CS 153: Concepts of Compiler Design

November 30 Class Meeting

Department of Computer Science
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

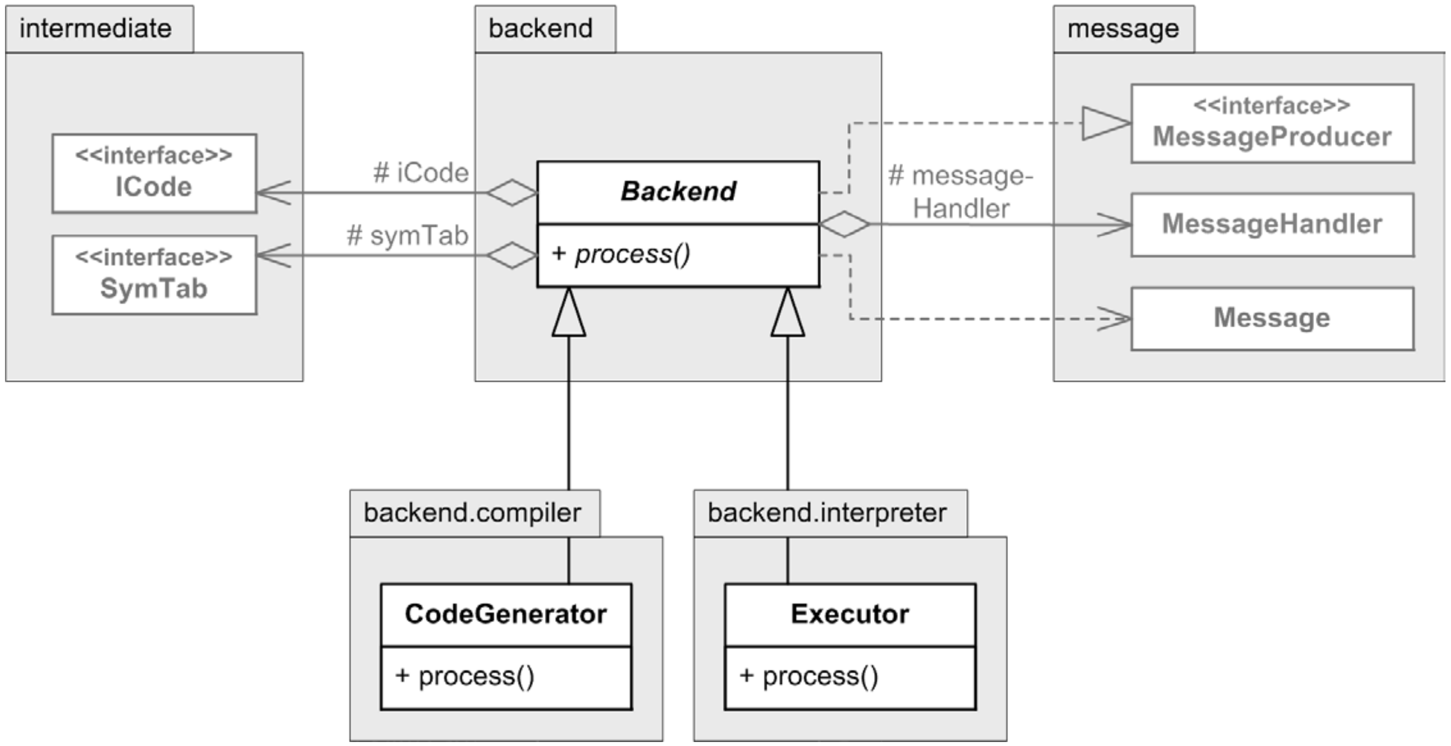
Extra-Credit Oral Presentations

- Let me know if your team would like to do an oral presentation for extra credit.
- Tell about your language.
  - Show some example programs.
- Describe its grammar
  - Show syntax diagrams.
- What Jasmin code do you generate?
  - Show some code diagrams.
- Demo: Compile, execute, and run some sample programs.

