out of your program, you can rely on assertions.

Assertions. An assertion is a statement about the world; for instance, I assert that PowerPoint is often harmful to education. In the context of code, assertions are logical expressions that should always be true. When the program executes an assertion, it verifies that the logical expression indeed evaluates to true. If not, then it throws an `AssertionError`, which will usually stop the program. You can also supply a second parameter to the assertion, containing a message to be reported with the error in the event of assertion failure.

Two examples of assertions:

- `if (i % 2 == 0) { ... } else { /* i is odd */ }`: we know that `i` is odd, so use an assertion to document this, e.g
  ```
  if (i % 2 == 0) { ... } else { assert i % 2 == 1; }
  ```
- When implementing a doubly-linked list, you know that following `next` and then `prev` should get back to the original node. Include:
  ```
  assert this.next.prev == this : "List fails doubly-linked node invariant";
  ```

Assertions should never have side effects.

Single-step execution

Single-step execution can help you both with localization and hypothesis testing. Since programs can be huge, you don't want to run through the whole program every time. Hence *breakpoints*. You can run the program until it hits the breakpoint and then single-step.

Sprinkle breakpoints throughout the program, before lines that you suspect to be faulty. Then single-step across the questionable lines, inspecting (and perhaps modifying) relevant program state both before and after the line. If the state is infected, then you've isolated the bug.

Kinds of Breakpoints. Eclipse supports a variety of different breakpoint types[1], although most of the time a Line Breakpoint will suffice.

- *Line breakpoints* halt program execution at a particular line. (You can also make them conditional: only halt program execution when a particular condition is true; or when a particular value changes.)
- *Exception breakpoints* halt program execution and load the debugger when the program throws a particular exception.
- *Watchpoints* halt program execution when a particular memory location changes (or is accessed).
- *Method breakpoints* halt program execution upon entry or exit from a particular method. (Entry is easy to simulate with line breakpoints, but exit is potentially hard for large methods.)

It's easy to waste a lot of time fiddling with the debugger. Debuggers work best if you know what you're looking for ahead of time, and have a specific experiment in mind for a particular debugger invocation. Otherwise you're likely to just waste time. Single-stepping may be useful as you're gaining experience with software development, but tends to be generally less useful as you get better as debugging.

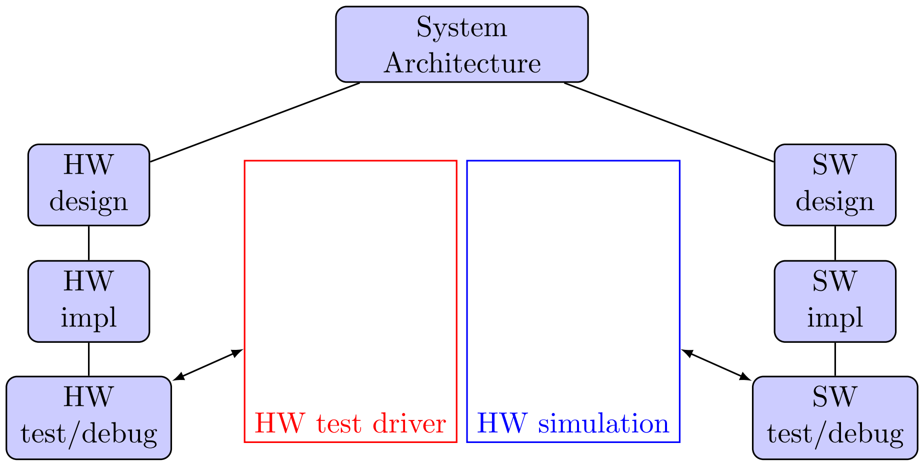
Debugging for Embedded Systems

Here is some additional about debugging embedded systems.

First step. Debug as much as you can separately, notably the hardware and software. This may require:

- for software debugging: a hardware simulator;
- for hardware debugging: test drivers and a test harness

Consider the following diagram:



[1]http://www.eclipse-tips.com/tips/29-types-of-breakpoints-in-eclipse

Debugging Challenges. After you've integrated the hardware and software, you'll face more challenges:

- *fault (bug) localization*: Is the bug in the hardware or in the software? If it's in the software, which module is the problem? If it's in the hardware, which component is at fault?
- *need for instrumentation*: You'll need to add instrumentation to get diagnostic information. Print statements probably won't work, so you'll need to find some way of getting debug information from the software. And you'll need to figure out how to determine the state of the hardware. At the HW/SW interface, you can use a *logic analyzer*.

  Plan ahead and figure out how you'll instrument the system! (This separates the adults from the kids).
- *software breakpoints are hard*: If you use a breakpoint, then it'll stop the software, but the embedded hardware continues to run (e.g. think about the previous students' LEGO NXT robots). Ideally, when a software breakpoint occurs, "freeze" the hardware; this requires special support, which is hard to do.

Debugging of Field Problems. Once you've deployed your system, users will always find more problems. How do you diagnose them? This is a major challenge; you'll need to figure out how to get initial diagnostic information. In aviation, the black box is one way of getting diagnostic information from crashed aircraft. Windows allows programs to upload diagnostic information upon crashes.