Software Development
Development Tools

Davide Balzarotti

Eurecom – Sophia Antipolis, France

Software Development Tools

1. Writing and managing code
2. Configuring and Building a program/library
3. Packaging and Distributing an application
4. Debugging and Profiling

Software Development Tools

1. Writing and managing code
   - Text Editors
   - Navigating source files
   - Diffutils
   - Versioning Systems
2. Configuring and Building a program
3. Packaging and Distributing an application
4. Debugging and Profiling

Text Editors

- In a world of text streams, the Editor is your best friend
- You spend a lot of time editing files, writing documents, writing source code, writing emails...
  - Choose an editor
    (no, they are not all the same - pick a good one)
  - Learn how to use it
    (really, the better you master the cryptic commands, the more productive you will be)
  - Use it all the time
    (using different editors for different task is not usually a good idea)
- Historically, Unix featured single line editors (`ed`) and stream editors (`sed` and `awk`)
- Now, you have two main choices

The Editor War

- VI vs. Emacs
  - They are both very good editors and they are both widely available
  - They are not for the faint of heart (it takes some time to learn them)
- Historical rivalry between users of the the two editors (editor war)
  - EMACS is “a great operating system, lacking only a decent editor”
  - VI has two modes: "beep repeatedly" and "break everything"

VI

- One of the first full screen editor (written by Bill Joy)
  - VIM is the version commonly used in many systems nowadays
- It was designed according to the Unix philosophy
- Ubiquitous. It's standard in all the Unix systems
  - If you have a shell, you have vi
  - Also emacs users must know at least the very basic vi commands
- Historically smaller and faster (does not really matter nowadays)
- vi is a modal editor
  - `Command mode` to type commands, `insert mode` to type text
  - Weird, difficult, un-intuitive, but tremendously powerful

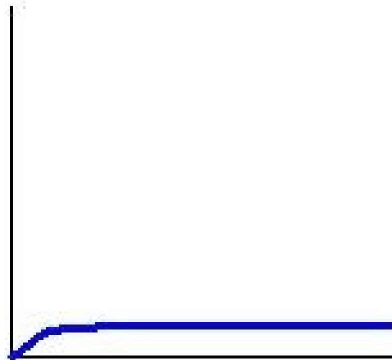
Emacs

- It is essentially an execution environment for a Lisp interpreter designed for text-editing
  - The best thing about emacs is that it can do everything (mails, calendar, text games, web browser, personal organizer,...)
  - The worst thing about emacs is that it can do everything (far from the Unix philosophy)
- Ported everywhere, but not always installed by default
- Use sequences of meta-key combinations for commands
- Easier to learn than vi, at least at the beginning