Extra-Credit Oral Presentations, *cont'd*

- Submit a note by Friday, Dec. 1 into Canvas: Assignments | Miscellaneous | Presentation if your team wants to present.
  - Choose either Dec. 5 or 7 to present.
- 15-20 minutes.
- Can add up to 50 points to each team member's total assignment score.

An Interactive Source-Level Debugger



- Control the interpreter at run time.
- Straightforward to implement within our framework.

Source-Level Debugger, *cont'd*

- When you're an interpreter, you're in <u>complete control</u> at run time of the source program's execution.
  - You can start and stop the execution.
  - You can examine and modify values of variables.
  - You have access to the entire runtime stack.

Machine-Level vs. Source-Level Debugging

- Machine level
  - Low level, close to the machine language.
  - Single stepping: Execute one machine (or assembly) instruction at a time.
  - Monitor and set the values of machine registers.

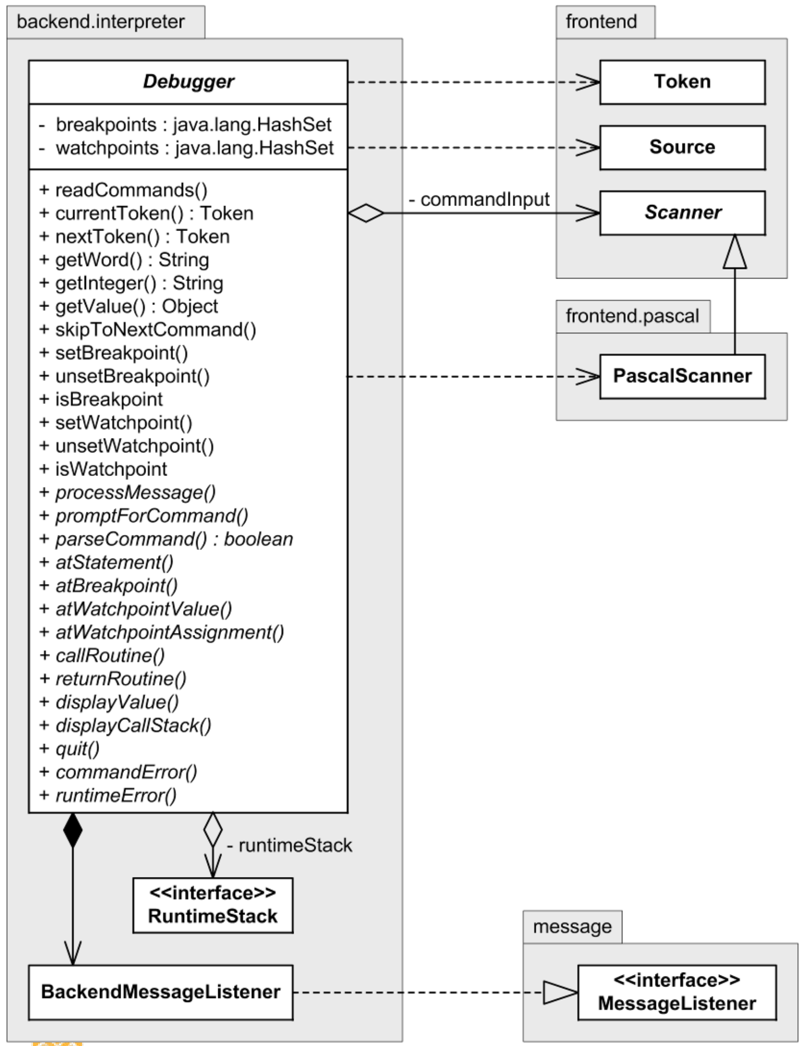
Machine-Level vs. Source-Level Debugging

- Source level
  - Debug at the high level of the source language.
  - Refer to variables by their names in the source program.
  - Refer to statements by their source line numbers.
  - Refer to procedures and functions by their names.
  - Access to the runtime stack.
  - AKA: symbolic debugger

Simple Debugger Command-Line Language

- Breakpoints: Pause program execution at certain statements.
- Watchpoints: Monitor the values of certain variables.
- Assignments: Change the values of variables.
- Single-step source program execution statement by statement.
- Display or set the value of a variable.
- Display the call stack with local values of each routine.

