Course "Debugging"

Using debugger software

Prof. Dr. Lutz Prechelt
Freie Universität Berlin, Institut für Informatik
http://www.inf.fu-berlin.de/inst/ag-se/

- Basic concepts
  - Debugger
  - Breakpoint
  - Conditional breakpoint
  - Inspection
  - Manual control
- Demo
- Usage modes
  - Postmortem analysis
  - Program understanding
  - Hypothesis testing
- Methodological hints

Basic concepts: Debugger

- Definition "debugger":

  A debugger is a program execution platform that allows
  - pausing program execution at specific points
  - inspecting the values of program variables
  - modifying the values of program variables
  - evaluating arbitrary expressions
    and calling arbitrary routines

Basic concepts: breakpoint

- A breakpoint is a pseudo instruction that the user can insert at any place into the program at debug time.
- When execution crosses a breakpoint, the debugger will pause program execution
  - The user can then
    - inspect variables,
    - set or clear breakpoints, and
    - continue execution
- Implementation
  - Conceptually, the execution is controlled by the debugger and the debugger will interpret the breakpoints.
  - Technically, the debugger indeed modifies the existing program code
    - and inserts a 'stop' instruction, but keeps the old instruction around

Basic concepts: Conditional breakpoints

Freie Universität Berlin

- A generalization of breakpoints:
  - When a conditional breakpoint is crossed during execution, the program will pause only if the breakpoint's break condition holds

- Types of break conditions:
  - A given expression is true
    - e.g. "stop when *counter* == 255"
  - The breakpoint has been crossed N times ("hit count")
    - e.g. "stop after 99 iterations"
  - A given expression has changed its value
    - (sometimes also called 'watchpoints')
    - e.g. "stop if *filename* has been modified"

Basic concepts: Inspection

- A debugger allows inspecting the contents of program variables:
  - show contents of primitive variables
  - show fields of compound variables (records, arrays)
  - follow pointers/references

- These contents can also be changed
  - maybe even new objects created at debug time

Basic concepts: Manual control

- Many debuggers allow manual procedure calls
  - when the program is stopped, it continues not with the next instruction, but rather with a call typed by the user
  - to be useful, this also requires that the debugger can evaluate most kinds of expressions

Debugging and Java

- Java Debuggers tend to be more powerful than those for languages that translate into machine code
  - Java bytecode always contains the names of all classes, methods, and attributes
    - A 'debug' compiler option is needed only for keeping the names of local variables in the code
    - In many languages, e.g. libraries cannot be debugged as well
  - Java's strong typing always provides data semantics
    - Compare this to e.g. **`void*`** in C
  - Java Reflection naturally allows for inspection and manual control
    - look at *java.lang.reflect* if you have not heard of reflection
  - Exception objects provide rich information (stack trace) still *after* some event
    - makes many problems much easier to catch *without* a crash

Basic demo

- Who has never used a debugger before?

- Demo (program EnglishNumber; inputs 2006, 6509213)
  - single-stepping
    - implicit breakpoints: into/over/to-line/return
  - setting a breakpoint
    - normal, count, Exception
  - inspecting variables
    - inspect, lib innards, watch
  - modifying variables
  - setting a watchpoint
    - access, modify

- (Realistic programs have very different properties)

Kinds of debugger programs

- Capabilities and usage details of debuggers vary wildly
- The basics, however, are always the same:
  - breakpoint, single-step
  - inspecting data
- The major differences are:
  - works on machine code <u>or</u> on source code
    - and then: language-dependent <u>or</u> not
  - is ignorant of parallelism <u>or</u> knows about threads
  - can only inspect memory <u>or</u> can evaluate expressions
  - cannot handle conditional breakpoints <u>or</u> can
  - runs on the application host <u>or</u> runs remotely
    <u>or</u> is based on an emulator
    - remote for embedded programs may require special hardware
  - has a text-based user interface only <u>or</u> a GUI

The Eclipse Java debugger

- Is a most advanced kind of debugger:
  - works on the source code level
  - knows about threads (even displays them in a nice way)
  - can evaluate expressions
  - can handle conditional breakpoints
  - runs on the application host or remotely as needed
  - has an advanced GUI
  - is integrated with the editor environment

Debugger usage modes

- Postmortem analysis
- Program understanding
- Hypothesis testing

And non-modes:

- Program tracing

Mode 1: Postmortem analysis

- When a program that is run in a debugger crashes, the debugger is used to understand the reason for the crash
  - This analysis is done after the fact ("post mortem", "after death")
  - It is based on analyzing
    - which calls in the program produced the call nesting that was active at the time of the crash
    - the variables in these active calls