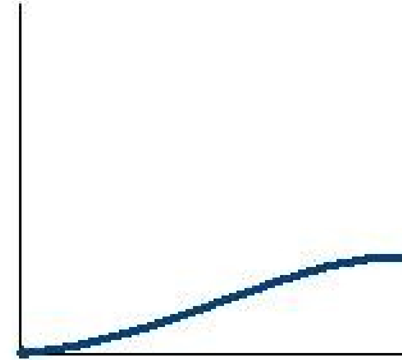
Don't expect Love at the first Sight

Classical learning curves for some common editors

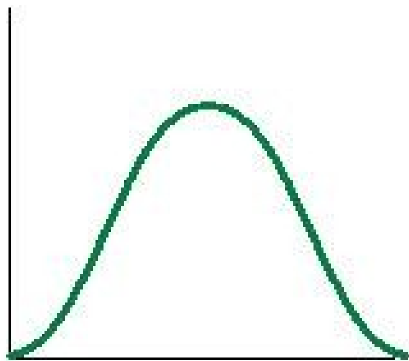
Notepad



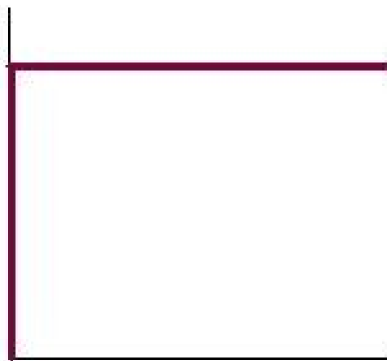
Pico



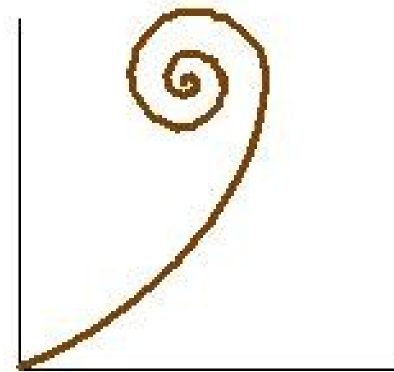
Visual Studio



vi



emacs



Navigating Source Code

- Navigating large projects with a text editor can be complicated
- `ctags` creates a tags file, which contains indexes of programming language identifiers
  - Tags files allow definitions to be quickly and easily located by a text editor (if the editor support them, that is true for the good ones)
  - Supported Languages:
    (by the exuberant-ctags package)
    Assembler, AWK, ASP, Basic, BETA, Bourne/Korn/Zsh Shell, C, C++, C#, COBOL, Eiffel, Erlang, Fortran, HTML, Java, Javascript, Lisp, Lua, Make, Pascal, Perl, PHP, Python, REXX, Ruby, S-Lang, Scheme, SQL, Tcl, Vera, Verilog, Vim, and YACC
  - For the list of tags it can generate for each language:

```
ctags --list-kinds
```

Using ctags

- `ctags *.c`
  Create a ctag files containing the symbols index
- `ctags -x *.c`
  Print the output in a human readable form

- ctags & your favorite editor
  - `vim -t tagname`
    Open the right file at the right position
  - `:ta tagname`
    Jump to tagname (with autocompletion)
  - `Ctrl-]`
    Jumps to the definition of the symbol under the cursor

Comparing Files

```
diff [options]  fileA fileB
```

- `diff` compares two files line by line
- If there is no difference, it says nothing (“silence is gold”)
- Otherwise it prints the different lines to the standard output in a particular format (the format depends on the options)
  - By default, it prints the output in *normal* format, suitable for a machine
  - `-u` produce the output in a *unified* format including context lines for each change
    - more human readable
    - most common format nowadays
  - The output of `diff` is called a patch, because it contains all the information required to transform one file to the other

hello1.c

```
void main(){
    printf("Hello World");
}
//end
```

hello2.c

```
#include <stdio.c>

void main(){
    printf("Hello World\n");
}
```

normal form

```
balzarot> diff hello1.c hello2.c
0a1,2
> #include <stdio.c>
>
2c4
<       printf("Hello World");
---
>       printf("Hello World\n");
4d5
<       //end
```

hello1.c

```
void main(){
    printf("Hello World");
}
//end
```

hello2.c

```
#include <stdio.c>

void main(){
    printf("Hello World\n");
}
```

normal form

```
balzarot> diff hello1.c hello2.c
0a1,2
> #include <stdio.c>
>
2c4
<     printf("Hello World");
---
>     printf("Hello World\n");
4d5
<     //end
```

Command format:
`linerange-f1 command linerange-f2`
- a = add
- c = change
- d = delete

`diff` is line based
Also if you change a single character, it substitutes the entire line

< identifies the first file
> identifies the second file

hello1.c

```
void main(){
    printf("Hello World");
}
//end
```

hello2.c

```
#include <stdio.c>

void main(){
    printf("Hello World\n");
}
```

unified form

```
balzarot> diff -u hello1.c hello2.c
--- hello1.c    2009-12-07 15:20:45.000000000 +0100
+++ hello2.c  2009-12-07 15:21:38.000000000 +0100
@@ -1,3 +1,5 @@
+#include <stdio.h>
+
 void main(){
-    printf("Hello World");
+     printf("Hello World\n");
 }
- //end
```