Debugger Architecture

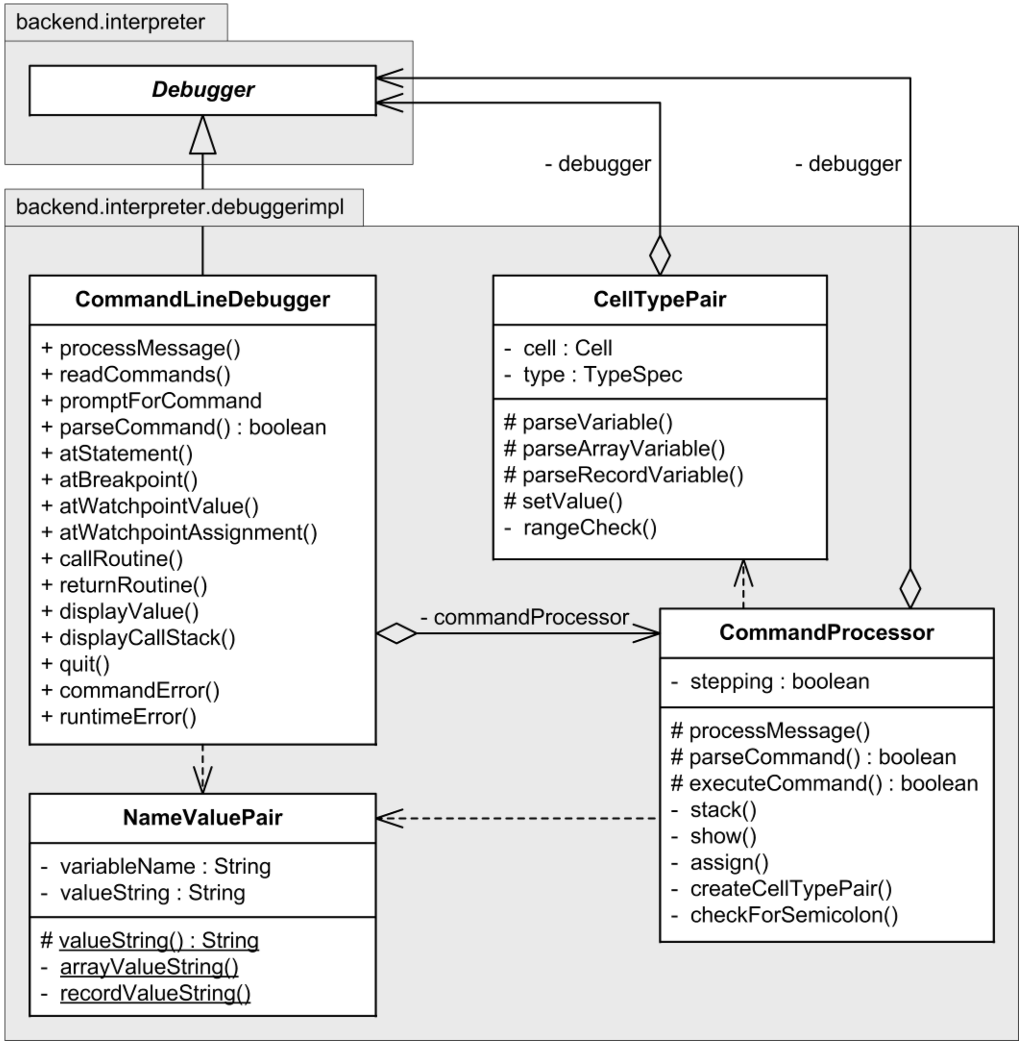


- **Debugger** is an abstract class with two subclasses:
  - **CommandLineDebugger**
  - **GUIDebugger**

- The **Debugger** class listens to messages from the back end.
  - Formerly, the main **Pascal** class listened to the back end messages.

- We need a parser for the command language.
  - Reuse the scanner and token classes from the front end!

Command Line Debugger Architecture



- **`CommandProcessor`**
  - Processes messages from the back end.
  - Parses debugger commands.
- **`CellTypePair`**
  - Keeps track of each memory cell and the data type of its value.
  - Parses variables in debugger commands.
- **`NameValuePair`**
  - Displays the current value of a variable given its name.

