# Software Development Management

Mathieu Lacage - DREAM

# Facts

- Every piece of software is damn horrible. It is:
  - hard to use
  - buggy
  - late
  - hard to build
- There is a silver bullet to solve these problems.
- It is called Software Development Management.
- My grandfather used to call it Common Sense.

# Macro vs Micro Management

- Macro Decisions
  - do you need to use library X ?
  - when do you want to complete the project ?
  - what features must be implemented ?
- Micro Decisions
  - how do you add featurelet X to the project ?
    - hack it into objects A, B and C ?
    - create a new object D ?

# Macro Decisions

- If you try to avoid these decisions, you will regret it
- To avoid future problems, you need an ordered list of requirements. For example:
  - Platforms the software must run on (typically, OS)
  - Environments in which the software must be built (this includes the OS, compiler, library versions, etc.)
  - List of people who will write the software. Are they proficient in many programming languages ?
  - Typical user profile: researcher, Mr Smith, etc. ?
  - Availability date
- Each requirement has an influence on the time needed to create the software and the quality of the final product.

# Micro Decisions

- Every day, every second, small decisions must be made on how to build the software, how to add a function:
  - language-specific: requires knowledge of the language idioms
  - language-independent: requires knowledge of many development idioms to pick the right one
- Obviously, the hard part here is the “knowledge” issue: you need to acquire experience.

# How can you acquire more experience ?

- Even a level 5 guru started at level 0.
- A few ideas:
  - read code from open source projects
  - talk to your co-workers about your code and their code
  - try to ask yourself how you can improve your own code
  - obey the “7 Rules”

# *Rule 1*: SCM

- Source Code Management is not a luxury:
  - zip archives are worthless
  - named directories are worthless
- Required features:
  - easy log browsing (thus the need for decent log messages)
  - easy rollback to earlier versions
  - branch creation and merging
- Possible solutions:
  - CVS (http://www.nongnu.org/cvs/)
  - SVN (http://subversion.tigris.org/)
  - Mercurial (http://www.selenic.com/mercurial)

# *Rule 2*: SCM Tagging

- Tag the Repository for each release
- Yes, zip archives are worthless
- This is the only reliable way to reproduce the bugs reported by the users

# *Rule 3*: ChangeLog

- One change = one entry in the ChangeLog file
- Describe the goal of the change and how it was implemented in each entry
- Copy the ChangeLog entry in your SCM commit message
- It is possible to automate the task of generating the bulk of each ChangeLog entry
- Add an entry for each release

# *Rule 3*: ChangeLog (2)

```
2003-06-25  David Bordoley <borodley@msu.edu>
    * src/nautilus-shell.c: (open_window): Prefer an
      existing window for a location when opening a
      location from the command line and the user's
      preference is open in new window mode.

=== nautilus 2.3.5 ===

2003-06-23  Dave Camp <dave@ximian.com>
    * NEWS
    * configure.in: Bumped version to 2.3.5.
```

# *Rule 4*: Coding Style

- Do not invent one
- Do not change the existing coding style
- Use the coding style of the file changed
- Ideally, you cannot guess who wrote what
- Examples:
  - C:
    - http://www.gnu.org/prep/standards/standards.html#Formatting
    - http://lxr.linux.no/source/Documentation/CodingStyle
  - C++
    - http://geosoft.no/development/cppstyle.html
  - Java
    - http://geosoft.no/development/javastyle.html

# *Rule 5*: Documentation

- It is a communication medium: it is worthless if no one reads it
- Do not document that copy_object copies object:
  - the name is obvious (great choice of name btw !)
  - the source code is most likely trivial: we can read it
- Focus on the important issues:
  - global program structure
  - global dynamic control flow

# *Rule 6*: Automated tests

- Why using automated tests ?
  - they help catch regressions: new code is tested against known bugs.
- How to do it ?
  - do not test the normal behavior: while useful, it has a very low return on investment to cost ratio.
  - test side effects
  - focus on the easy stuff
  - add a test for each bug for which you can easily write an automatic test case
- Use a test coverage tool. i.e., gcov
  - http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

  Example: http://tinyurl.com/8gnse

# *Rule 7*: Code review

- It helps you focus on what you did and why you did it
- Reviewer:
  - How did you test ?
  - Why did you do this there ?
- Reviewee:
  - must be able to justify each line changed

# Recommended reading

- Books
  - “Refactoring, improving the design of existing code”, by Martin Fowler, Addison Wesley
  - “The mythical man-month” by Frederick P. Brooks, Addison Wesley
  - “Find the Bug: A Book of Incorrect Programs”, by Adam Barr (Chapter 2, which deals with code reading techniques, is a gem)
- Papers:
  - Agile Software Development Methods: http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf
  - A collection of great Software Design articles by Alistair Cockburn: http://alistair.cockburn.us/crystal/articles/alistairsarticles.htm

# Conclusion

- we can provide ChangeLog generation scripts
- we can answer questions on programming/refactoring/debugging/optimisation subjects
- we can provide advice on program architecture
- DREAM answers all emails: dream@sophia.inria.fr