Yan Peng
921759056
Dec 5th, 2021

Assignment 5 Documentation

GitHub Repository

Link for assignment 5

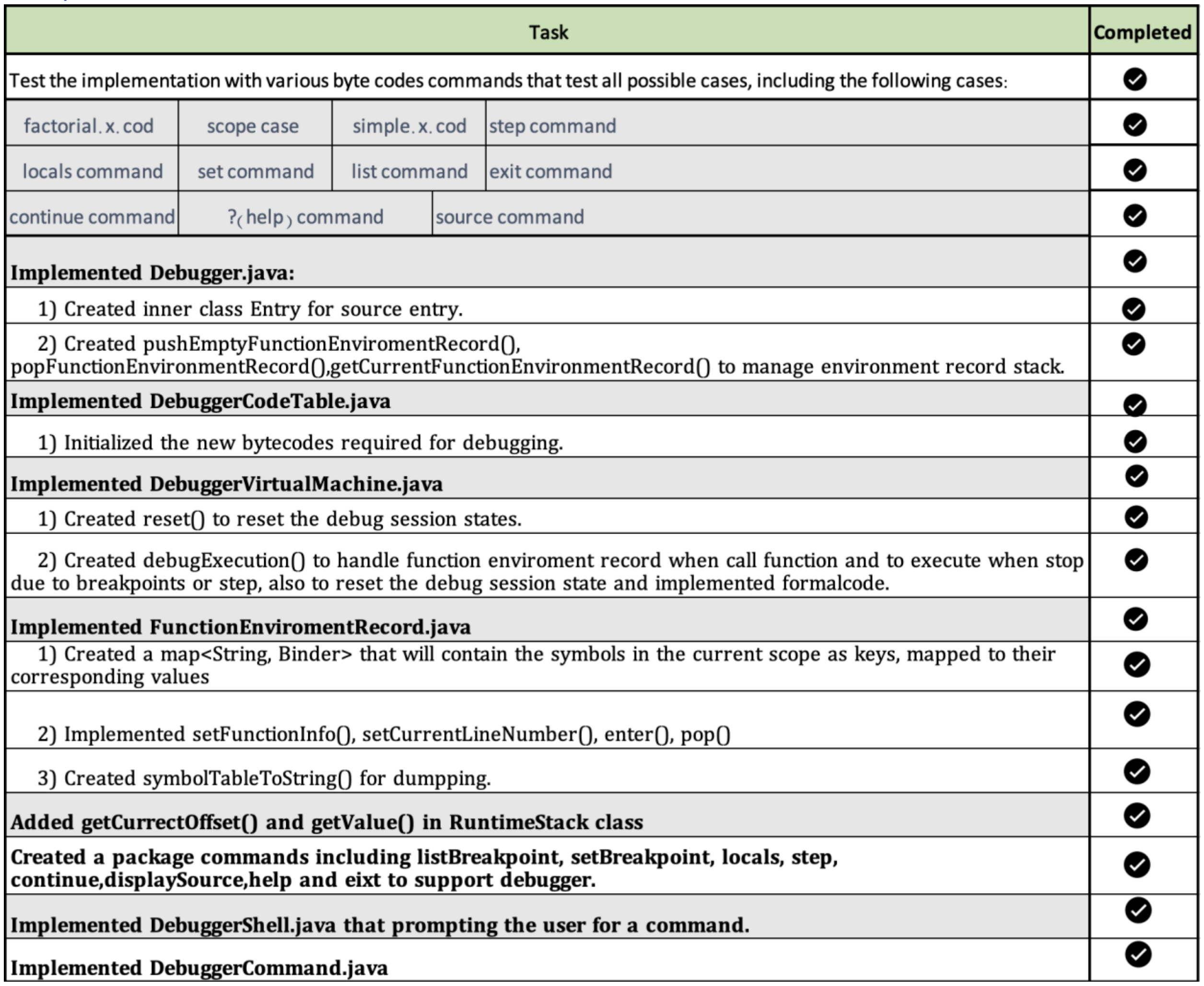
https://github.com/sfsu-csc-413-fall-2021/assignment-5---debugger-yuqiao1205

Project Introduction and Overview

The purpose of this project is to implement a debugger by continuing the work done on the large compiler code base.