Back End Messages

- Messages sent by the interpreter during run time:

Message type	Sent by the back end …
`SOURCE_LINE`	Just before executing each statement.
`FETCH`	Whenever any variable's value is accessed.
`ASSIGN`	Whenever any variable's value is set.
`CALL`	Whenever a procedure or function is called.
`RETURN`	Upon returning from a procedure or function.
`RUNTIME_ERROR`	Whenever a runtime error occurs.

Method **StatementExecutor.execute()**

```
public Object execute(ICodeNode node)
{
    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();

    // Send a message about the current source line.
    sendSourceLineMessage(node);

    switch (nodeType) {

        case COMPOUND: {
            CompoundExecutor compoundExecutor = new CompoundExecutor(this);
            return compoundExecutor.execute(node);
        }
        ...
    }
}
```

- Send a **SOURCE_LINE** message before executing each statement.
  - The listener for this message is the debugger.

Method **CommandProcessor.processMessage()**

```
protected void processMessage(Message message)
{
    MessageType type = message.getType();

    switch (type) {

        case SOURCE_LINE: {
            int lineNumber = (Integer) message.getBody();

            if (stepping) {
                debugger.atStatement(lineNumber);
                debugger.readCommands();
            }
            else if (debugger.isBreakpoint(lineNumber)) {
                debugger.atBreakpoint(lineNumber);
                debugger.readCommands();
            }

            break;
        }
        ...
    }
}
```

Single-stepping.

Hit a breakpoint.

