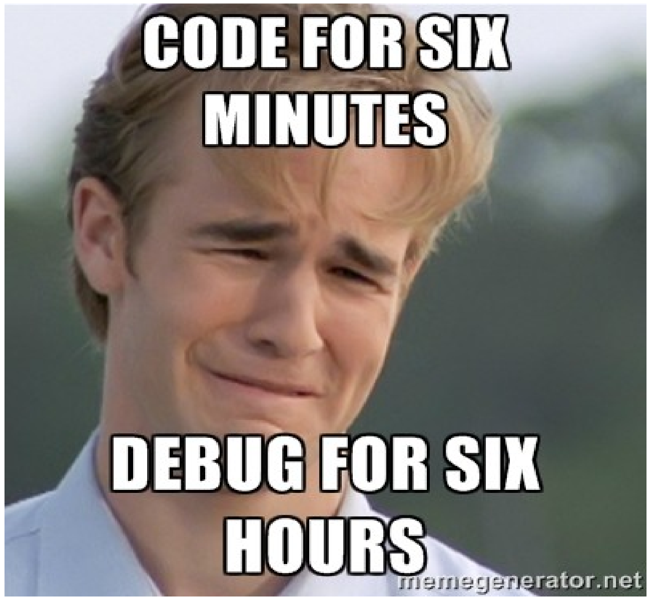
Debugging - Hints and techniques



Rodrigo Alvares de Souza
IAG-USP

What is debugging?

- According to Wikipedia, debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program.
- However, most people involved in spotting and removing those defects would define it as an art rather then a method.

What is debugging? (cont.)

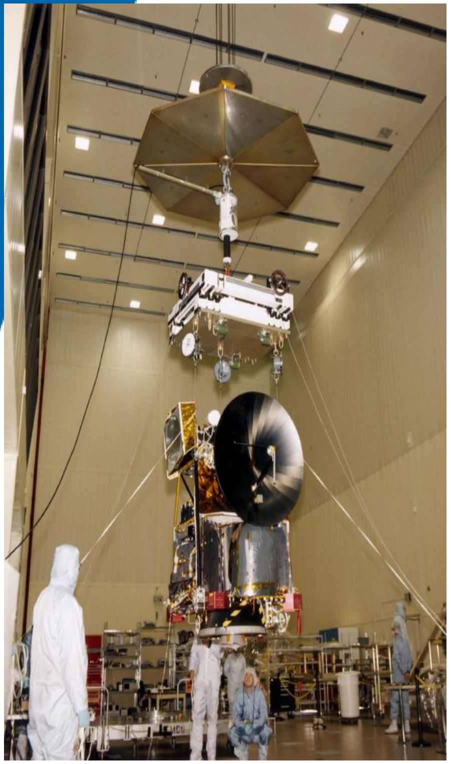
- The importance of a method of finding errors and fixing them during the life-cycle of a software product cannot be stressed enough.
- Testing and debugging are fundamental parts of programmer's everyday activity but some people still consider it an annoying option.
- Finding a bug is a process of confirming what is working until something wrong is found.

Some facts

- In 1998, a crew member of the guided-missile cruiser USS Yorktown mistakenly entered a zero as data value, which resulted in a division by zero.
- The error cascaded and eventually shut down the ship's propulsion system. The ship was dead in water for several hours because a program didn't check for valid input.

Some facts (cont.)

- In 1999, the 125 million dollars Mars Climate Orbiter was assumed lost by officials at NASA. The failure responsible for loss of the orbiter was attributed to a failure of NASA's system engineering process.
- The process did not specify the system of measurement to be used on the project. As a result, one of the development teams used Imperial measurement while the other used the metric system.
- When parameters from one module were passed to another, during orbit navigation correction, no conversion was performed, resulting in the loss of the craft.



Why does software have bugs?

- ✓ Human factor
- ✓ Communication Failure
- ✓ Unrealistic development timeframe
- ✓ Poor design logic
- ✓ Poor coding practices
- ✓ Lack of version control
- ✓ Buggy third-party tools
- ✓ Lack of skilled testing
- ✓ Last minute changes
- ✓ Advisor, of course...

