Software Development Management

A simplistic howto

Mathieu Lacage <lacage@sophia.inria.fr>
service DREAM

The goal of this presentation is to explain a few simple techniques I have found useful when working on different software projects and which I believe can be easily used by others. These techniques require neither a lot of time nor a lot of knowledge.

What is software ?

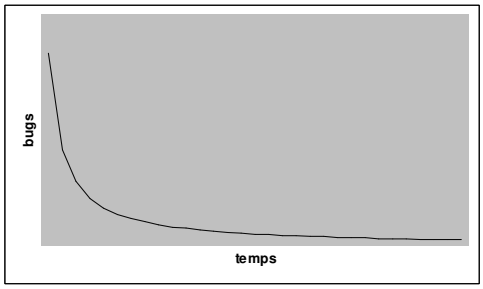
- Examples:
  - excel macros
  - javascript
  - graphical user interface for a database
  - real-time control of an articulated arm
  - numerical computation

  ➡ The goals, requirements, and know-how are extremely different

Some claim it is impossible (it is at least very hard) to create generic rules describing how to manage a software development project because each software development project is different. Clearly, the difference both in size and in the technical knowledge required to complete the project can be huge but there are a few things which all software projects share.

What is software ? (2)

- « Features »:
  - development methods (or lack thereof)
  - bugs are always present
  - all software engineers spend 80% of their time completing the last 20% of a software



INRIA SOPHIA ANTIPOLIS

Software development projects almost never follow any kind of method or structure which is surprising if one considers other engineering activities. I can see two reasons for this:

• A lot of developers hate and resist any form of management because they believe that what they do is « art ». As such, they are entitled to do whatever they want. Of course, this is completely wrong: there is no reason not to, at least, try to organize one's work.

• Developers have no experience or formal training in software development because this software is the byproduct of another activity.

All developers are too optimistic: they forget that writing code is the easy part of any project. They forget that they will (or should) spend 80% of their time debugging and testing the final product.

Even old projects (10 years old) uncover bugs that are several years old every week: think bind, sendmail.

What are the goals ?

- Complete current project ASAP
- As few bugs as possible
- Complete next project ASAP (maybe)
- Means to achieve these goals:
  - reuse of libraries and/or programs
  - write reusable code
  - write bug-free code (?!)
  - ...

It is obvious to any manager (provided he has a brain which is something developers tend to dismiss) that he might need to spend more time on the current project to be able to complete the next one within the given deadline. For example, he might decide to spend more time than strictly needed to test, debug, and cleanup a given piece of code, hoping to be able to reuse that piece of code in a later project.

While reusing code might save time (there is no way to be sure it will indeed save time), writing reusable code will drain time faster than any form of customer support and might not save any time later. As such, remember that writing reusable code is not a goal by itself. Completing the project on time is the ultimate goal.

How to achieve these goals

- Software development « management »
  - keep in mind the ultimate goal during technical discussions or when you must take a decision on your own: complete the projet asap within the given constraints
  - none of this is technically challenging. It is a (not so simple) matter of communication and organization
- Each and every project, small or big, can save time with such « management »

INRIA SOPHIA ANTIPOLIS

It is important to:

• clearly state the goal of any technical discussion

• archive all technical discussions with a summary of the decision taken and the *rationale* for this decision.

Some people will disagree with the decision but they cannot claim they did not know about it and explaining exactly why the decision was taken might help these people to happily accept the decision.

The following slides try to give an overview of how this principle of « keeping the goal in mind » can be used.

Decisions

- Macro-decisions:
  - will library X be used ?
  - when do you want to complete the project ?
  - what features must be implemented ?
- Micro-decisions:
  - how do you add featurelet Y to the project ?
    - hack into object A, B, or C ?
    - create a new clean object D ?

Imagine a 3D mesh editing application which allows you to edit a mesh displayed on screen with your mouse.

Macro-decisions

- Features:
  - end-user: feature list, user documentation
  - developer side: documentation
- Deadlines: prototype, final version, hard deadlines ?
- Development environment:
  - libraries, compiler, and OS versions
- Production environment:
  - different from development environment ?

Features: the more detailed, the better. Real life has taught me that it is rarely detailed enough because people just do not know what they want. The developers thus have to design the software such that it can be modified and adapted to the changes and refinements of the feature list. This is really hard. Such is life. (for example, you might wish to add loading of new mesh file formats in our 3D mesh-editing application after it is completed)

Development/production environment: as they grow apart, the amount of work to achieve proper testing and debugging on production systems grows. Different library versions is a big-enough difference to make a developer 's life become Hell On Earth™ and waste huge amounts of valuable time. (If you use OpenGL to render your 3D meshes, you might use different versions of the OpenGL or GLX libraries on development and production systems. Worse, you might be using libraries from different vendors with different bugs on different hardware...)

Macro-decisions (2)

- Each decision has a direct and fundamental influence on:
  - architecture
  - maintainance cost
  - reuse cost
  - initial cost
- Each decision must be weighed against its consequences

➡ This requires experience to get right

Weighing decisions is hard: if you doubt about your ability to weigth these decisions and/or you do not know enough about what the project really needs, it is easier to ask and get a written (email) decision from your boss. This will help you to describe what you need to do to complete your job and will make your everyday-life easier.

To do this, you must provide a thorough description of what the different decisions are and what you think their consequences are on the outcome of the project (deadlines, feature list, …)

If none of this is done, a set of implicit decisions will be made which are very unlikely to match the needs of the users thus making your work literally worthless.