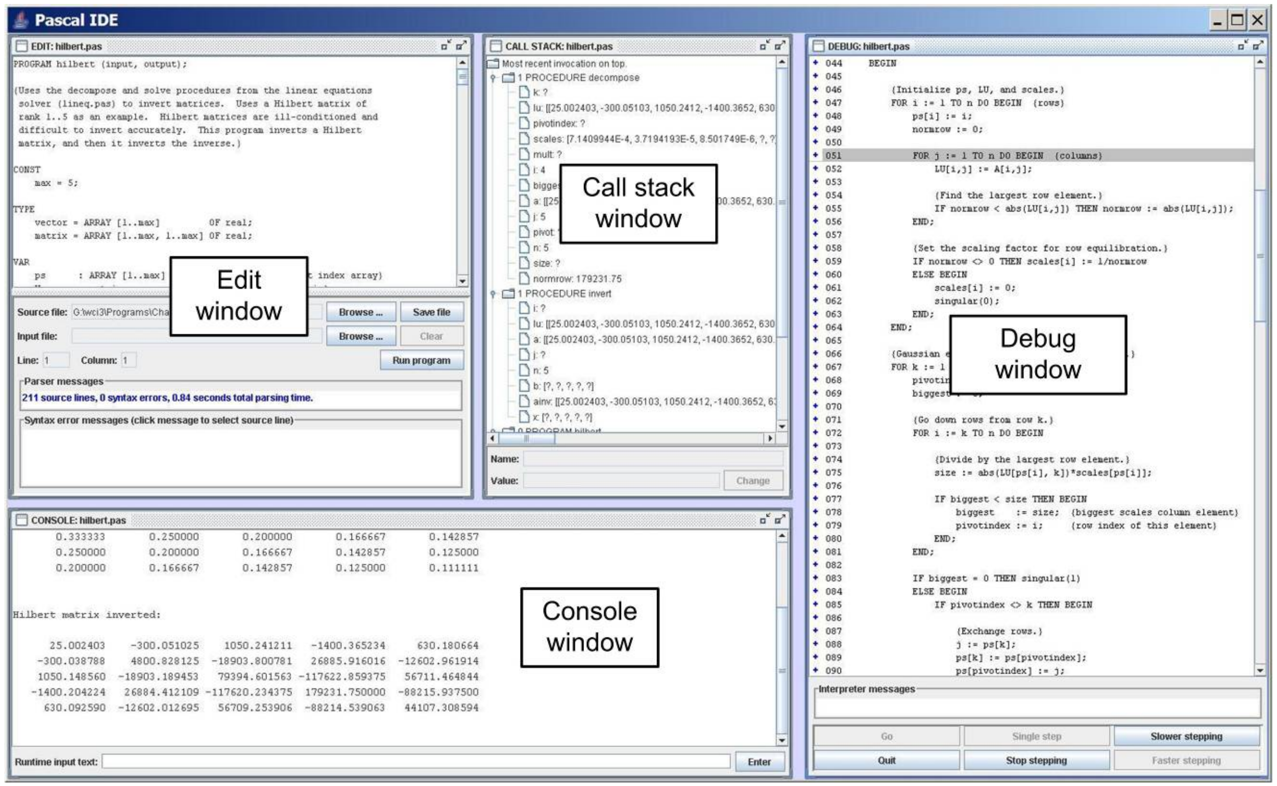
Demo

A Really Cool Debugger ...

- ... would have a graphical user interface (GUI).
- It would be part of a complete integrated development environment (IDE).
  - Different windows for editing, executing, monitoring, input and output, etc.
  - Buttons to invoke debugger operations.
  - Animate the execution of a program.
- Just like Eclipse!
  - Only perhaps not quite as good.

Integrated Development Environment (IDE)

- A graphical user interface (GUI) that integrates:
  - Edit window
  - Debug window
  - Call stack window
  - Console window
- Implemented with the Java Foundation Classes (Swing)
- Uses <u>multithreaded programming</u>.



```
PROGRAM hilbert (input, output);

{Uses the decompose and solve procedures from the linear equations
 solver (lineq.pas) to invert matrices.  Uses a Hilbert matrix of
 rank 1..5 as an example.  Hilbert matrices are ill-conditioned and
 difficult to invert accurately.  This program inverts a Hilbert
 matrix, and then it inverts the inverse.}

CONST
   max = 5;

TYPE
   vector = ARRAY [1..max]         OF real;
   matrix = ARRAY [1..max, 1..max] OF real;

VAR
   ps     : ARRAY [1..max]                 index array}
```

211 source lines, 0 syntax errors, 0.84 seconds total parsing time.

```
Most recent invocation on top.
1 PROCEDURE decompose
    k: ?
    lu: [[25.002403, -300.05103, 1050.2412, -1400.3652, 630
    pivotindex: ?
    scales: [7.1409944E-4, 3.7194193E-5, 8.501749E-6, ?, ?]
    mult: ?
    i: 4
    bigges
    a: [[25                                          00.3652, 630.
    j: 5
    pivot:
    n: 5
    size: ?
    normrow: 179231.75
1 PROCEDURE invert
    i: ?
    lu: [[25.002403, -300.05103, 1050.2412, -1400.3652, 630
    a: [[25.002403, -300.05103, 1050.2412, -1400.3652, 630.
    j: ?
    n: 5
    b: [?, ?, ?, ?, ?]
    ainv: [[25.002403, -300.05103, 1050.2412, -1400.3652, 63
    x: [?, ?, ?, ?, ?]
0 PROGRAM hilbert
```

```
    BEGIN

        {Initialize ps, LU, and scales.}
        FOR i := 1 TO n DO BEGIN  {rows}
            ps[i] := i;
            normrow := 0;

            FOR j := 1 TO n DO BEGIN  {columns}
                LU[i,j] := A[i,j];

                {Find the largest row element.}
                IF normrow < abs(LU[i,j]) THEN normrow := abs(LU[i,j]);
            END;

            {Set the scaling factor for row equilibration.}
            IF normrow <> 0 THEN scales[i] := 1/normrow
            ELSE BEGIN
                scales[i] := 0;
                singular(0);
            END;
        END;

        {Gaussian e                                    }
        FOR k := 1
            pivotin
            biggest

            {Go down rows from row k.}
            FOR i := k TO n DO BEGIN

                {Divide by the largest row element.}
                size := abs(LU[ps[i], k])*scales[ps[i]];

                IF biggest < size THEN BEGIN
                    biggest    := size;  {biggest scales column element}
                    pivotindex := i;     {row index of this element}
                END;
            END;

            IF biggest = 0 THEN singular(1)
            ELSE BEGIN
                IF pivotindex <> k THEN BEGIN

                    {Exchange rows.}
                    j := ps[k];
                    ps[k] := ps[pivotindex];
                    ps[pivotindex] := j;
```

