Software Development Management

Mathieu Lacage - DREAM

INRIA
SOPHIA ANTIPOLIS

Outline

Two parts:

- First, an introduction to what you need to do to build software, why it is so hard, what needs to be done to try to make it easier

- Then, an outline of a few simplistic rules you should follow to avoid some of the biggest traps: “The 6 Rules”.

Facts

- It is hard to build software:
  - Code is hard to write
  - Code is harder to debug
  - Code is even harder to modify
- It is hard to use software:
  - Always too many bugs
  - Always missing functionality
- 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
  - what features must be implemented ?
  - when do you want to complete the project ?
  - do you need to use library X ?
- Micro Decisions
  - how do you add feature 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-independent: requires knowledge of many development idioms to pick the right one. One example is “Design Patterns”.
  - language-specific: requires knowledge of the language idioms
- 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 “6 Rules”

Rule 1: Source Code Management

- Required features:
  - Analyse development history
  - Rollback to earlier versions of the software
  - Work with other people on the same software
- Solution 1: Zip archives/named directories
  - You need to remember to create a 'backup' every morning
  - You need to remember which zip file/named directory contains version Y
  - You need to merge different versions by hand
- Solution 2: Use a real SCM tool
  - CVS (http://www.nongnu.org/cvs/)
  - SVN (http://subversion.tigris.org/)
  - Mercurial (http://www.selenic.com/mercurial)

Rule 1: Source Code Management (2)

- Concepts:
  - Commit: a single change to a file or set of files. Usually a user-specified “commit message” is associated to each commit.
  - Repository: a set of commits which represent all the changes ever made to every file
  - Tag: a unique global identifier in the history of changes
  - Branch: a diverging version of the software maintained in the SCM history (identified by an initial diverging point)
  - Log: a set of “commit messages” for a given file in chronological order
- Advice:
  - Commit regularly (every day/hour) with meaningful “commit messages”
  - Tag the Repository for each release (the only way to reliably reproduce bugs reported by users)
  - Create branches when you start work on a new feature

Rule 2: Commit messages

- Describe the goal of the change and how it was implemented in each commit
- It is possible to automate the task of generating the bulk of each commit message:

  http://www-sop.inria.fr/dream/prepare-ChangeLog
- If you use CVS or SVN:
  - Copy the commit message in a ChangeLog file ordered chronologically.
  - Add an entry in the ChangeLog file for each release

Rule 2: Commit messages (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 3: 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 4: 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 5: 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/ycwu94

Rule 6: 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 answer questions on programming/refactoring/debugging/optimisation subjects
- we can provide advice on program architecture
- DREAM answers all emails: dream@sophia.inria.fr