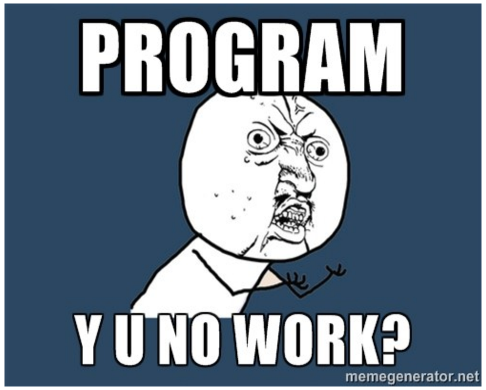
General concepts about debugging

- After many days of brainstorming, designing and coding, the programmer finally have a wonderful piece of code. Everything seems pretty straightforward but unfortunately it doesn't work! And now? Now the great fun starts! Time to dig into the wonderful world of debugging.

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTER-NOON!
S. Adams

6 Stages of Debugging Grief

- ✓ DENIAL: That can't happen to my code.
- ✓ DISBELIEF: Works fine on my machine!
- ✓ ANGER: That shouldn't happen. WTF!!
- ✓ DEPRESSION: Why does that happen?
- ✓ ANGER (Again): Oh I see...
- ✓ ACCEPTANCE: How did that ever work?



Main steps

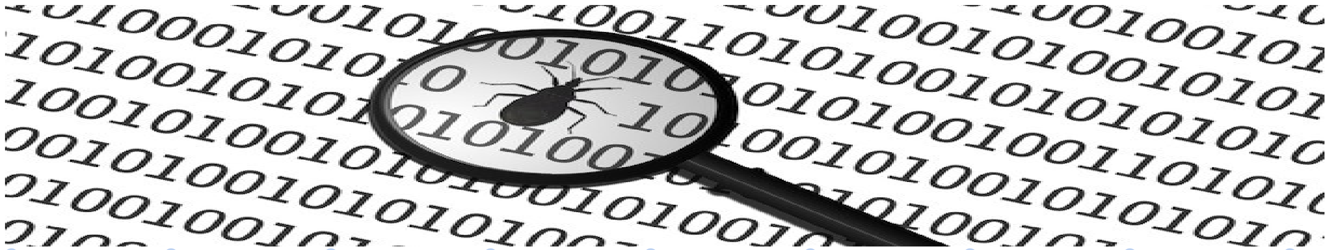
The debugging process can be divided into four main steps:

- ✓ Localizing a bug
- ✓ Classifying a bug
- ✓ Understanding a bug
- ✓ Repairing a bug

PUT MORE BUGS IN THE SOFTWARE! I'M MAKING A FORTUNE OUT HERE!

1 – Localizing a Bug

A typical attitude of inexperienced programmers towards bugs is to consider their localization an easy task: they notice their code does not do what they expected, and they are led astray by their confidence in knowing what their code should do. This confidence is completely deceptive because spotting a bug can be very difficult.



2 – Classifying a bug

Despite the appearance, bugs have often a common background. This allows to attempt a quite coarse, but sometimes useful, classification:

- ✓ Syntactical Errors: should be easily caught by your compiler. I say "should" because compilers, beside being very complicated, can be buggy themselves. In any case, it is vital to remember that quite often the problem might not be at the exact position indicated by the compiler error message.

2 – Classifying a bug (cont.)

- Build Errors: derive from linking object files which were not rebuilt after a change in some source files. These problems can easily be avoided by using tools to drive software building, like GNU Make.
- Basic Semantic Errors: comprise using uninitialized variables, dead code and problems with variable types. A compiler can highlight them to your attention, although it usually has to be explicitly asked through flags.
- Semantic Errors: include using wrong variables or operators (e.g., & instead of && in C++). No tool can catch these problems, because they are syntactically correct statements, although logically wrong.

Physicists' way to classify a bug

A funny physical classification distinguishes between Bohrbugs and Heisenbugs.