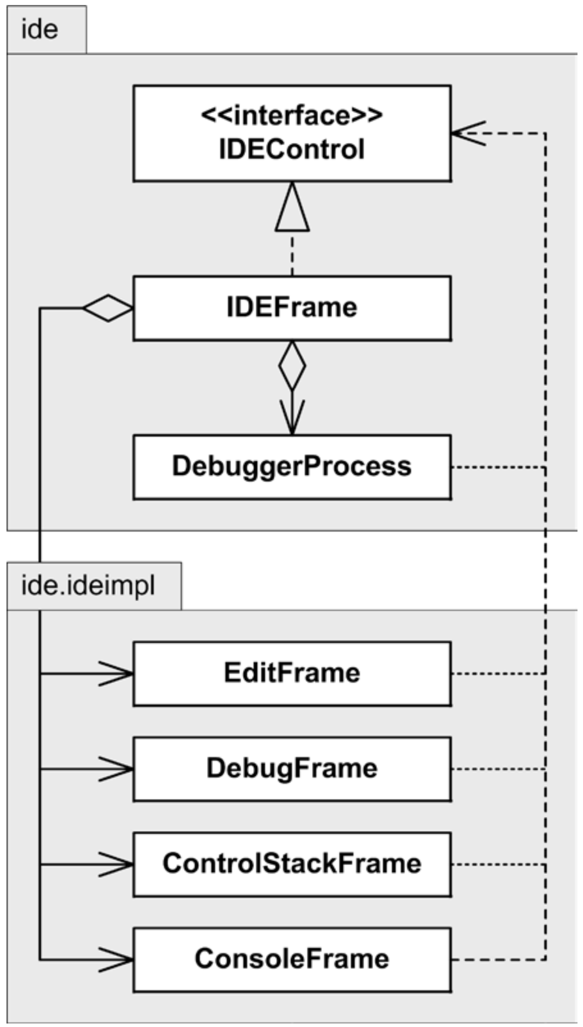
```
      0.333333      0.250000      0.200000      0.166667      0.142857
      0.250000      0.200000      0.166667      0.142857      0.125000
      0.200000      0.166667      0.142857      0.125000      0.111111


Hilbert matrix inverted:

     25.002403   -300.051025   1050.241211  -1400.365234    630.180664
   -300.038788   4800.828125 -18903.800781  26885.916016 -12602.961914
   1050.148560 -18903.189453  79394.601563-117622.859375  56711.464844
  -1400.204224  26884.412109-117620.234375 179231.750000 -88215.937500
    630.092590 -12602.012695  56709.253906 -88214.539063  44107.308594
```

"Wrap a GUI" around the command-line debugger.

The Basic Idea Behind the IDE

- Run the Pascal command-line debugger in one process.
- Run the IDE GUI code in another process.
- The user manually performs an action on the IDE GUI (e.g., click the Step button)
  - The IDE sends the appropriate command-line command to the debugger.
  - The debugger reads the command from its standard input just as if the user had typed it on the command line.