Summary of technical work: In order to implement the debugger. I created three new byte codes classes as subclasses of ByteCode to support debugging. I implemented the FunctionEnviromentRecord class to hold all the information we need such as function name, start and end lines of the function in source code, and a way to track the current function’s state in while debugging. I modified the DebuggerCodeTable to accommodate the three new byte codes. I also realized the Debugger that extends the Interpreter. I implemented the suite of commands for the debugger including set, locals, source and so on.

Scope of Work



Task				Completed
Test the implementation with various byte codes commands that test all possible cases, including the following cases:				✓
factorial.x.cod	scope case	simple.x.cod	step command	✓
locals command	set command	list command	exit command	✓
continue command	?(help) command	source command		✓
Implemented Debugger.java:				✓
1) Created inner class Entry for source entry.				✓
2) Created pushEmptyFunctionEnviromentRecord(), popFunctionEnvironmentRecord(),getCurrentFunctionEnvironmentRecord() to manage environment record stack.				✓
Implemented DebuggerCodeTable.java				✓
1) Initialized the new bytecodes required for debugging.				✓
Implemented DebuggerVirtualMachine.java				✓
1) Created reset() to reset the debug session states.				✓
2) Created debugExecution() to handle function enviroment record when call function and to execute when stop due to breakpoints or step, also to reset the debug session state and implemented formalcode.				✓
Implemented FunctionEnviromentRecord.java				✓
1) Created a map<String, Binder> that will contain the symbols in the current scope as keys, mapped to their corresponding values				✓
2) Implemented setFunctionInfo(), setCurrentLineNumber(), enter(), pop()				✓
3) Created symbolTableToString() for dumpping.				✓
Added getCurrectOffset() and getValue() in RuntimeStack class				✓
Created a package commands including listBreakpoint, setBreakpoint, locals, step, continue,displaySource,help and eixt to support debugger.				✓
Implemented DebuggerShell.java that prompting the user for a command.				✓
Implemented DebuggerCommand.java				✓

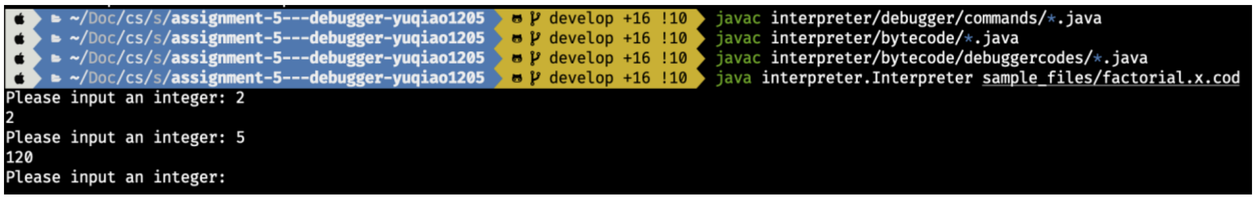
Execution and Development Environment

I developed from this code base using IntelliJ IDEA Community v. 2021.2.3. This Java application was compiled using Java JDK version 11.0.12 which is the most up to date version of the JDK.

Compilation Result

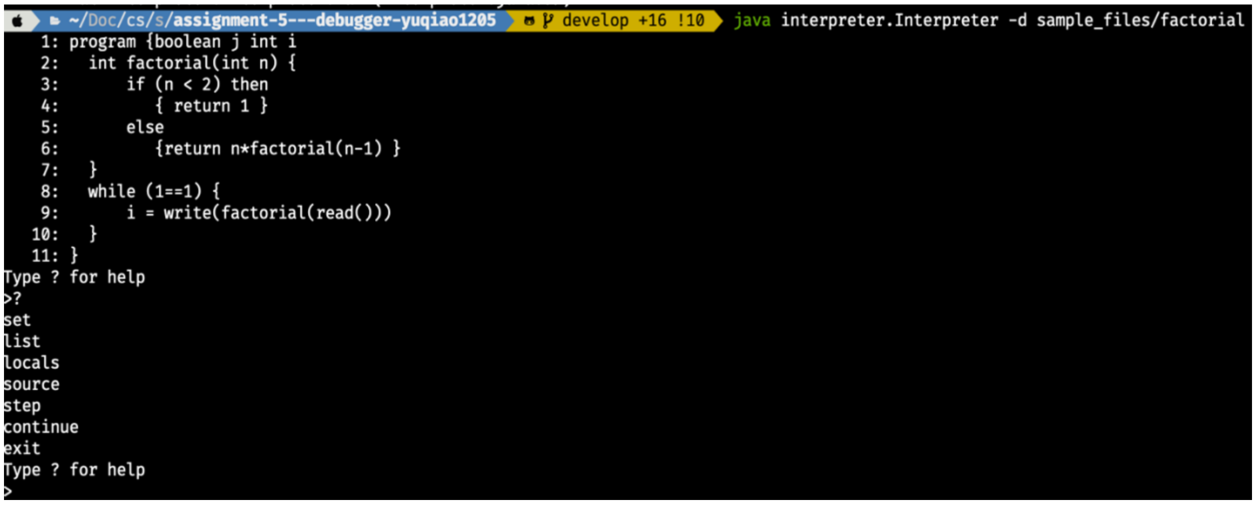
Using the instructions provided in the assignment one specification:

```
>  javac interpreter/debugger/commands/*.java
>  javac interpreter/bytecode/*.java
>  javac interpreter/bytecode/debuggercodes/*.java
>  javac interpreter/Interpreter.java
>  java interpreter.Interpreter sample_files/factorial.x.cod
```