- Bohrbugs are deterministic: a particular input will always manifest them with the same result.
- Heisenbugs are random: difficult to reproduce reliably, since they seem to depend on environmental factors (e.g. a particular memory allocation, the way the operating system schedules processes, the phase of the moon and so on). In C++ a Heisenbug is very often the result of an error with pointers.

3 – Understanding a bug

A bug should be fully understood before attempting to fix it. Trying to fix a bug before understanding it completely could end in provoking even more damage to the code, since the problem could change form and manifest itself somewhere else, maybe randomly. The following check-list is useful to assure a correct approach to the investigation:

- ✓ Do not confuse observing symptoms with finding the real source of the problem;
- ✓ Check if similar mistakes (especially wrong assumptions) were made elsewhere in the code;
- ✓ Verify that just a programming error, and not a more fundamental problem (e.g. an incorrect algorithm), was found.

4 – Repairing a bug

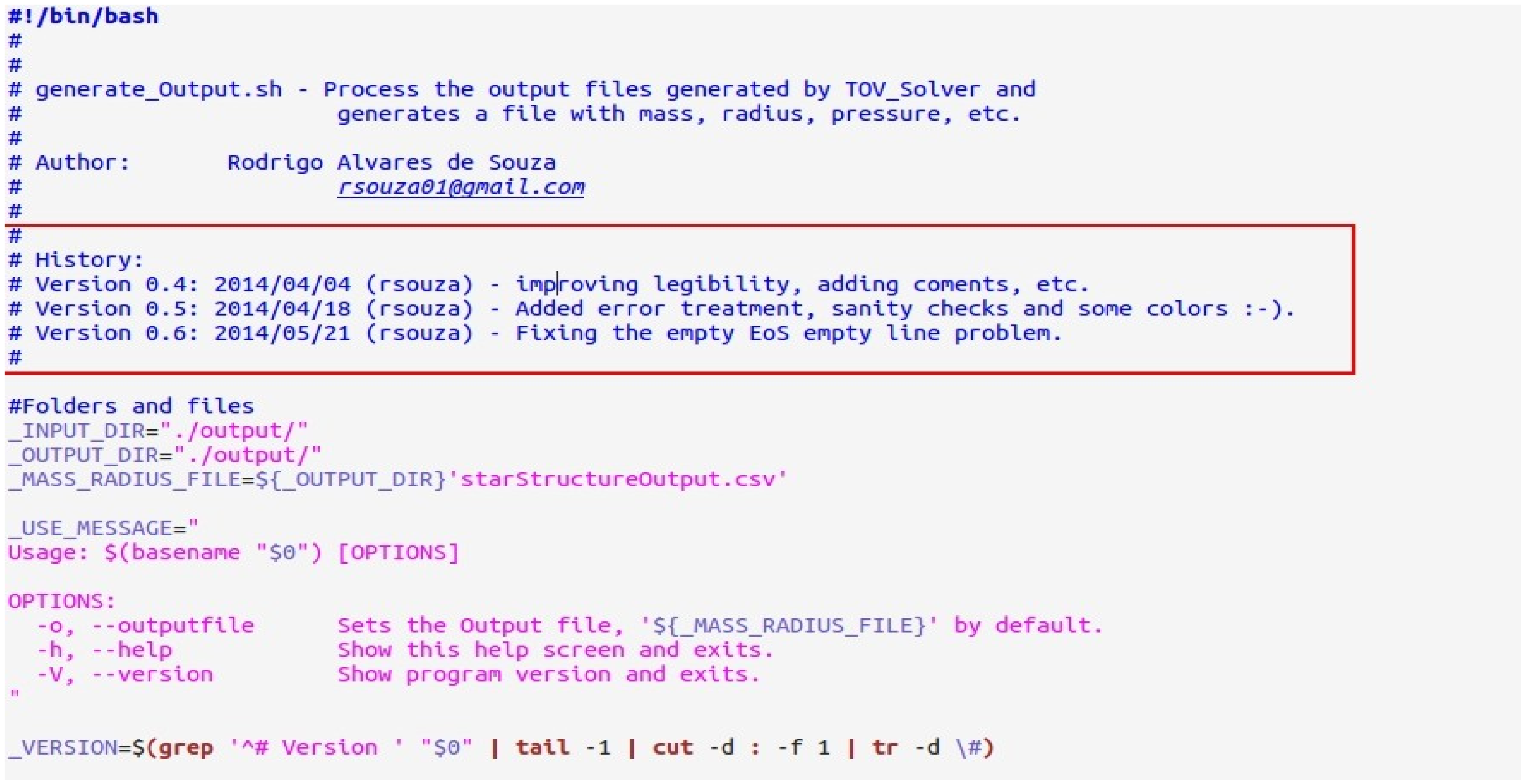
The final step in the debugging process is bug fixing. Repairing a bug is more than modifying code. Any fixes must be documented in the code and tested properly. More important, learning from mistakes is an effective attitude: it is good practice filling a small file with detailed explanations about the way the bug was discovered and corrected.