- Advantage:
  - Postmortem analysis is simple and possibly quick

- Disadvantage:
  - One cannot analyze how the program arrived at the program state that produced the crash

Postmortem analysis: Method hints

- Portmortem analysis can go bottom-up or top-down
  - Both require first understanding what the program was just trying do to (globally). Then:
- Bottom-up:
  - Find the faulty data element in the innermost method call
  - Try tracing through surrounding calls to the origin of the state fault
- Top-down:
  - Check the state of each call, beginning with the outermost
  - Compare the states to your expectations
- Tradeoff:
  - Bottom-up is often preferable, because it is highly focused
  - But top-down may be better if the state is complex and the crash was many calls away from the original error

Postmortem analysis: Technical issues

Freie Universität Berlin

- Successful use of the debugger usually relies on debug information inserted in the program during compilation
  - Understand the respective compiler switches
  - Crucial for most languages translated into machine code

- Some systems produce a 'dump file' when crashing
  - This contains all or part of the program state
  - so that a postmortem can be done even if the debugger was not running.
  - Understand the respective operating system settings

Mode 2: Program understanding

Freie Universität Berlin

- A debugger helps understanding code one does not know well
  - Choose simple program inputs first
  - Single-step through the program
    - Step into methods of interest; step over the rest
    - Inspect variables as required
    - Use a breakpoint to finish loops you have understood
- Advantage over purely reading the program:
  - No thinking errors with respect to control flow
  - Computer keeps track of variable values for you
- Limitations:
  - Difficult to apply for non-sequential programs
  - Not straightforward for GUI programs
    - Must set breakpoints at handlers or stop event handling thread

Program understanding: Method hints

- Don't try to understand everything at once
  - Formulate specific questions
  - Work in multiple passes
- Start with a rough top-down understanding of the program design
  - Try to get used to the terminology and naming conventions
- Switch to mode 3 in due time
  - Maybe switch back and forth

Mode 3: Hypothesis testing

- A debugger helps gathering information about specific aspects of a program without modifying the program
  - No need to insert analysis code, recompile, redeploy

- Kinds of information:
  - Program state at a certain point during execution
  - Validating specific expected events
    - their occurrence
    - their order
    - their frequency
  - Catching specific "impossible" (unexpected) events

Hypothesis testing: Technical issues

- Efficiently using a debugger for hypothesis testing requires mastery of its advanced features
  - conditional breakpoints
  - working with multiple threads
  - value tracing etc.

- If the program is part of a complex system
  - debugging may have to occur "in place".
  - In this case, remote debugging may be required
    - Learn how to set that up

Hypothesis testing: Method hints

- A common debugging error:
  - Not doing enough testing first and then going into debugging without hypotheses what goes wrong
  - This can result in needlessly spending hours in what is really mode 2 (program understanding)

- If it appears too cumbersome to evaluate multiple narrow hypotheses only, some automation often helps
  - But do not try formulating hypotheses if you do not understand the program well enough

General hints

- A debugger does not do the thinking for you
  - **You need a clear insight goal at all times**
  - Corollary: If you have narrowed down where the defect must be, but do not find it soon, stop using the debugger

- If the code is difficult to understand, transform it
  - **Simplify code that you cannot understand easily.**
    - E.g. use more intermediate steps, clearer names, etc.
    - Use the refactoring functions of your IDE!
  - Corollary: Write simple code

- Know your tools
  - **You need to be skilled enough with your debugger** that you can concentrate on the problem

The nine rules: How to use a debugger

Freie Universität Berlin

When applying this rule	***use this mode***
• Understand the system	Mode 2
• Make it fail	Mode 3, perhaps 1
• Quit thinking and look	Modes 1, 2, 3
• Divide and conquer	Modes 1, 2, 3
• Change one thing at a time	Mode 3
• Keep an audit trail	? (see next week: tracing)
• Check the plug	Modes 1, 3
• Get a fresh view	-
• If you don't fix it, it ain't fixed	Mode 3

Summary

- Basic concepts
  - Debugger
  - Breakpoint
  - Conditional breakpoint
- Usage modes
  - Mode 1: Postmortem analysis
  - Mode 2: Program understanding
  - Mode 3: Hypothesis testing
- General hints
  - A debugger does not do the thinking for you
  - If the code is difficult to understand, transform it
  - Know your tools

Thank you!