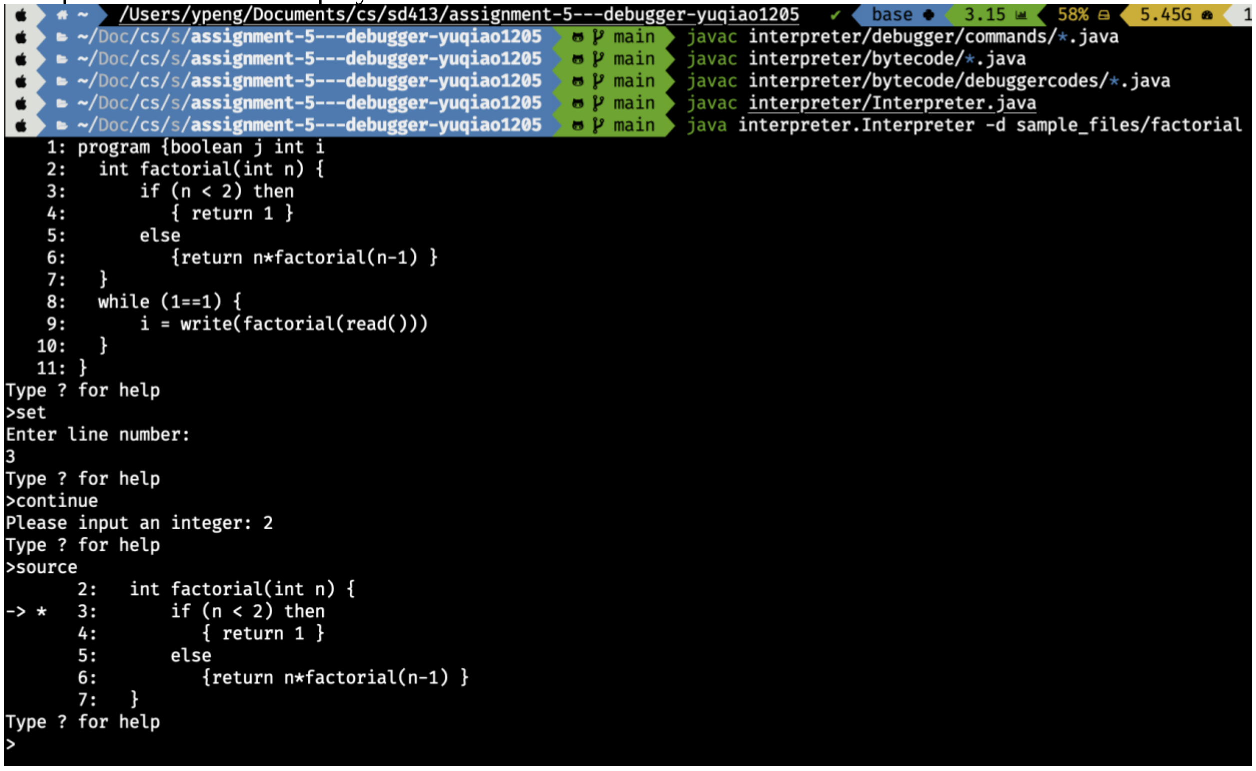
```
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  develop +16 !10  javac interpreter/debugger/commands/*.java
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  develop +16 !10  javac interpreter/bytecode/*.java
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  develop +16 !10  javac interpreter/bytecode/debuggercodes/*.java
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  develop +16 !10  java interpreter.Interpreter sample_files/factorial.x.cod
Please input an integer: 2
2
Please input an integer: 5
120
Please input an integer:
```

```
> java interpreter.Interpreter -d sample_files/factorial
```



```
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  develop +16 !10  java interpreter.Interpreter -d sample_files/factorial
    1: program {boolean j int i
    2:   int factorial(int n) {
    3:       if (n < 2) then
    4:          { return 1 }
    5:       else
    6:          {return n*factorial(n-1) }
    7:   }
    8:   while (1==1) {
    9:       i = write(factorial(read()))
   10:   }
   11: }
Type ? for help
>?
set
list
locals
source
step
continue
exit
Type ? for help
>
```