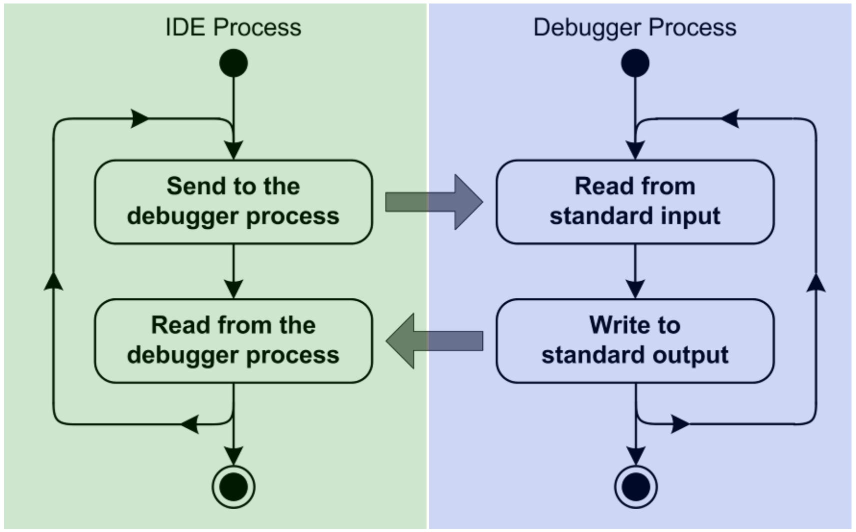
The IDE Framework



- Multithreaded programming:
  - The IDE GUI code runs in the IDE process.
  - The debugger code runs in the debugger process.

Interprocess Communication

- The IDE process sends debugger commands to the debugger process.
  - The IDE writes the commands to its standard output.
  - The debugger reads the commands via its standard input.
- The debugger process sends status information or program output to the IDE process.
  - The debugger writes to its standard output.
  - The IDE reads the debugger’s output via its standard input.



- The debugger running in the debugger process believes that it’s reading debugger commands typed on the command line and that it’s writing to the console.

Recall: Class **`CommandLineDebugger`**

```
public class CommandLineDebugger extends Debugger
{
    ...

    public void atStatement(Integer lineNumber)
    {
        System.out.println("\n>>> At line " + lineNumber);
    }

    public void atBreakpoint(Integer lineNumber)
    {
        System.out.println("\n>>> Breakpoint at line " + lineNumber);
    }

    ...
}
```

- The command-line debugger writes messages to the console (via its standard output) for the user.

Compare to: Class **`GUIDebugger`**

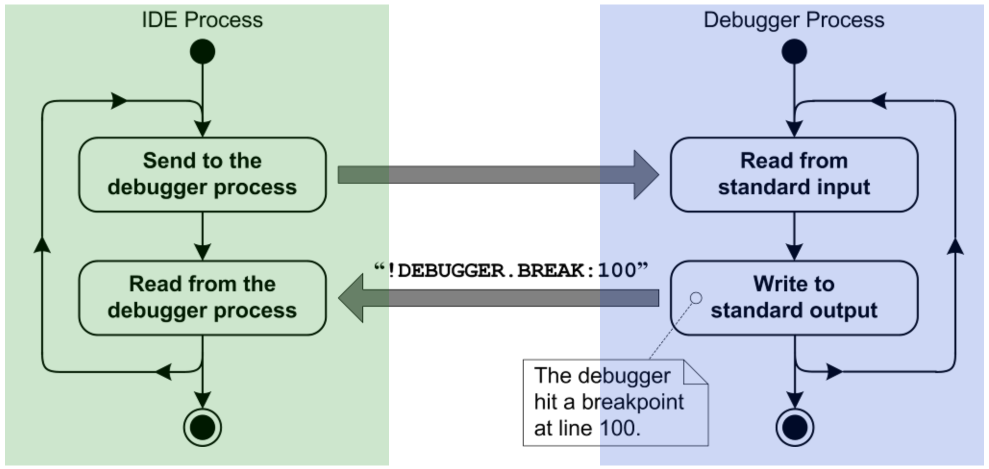
```
public class GUIDebugger extends Debugger
{
    ...

    public void atStatement(Integer lineNumber)
    {
        System.out.println(DEBUGGER_AT_TAG + lineNumber);
    }

    public void atBreakpoint(Integer lineNumber)
    {
        System.out.println(DEBUGGER_BREAK_TAG + lineNumber);
    }

    ...
}
```

- The GUI debugger writes tagged messages to the "console" (via its standard output) for the GUI process.