4 – Repairing a bug (Keeping track)

Several points are worth recording:

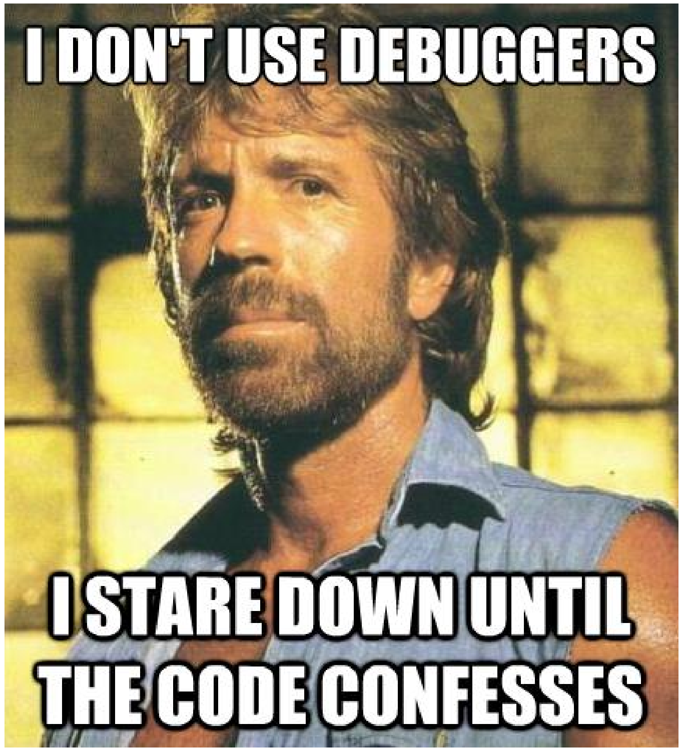
- ✓ how the bug was noticed, to help in writing a test case;
- ✓ how it was tracked down, to give you a better insight on the approach to choose in similar circumstances;
- ✓ what type of bug was encountered;
- ✓ if this bug was encountered often, in order to set up a strategy to prevent it from recurring;

4 – Repairing a bug (Keeping track)