From one Version to Another

```
cat patchfile | patch original-file
```

- patch takes a patch file produced by the diff program and applies those differences to one or more original files, producing patched versions
  - It is usually able to automatically determine the format of the `diff` file
  - ignores leading and trailing “garbage”
    - You can feed patch with an email, an online forum message.. and it should work
  - `-b` – create a backup of the original files
  - `-R` – reverse the patch

Patching Entire Projects

- `diff` can also produce a list of changes between two directories trees
  - `diff -u -r directoryA/ directoryB/`
    Produce one patch file containing all the differences between the files in the two directories
  - By default, it does not include in the patch the full content of new files. When you need it, use the `-N` option
- `patch` tries to apply each single patch as if they came from separate patch files
  - It tries to guess the name of the file to patch. Much easier if you use diff in unified form
  - By default, it consider filename without the path. Use the `-p` option to change this behavior

Patching Entire Projects

- `-p num` or `--strip=num`

  Strip the smallest prefix containing num leading slashes from each file name found in the patch file. For example, supposing the file name in the patch file was: /u/howard/src/blurfl/blurfl.c

  - -p0 gives the entire file name unmodified,
  - -p1 gives u/howard/src/blurfl/blurfl.c without the leading slash
  - -p4 gives blurfl/blurfl.c
  - not specifying -p at all just gives you blurfl.c.
- Whatever you end up with is looked for in the current directory

```
balzarot> diff -Nur  myproj_v1/  myproj_v2/ > patch_1_2
balzarot> patch < patch_1_2
Can't find file to patch at input line 4
Perhaps you should have used the -p or --strip option?
…

balzarot> patch -p0 < patch_1_2
patching file myproj_v1/fileA
patching file myproj_v1/subdir/fileB
```

Sending Patches

- `diff` and `patch` play a vital role in software development and are at the heart of Open Source development
  - The program maintainer places the source code of the program on the Internet for free use
  - A user identifies a bug in the program and he fix it
  - To share the improved version with the community, he runs diff on the original version and his modified version to produce a patch that fix the bug
  - He then sends the patch to the original author, who can then approve the patch, apply it to his version, and finally post the corrected version

Three-way Comparison

- When two people have made independent changes to a common original, diff3 can report the differences between the original and the two changed versions
- The output of diff3 cannot be used for patching but..
- diff3 can also produce a merged file that contains both users changes (with warnings about conflicts)

```
diff3 -m mine older yours
```

- merge into “mine” the changes that would turn “older” into “yours”

main

line 1
line 2
line 3
line 4
line 5
line 6

bob

line 1
line 2
line 3
new a
new b
line 4
line 5
line 6

alice

line 1
line 3
line 4
line 5 xxx
line 6

> diff3 -m bob main alice

line 1
line 3
new a
new b
line 4
line 5 xxx
line 6

main

```
line 1
line 2
line 3
line 4
line 5
```

bob

```
line 1
line 2
line 3
bobline
line 5
```

alice

```
line 1
line 2
line 3
aliceline
line 5
```

```
> diff3 -m bob main alice
```

```
line 1
line 2
Line 3
<<<<<< bob
bobline
|||||| main
Line 4
=======
aliceline
>>>>>> alice
line 5
```

Conflict !!!

Working with Binary Files

- `diff/patch` are line-based and do not work with binary data
  - because there is no end of lines in binary files
  - If `diff` finds that one of the files contains binary data, it just prints if it is different from the other file
    - But it does not generate a patch !
- For creating patches for binary file you can use `bsdiff` and `bspatch` instead
  - Creates quite compact binary patches
  - Requires a lot of memory (>17 times the size of one file)
    - Suitable for patching executables, not to create a diff of ISO files