The picture shown is displayed the source code of the current function.



```
  ~  /Users/ypeng/Documents/cs/sd413/assignment-5---debugger-yuqiao1205  ✔  base  3.15  58%  5.45G  1
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  main  javac interpreter/debugger/commands/*.java
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  main  javac interpreter/bytecode/*.java
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  main  javac interpreter/bytecode/debuggercodes/*.java
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  main  javac interpreter/Interpreter.java
 ~/Doc/cs/s/assignment-5---debugger-yuqiao1205  main  java interpreter.Interpreter -d sample_files/factorial
    1: program {boolean j int i
    2:   int factorial(int n) {
    3:       if (n < 2) then
    4:          { return 1 }
    5:       else
    6:          {return n*factorial(n-1) }
    7:   }
    8:   while (1==1) {
    9:       i = write(factorial(read()))
   10:   }
   11: }
Type ? for help
>set
Enter line number:
3
Type ? for help
>continue
Please input an integer: 2
Type ? for help
>source
       2:   int factorial(int n) {
-> *   3:       if (n < 2) then
       4:          { return 1 }
       5:       else
       6:          {return n*factorial(n-1) }
       7:   }
Type ? for help
>
```

No error messages or warnings were displayed, and the application ran as expected

Assumptions

I implemented some error handling for some byte codes case.

Implementation

DebuggerShell

This class implemented to encapsulate prompting the user for a command.

Debugger

This class is the heart of system, loads the byte code file, as well as the source file. Also, it initializes, and creates main environment record stack, initializes the debug shell. In the Debugger class I encapsulated some functionality relating to the FunctionEnvironmentRecord and exported this behavior to be used by the collaborating class DebuggerVirtualMachine.

FunctionEnvironmentRecord

The major role of this record is to keep a history of the values of variables forming the function's environment. It also collects together information relating to the source, e.g. function name, start and end line numbers.

DebuggerVirtualMachine

DebuggerVirtualMachine collaborate with debugger to execute a program with debugger bytecode.