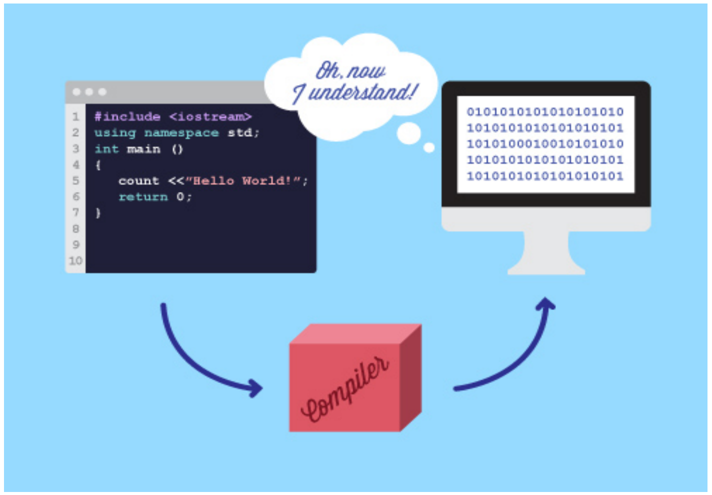
```
#!/bin/bash
#
#
# generate_Output.sh - Process the output files generated by TOV_Solver and
#                        generates a file with mass, radius, pressure, etc.
#
# Author:        Rodrigo Alvares de Souza
#                        rsouza01@gmail.com
#
#
# History:
# Version 0.4: 2014/04/04 (rsouza) - improving legibility, adding coments, etc.
# Version 0.5: 2014/04/18 (rsouza) - Added error treatment, sanity checks and some colors :-).
# Version 0.6: 2014/05/21 (rsouza) - Fixing the empty EoS empty line problem.
#

#Folders and files
_INPUT_DIR="./output/"
_OUTPUT_DIR="./output/"
_MASS_RADIUS_FILE=${_OUTPUT_DIR}'starStructureOutput.csv'

_USE_MESSAGE="
Usage: $(basename "$0") [OPTIONS]

OPTIONS:
  -o, --outputfile       Sets the Output file, '${_MASS_RADIUS_FILE}' by default.
  -h, --help             Show this help screen and exits.
  -V, --version          Show program version and exits.
"

_VERSION=$(grep '^# Version ' "$0" | tail -1 | cut -d : -f 1 | tr -d \#)
```

General Debugging Techniques



Exploit compiler features

- The compiler is your friend. A good compiler can perform static analysis in your code. Static analysis can help in detecting a number of basic semantic problems, e.g. type mismatch or dead code.



```
#include <iostream>
using namespace std;
int main ()
{
   count <<"Hello World!";
   return 0;
}
```

```
010101010101010101010
101010101010101010101
101010001001010101010
101010101010101010101
101010101010101010101
```

Read the right documentation (i.e. RTFM)

The relevant documentation for the task, the tools, the libraries and the algorithms employed must be at fingertips to find the relevant information easily. As far as documentation is concerned, the most important distinction is between tutorials and references.

- A tutorial is a pedagogical paper, usually with plenty of examples, its first aim is to convey ideas about the subject.
- Reference manuals, on the contrary, are comprehensive and exhaustive descriptions, which allow to find the answers to questions through indexes and cross-references.

The Print Technique

Despite its popularity, this technique has strong disadvantages.

- Code insertion is temporary, to be removed as soon as the bug is fixed. A new bug means a new insertion, making it a waste of time. In debugging as well as in coding, the professional should aim to find reusable solutions whenever possible. Printing statements are not reusable, and so are deprecated.
- Printing statements clobber the normal output of the program, making it extremely confused.
- They also slow the program down considerably: accessing to the outputting peripherals becomes a bottleneck.

Finally, often they do not help at all, because for performance reasons, output is usually buffered and, in case of crash, the buffer is destroyed and the important information is lost, possibly resulting in starting the debugging process in the wrong place.

Logging

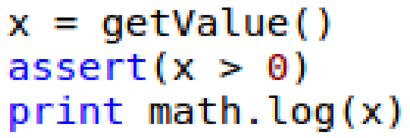
Logging takes the concept of printing messages, expressed in the previous item, one step further.

- ✓ Logging is a common aid to debugging. Everyone who has tried at least once to solve some system related problems (e.g. at machine start-up) knows how useful a log file can be. Logging means automatically recording information messages or events in order to monitor the status of your program and to diagnose problems.
- ✓ It is heavily used by daemons and services, exactly because their failure can affect the correct operation of the whole system.
- ✓ It can even form the basis of software auditing, that is the evaluation of the product to ascertain its reliability.
- ✓ Some languages/libraries can provide this functionality: Logger objects (Python), log4j (Java), FLIBS (Fortran), log4c (C) , log4cpp (C++), etc.

Defensive programming and assertions

Assertions are expressions which should evaluate to be true at a specific point in the code.

- If an assertion fails, a problem was found.
- The important point to remember about assertions is that it make no sense to execute a program after an assertion fails.
- Writing assertions in the code makes assumptions explicit.
- Since assert is a macro, it can be easily removed from the final version of your code by compiling it out.
- Example:



```
x = getValue()
assert(x > 0)
print math.log(x)
```

The Debugger

- When every other checking tool fails to detect the problem, then it is debugger's turn.
- A debugger allows working through the code line-by-line to find out what it is going wrong, where and why.
- It allows working interactively, controlling the execution of the program, stopping it at various times, inspecting variables, changing code flow while running.