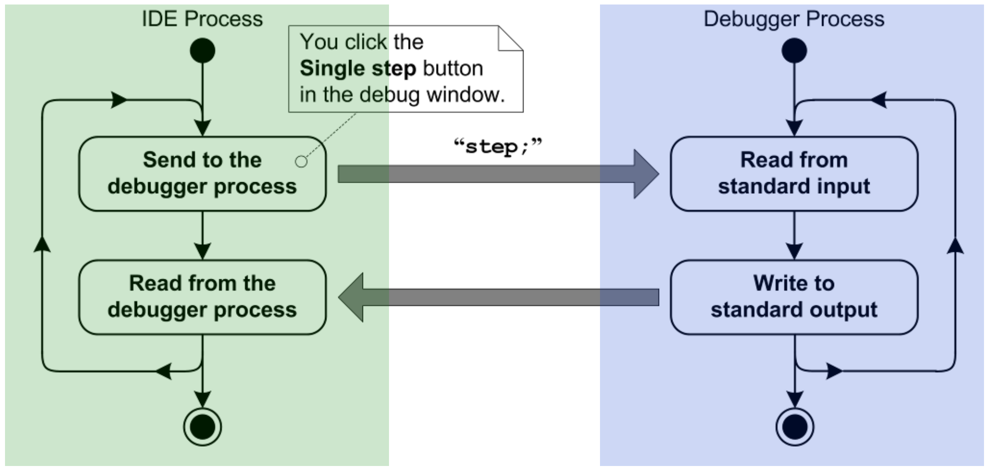
Tagged Messages for the GUI Process

```
package wci.ide;

public interface IDEControl
{
    // Debugger output line tags.
    public static final String LISTING_TAG = "!LISTING:";
    public static final String PARSER_TAG = "!PARSER:";
    public static final String SYNTAX_TAG = "!SYNTAX:";
    public static final String INTERPRETER_TAG = "!INTERPRETER:";

    public static final String DEBUGGER_AT_TAG = "!DEBUGGER.AT:";
    public static final String DEBUGGER_BREAK_TAG = "!DEBUGGER.BREAK:";
    public static final String DEBUGGER_ROUTINE_TAG = "!DEBUGGER.ROUTINE:";
    public static final String DEBUGGER_VARIABLE_TAG = "!DEBUGGER.VARIABLE:";

    ...
}
```

Interprocess Communication, *cont'd*

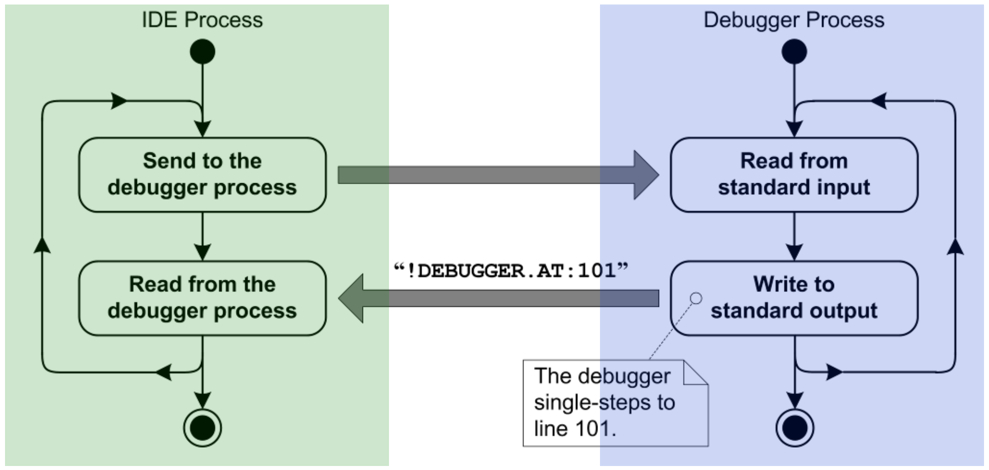

IDE Process
Send to the debugger process
Read from the debugger process
Debugger Process
Read from standard input
Write to standard output
"!DEBUGGER.BREAK:100"
The debugger hit a breakpoint at line 100.

Interprocess Communication, *cont'd*



IDE Process
You click the **Single step** button in the debug window.
Send to the debugger process
Read from the debugger process
"step;"
Debugger Process
Read from standard input
Write to standard output

Interprocess Communication, *cont'd*


IDE Process
Send to the debugger process
Read from the debugger process
Debugger Process
Read from standard input
Write to standard output
"!DEBUGGER.AT:101"
The debugger single-steps to line 101.