commands package

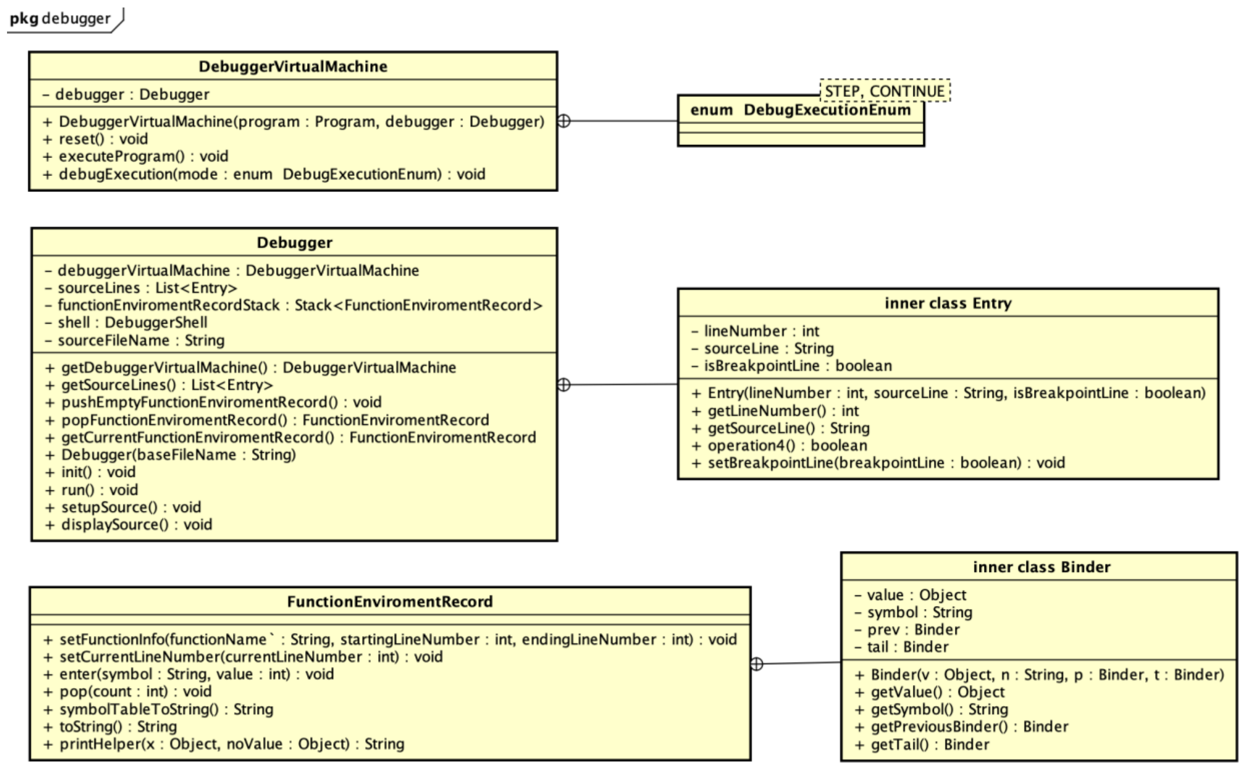
This package contains all the debugger commands we need to cause debugger execution such as set, step, continue, locals, source, and exit.