Debugger Features - Breakpoints

- Breakpoints stop program execution on demand: the program runs normally until it is about to execute the piece of code at the same address of the breakpoint. At that point it drops back into the debugger to look at variables, or continue stepping through the code.
- Breakpoints are fundamental in interactive debugging, and accordingly have many options associated with them.
- They can be set up on a specific line number, at the beginning of a function, at a specific address, or conditionally.
- After stopping the program as a consequence of a breakpoint, a debugger can resume its execution.

Debugger Features - Watchpoints

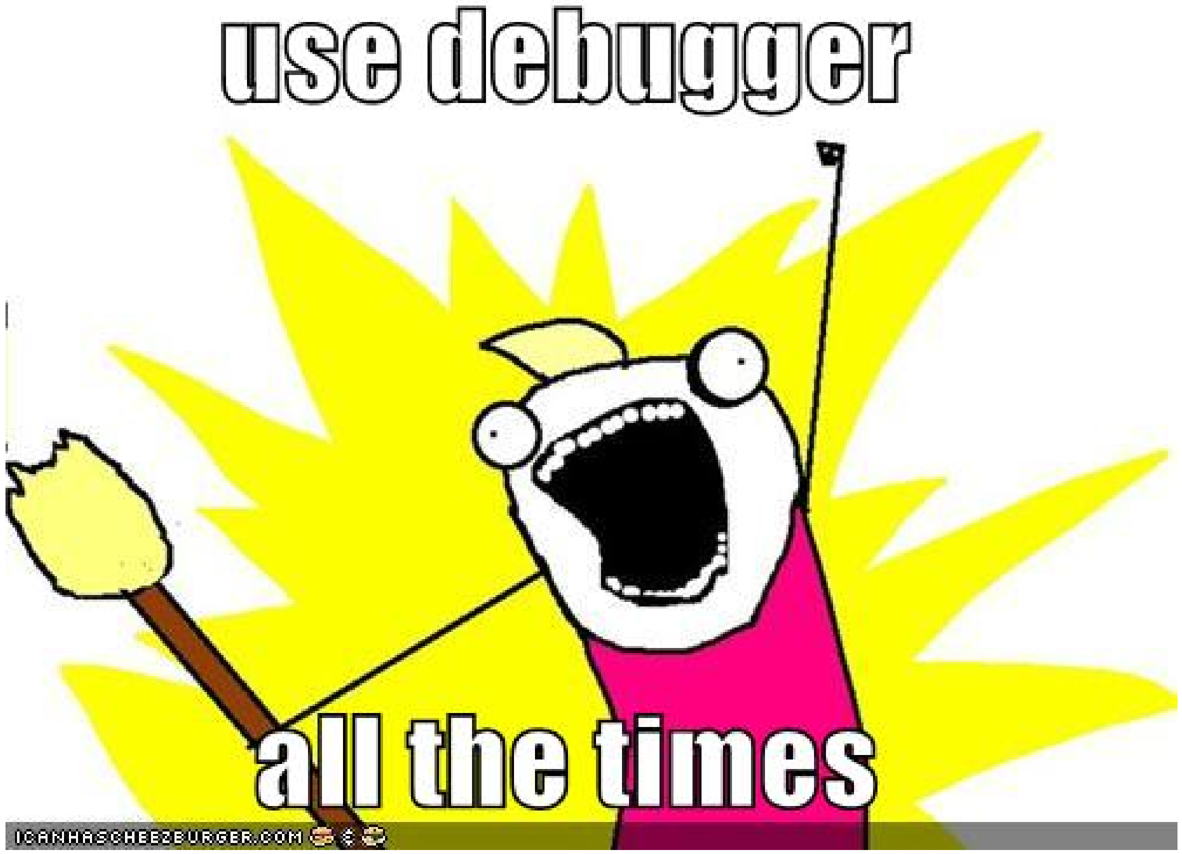
- Another important feature that all decent debuggers must offer is the possibility to set watchpoints.
- Watchpoints are particular type of breakpoints which stop the code whenever a variable changes, even if the line doesn't reference the variable explicitly by name.
- Instead, a watchpoint looks at the memory address of the variable and alerts the programmer when something is written to it.

