CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Interactive Development Environments

Feb. 27, 2007

Interactive Development Environments

- A system that covers many development tasks
  - Editor – usually with nice syntax coloring, indentation
  - Compiler – automatic compilation, errors linked to code
  - Debugger – step through source code
  - Etc... – Testing, documentation, search, code transformations, ...
- Examples: Eclipse, DrJava, NetBeans,, Visual Studio, emacs

Example IDE -- Eclipse

- Editing
  - Syntax coloring, auto-indent, brace matching
  - Integrated with JavaDoc
- Testing
  - Integrates with Junit testing framework
    - Uses suite() or auto-generated suite
  - Interaction panel allows interactive method invocations
- Debugging
  - Integrates with Java debugger
  - Interactions panel also useful

Debugging

- My program doesn't work: why?
- Use the scientific method:
  - Study the data
    - Some tests work, some don't
  - Hypothesize what could be wrong
  - Run experiments to check your hypotheses
    - Testing!
  - Iterate

Starting to Debug

- What are the symptoms of the misbehavior?
  - Input/output
  - Stack trace (from thrown exception)
- Where did the program fail?
- What could have led to this failure?
- Test possible causes, narrow down the problem

Checking that Properties Hold

- Print statements
  - Check whether values are correct
    - E.g., look at value of i to check if i > 0
  - Check whether control-flow is correct
    - E.g., see if f() is called after g()
- Automatic debugger
  - Allows you to step through the program interactively
  - Verify expected properties
    - Don't need to put in print statements and recompile
  - Use as part of testing

Interactions Panels (e.g., in Dr. Java)

- Can evaluate Java expressions interactively
  - Can bind variables, execute expressions/statements
- Benefits
  - Make sure that methods work as expected
  - Test invariants by constructing expressions not in program text
  - Combines with interactive debugger

Automatic Debuggers

- Set execution breakpoints
- Step through execution
  - **into**, **over**, and **out** of method calls
- Examine the stack
- Examine variable contents
- Set watchpoints
  - Notified when variable contents change

Using the Debugger

- Set debug mode to on
  - Turns on debug panel with state information
- Set break point(s) in Java source
- Run the program

Tips

- Make bug reproducible
  - If it's not reproducible, what does that imply?
- Zero-into smallest program that reproduces bug
  - Reveals the core problem
- Explain problem to someone else (i.e., instructor or TA)
  - Explaining may reveal the flaw in your logic
- Keep notes: don't make the same mistake twice

Defensive Programming

- Assume that other methods/classes are broken
  - They will misuse your interface

```
public Vector(int initialCapacity, int capacityIncrement)
  {
  super();
  if (initialCapacity < 0)
    throw new IllegalArgumentException(
      "Illegal Capacity: "+ initialCapacity);
  ... }
```

- Goal: Identify errors as soon as possible

Avoiding Errors

- Codify your assumptions
  - Include checks when entering/exiting functions, iterating on loops
- Test as you go
  - Using Junit
  - Using the on-line debugger
- Re-test when you fix a bug
  - Be sure you didn't introduce a new bug
- Do not ignore possible error states
  - Deal with exceptions appropriately