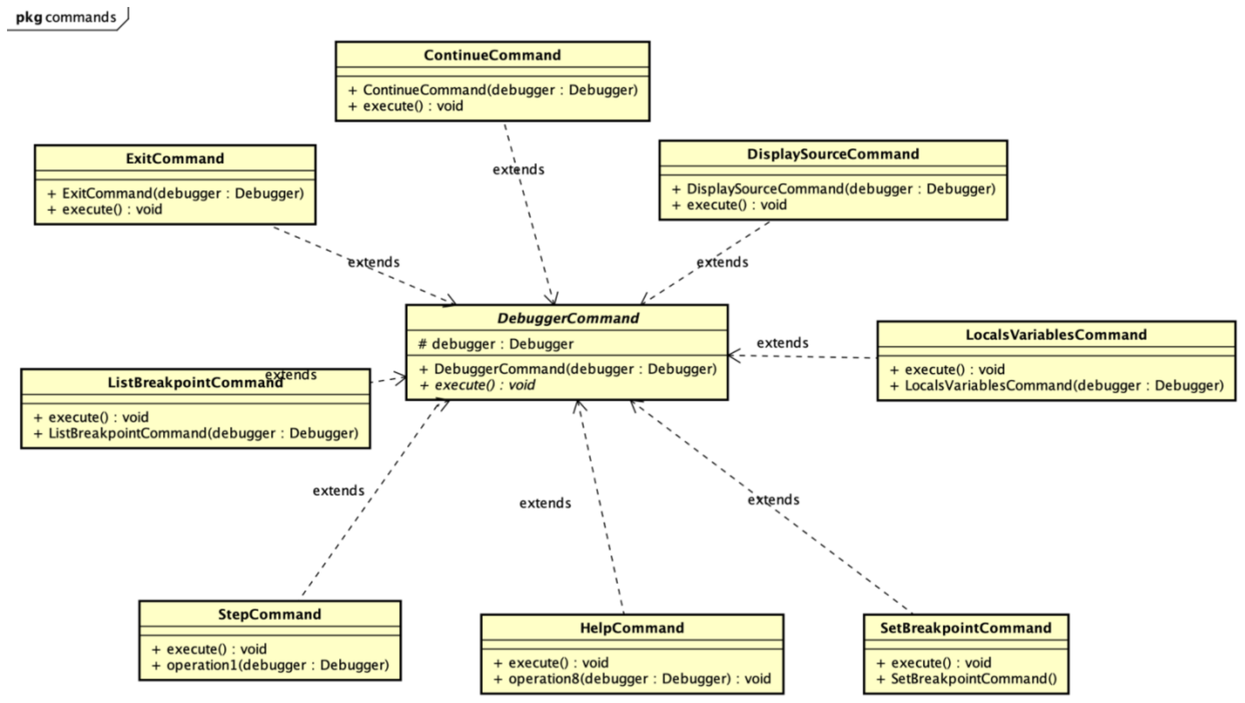
Code Organization

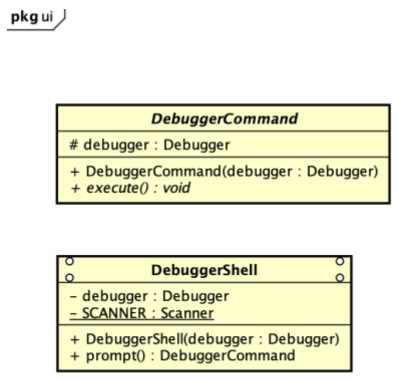
To organize the code as much as possible, I put three new byte codes in debuggercodes package. Each bytecode subclass will have its own file and behavior to support debugging.

Class Diagram

The following class diagrams show the details of all the classes in this project, including the inheritance hierarchy


pkg debugger
DebuggerVirtualMachine
– debugger : Debugger
+ DebuggerVirtualMachine(program : Program, debugger : Debugger)
+ reset() : void
+ executeProgram() : void
+ debugExecution(mode : enum DebugExecutionEnum) : void
STEP, CONTINUE
enum DebugExecutionEnum
Debugger
– debuggerVirtualMachine : DebuggerVirtualMachine
– sourceLines : List<Entry>
– functionEnviromentRecordStack : Stack<FunctionEnviromentRecord>
– shell : DebuggerShell
– sourceFileName : String
+ getDebuggerVirtualMachine() : DebuggerVirtualMachine
+ getSourceLines() : List<Entry>
+ pushEmptyFunctionEnviromentRecord() : void
+ popFunctionEnviromentRecord() : FunctionEnviromentRecord
+ getCurrentFunctionEnviromentRecord() : FunctionEnviromentRecord
+ Debugger(baseFileName : String)
+ init() : void
+ run() : void
+ setupSource() : void
+ displaySource() : void
inner class Entry
– lineNumber : int
– sourceLine : String
– isBreakpointLine : boolean
+ Entry(lineNumber : int, sourceLine : String, isBreakpointLine : boolean)
+ getLineNumber() : int
+ getSourceLine() : String
+ operation4() : boolean
+ setBreakpointLine(breakpointLine : boolean) : void
FunctionEnviromentRecord
+ setFunctionInfo(functionName` : String, startingLineNumber : int, endingLineNumber : int) : void
+ setCurrentLineNumber(currentLineNumber : int) : void
+ enter(symbol : String, value : int) : void
+ pop(count : int) : void
+ symbolTableToString() : String
+ toString() : String
+ printHelper(x : Object, noValue : Object) : String
inner class Binder
– value : Object
– symbol : String
– prev : Binder
– tail : Binder
+ Binder(v : Object, n : String, p : Binder, t : Binder)
+ getValue() : Object
+ getSymbol() : String
+ getPreviousBinder() : Binder
+ getTail() : Binder


pkg commands
ContinueCommand
+ ContinueCommand(debugger : Debugger)
+ execute() : void
ExitCommand
+ ExitCommand(debugger : Debugger)
+ execute() : void
DisplaySourceCommand
+ DisplaySourceCommand(debugger : Debugger)
+ execute() : void
extends
extends
extends
DebuggerCommand
# debugger : Debugger
+ DebuggerCommand(debugger : Debugger)
+ execute() : void
LocalsVariablesCommand
+ execute() : void
+ LocalsVariablesCommand(debugger : Debugger)
extends
ListBreakpointCommand
extends
+ execute() : void
+ ListBreakpointCommand(debugger : Debugger)
extends
extends
extends
StepCommand
+ execute() : void
+ operation1(debugger : Debugger)
HelpCommand
+ execute() : void
+ operation8(debugger : Debugger) : void
SetBreakpointCommand
+ execute() : void
+ SetBreakpointCommand()



Results and Conclusion

The Debugger works as described and has been tested by dumping the flow of execution. I tested many cases. For example, tested debugger factorial.x and show the correct results when I run them. Also displayed the source code of the current function, with an indication of where the execution has stopped (if executing), and any breakpoints that have been set.