Debugger Features – Step in, out, over

Once you have reached a breakpoint, you can control specifically how you want it to reenter your code. The step operations available to you include:

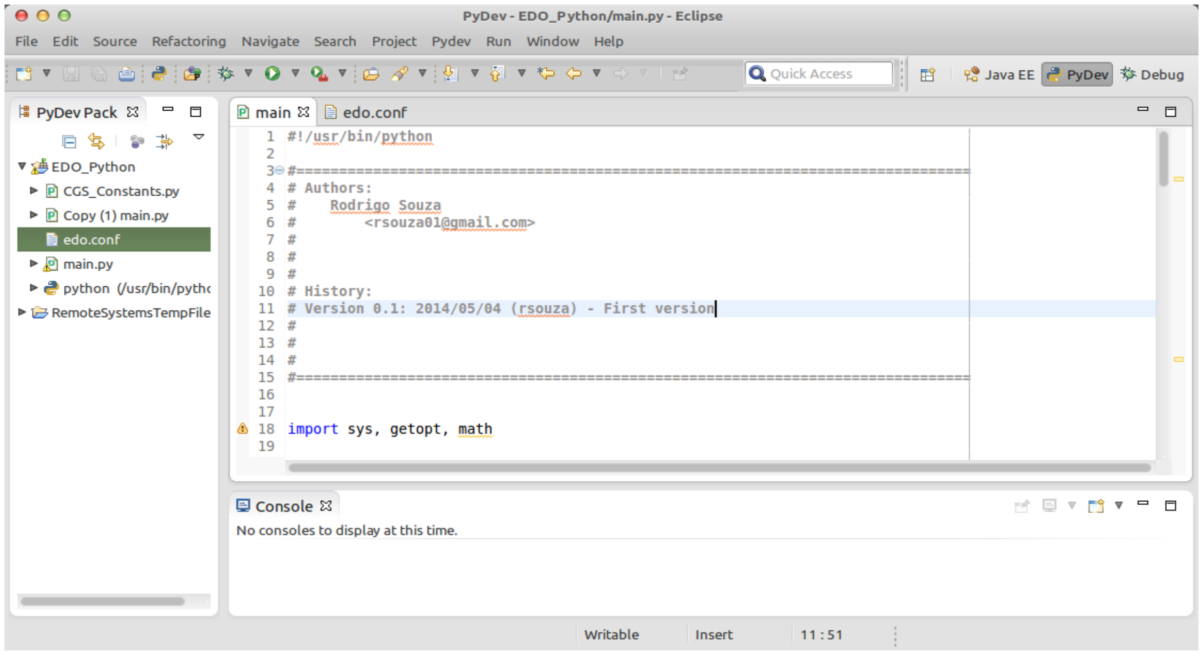
- **Step Into:** executes a single program statement at a time. If the current execution point (indicated by an arrow, usually) is located on a call to a method, the Step Into command steps into that method and places the execution point on the method's first statement.
- **Step Over:** enables you to execute program statements one at a time. However, if you issue the Step Over command when the current execution point is located on a method call, the debugger runs that method without stopping (instead of stepping into it), then positions the execution point on the statement that follows the method call.
- **Step Out:** will finish executing a method call you have stepped into, and return to the next executable line in the calling method.

Summary



Demonstration

Now I am going to demonstrate these concepts using the Eclipse IDE + Pydev Plugin (Python).



```
#!/usr/bin/python

#===============================================================================
# Authors:
#     Rodrigo Souza
#         <rsouza01@gmail.com>
#
#
#
# History:
# Version 0.1: 2014/05/04 (rsouza) - First version
#
#
#
#===============================================================================


import sys, getopt, math

```

```
No consoles to display at this time.
```