For example, if you do not specify what kind of editing features must be available to the users of our 3D mesh-editing application, the developer will implement the features he considers most useful which are unlikely to be those real users expect. If you specify too many features and not enough manpower, the application will be available 2 years later which is probably too late.

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

➡ If you have experience, it 's easy. If you do not you can learn easily

Let say you want to add a new tool to edit the 3D mesh of our 3D mesh-editing application: you want to apply arbitrary rotations around a given point to the mesh. From a UI point of view, this means selecting a point as the center of the rotation and then real-time dragging the mesh around this point with the mouse pointer until you reach the position you are aiming for.

Once you know what you want, you need to figure out how to implement it. Here, it might be as simple as adding a new tool class which implements a few methods to handle mouse events. This can be implemented in a number of ways: you might wish to create an MVC (Model/View/Controler) architecture to decouple the mesh representation code from the mesh model code.

Doing so would involve first knowing that such interesting programming models exist (this is a language-independent development idiom). It would also involve knowing how you can implement this programming model in your favorite language. For example, in C++, you might want to enforce the existance of only one instance at runtime of this rotation-tool class: this can be done with the singleton pattern but you need to know how to best implement it in C++. This is a language-dependent development idiom.

How to acquire experience ?

- No need to be a level 5 guru (gurus always start at level 0)
- A few ideas:
  - be very rigorous
  - question yourself
  - discuss your decisions with others
  - read the code of others
  - obey the following 7 basic rules

The key to improve your experience and knowledge is to always try to understand how your code works. These days, it is increasingly easy to write code without really knowing how and why it works: take a Java program which allocates thousands of objects without freeing any of them. The Garbage Collector destroys them when they are no longer needed most of the time: the developers thus do not really need to understand how the garbage collector works.
However, there exist Java APIs which require explicit object destruction (most often, Java APIs which allocate system ressources). If the developer is not used to ask: « when is my object destroyed, and by who ? », it is likely he will entirely forget to explicitely destroy the object. This might not be a problem at first but it might cause the application to fail after a 3 hour-long run in mysterious ways… Definitely not a good thing.

Rule n°1

- Source server
  - CVS, arch, subversion, bitkeeper, perforce
  - yes, zip archives are worthless
  - must allow concurrent work on the same file with easy merging
  - must allow branch creation and merging
  - must allow history browsing

Rule n°2

- Tag/Branch the source repository for each release
- Yes, numbered zip archives are worthless
- This is the only reliable way to reproduce bugs reported by users

Rule n°3

- ChangeLog
  - one change = one entry in the ChangeLog file
  - describe the goal of the change and how it was implemented in each entry
  - use the same entry as your source repository commit message
  - automatically generate the entry skeleton
  - add an entry for each release

This file will allow you to find the list of modifications between multiple versions of your software quickly and easily.

Rule n°3 (2)

- Example

```
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 n°4

- Coding style
  - please, do not invent one (no NIH)
  - do not change the existing coding style
  - use the coding style of the modified file
  - ideally, you cannot guess who wrote what
  - ex: in C, the Linux kernel, GNU, Gnome

Rule n°5

- Documentation
  - communication medium: it is worthless if no one reads it
  - please, do not document that copy_object copies objects:
    - the name is obvious
    - the source code is probably trivial: we can read it
  - focus on important stuff:
    - global program structure
    - global dynamic control flows

There is no need to worry about writing documentation in UML or XML or any buzzword-full technology. What matters is writing concise, focused documentation. A small diagram with a few arrows done with xfig or dia can be much more effective than a heavy UML diagram which you need to print on A3 paper to read. I personally like small text-based files which contain ascii-art diagrams: I find this low-tech aspect helps you focus on the content.

The program structure documentation should present a broad outline of how objects or functions interract. For example, it would make sense to start this outline by describing how objects are created if you do not use the normal C++ new operator (maybe you use a factory and there exist a static or dynamic database of object factories in your program), how they are destroyed, what memory management model is used throughout the code, and so on…

The control flow documentation should present an outline of what the call graph looks like at runtime: function A (implemented in file X) is called by function B (implemented in file Y) and then calls function C to achieve feature Z.

Rule n°6

- Automated Tests
  - Ex: data decoding
  - goal: find bugs in code modifications
  - do not test normal behaviour (it is useful but has a very low return on investment ratio)
  - test side effects
  - focus on the easy stuff
  - add a test for each bug for which you can easily write a test-case

Automatic testing takes a lot of time, can be very hard but can save you even more time.

For example, it is almost impossible to automate the testing of graphical user interfaces: one often has to write test plans which look like: « click on button A, wait .5s, make sure window closes, look at status bar, it should report 100% ».

However, it is really easy to test the correct behavior of a data decoding function: all you have to do is to accumulate a big-enough dataset of coded/decoded pairs, feed the coded data to the function and make sure the decoded version matches the original decoded data.

The goal here is to avoid regressions in code which was modified to add new features or to fix bugs. The automatic tests will ensure that the behavior of the function has not changed on the test dataset. This is useful because Murphy said (maybe he did not but he should have): all modified code is buggy.

If you hesitate, write white-box tests: these use both the public and the private APIs as opposed to black-box testing which uses only the public interfaces. White-box tests are easier and faster to write.

Example: a method to ensure the coherence of the internal state of the object can be called after a number of normal method calls

Rule n°7

- Reviews
  - for each commit, request review from another engineer
  - reviewer:
    - how did you test this change ?
  - reviewee:
    - justify each line changed
- Help you to focus on what you did and why you did it

Peer-review helps improve the quality of the code checked in the source repository: if done correctly, this can diminish the amount of bugs introduced by the developers themselves during debugging.

The other very important aspect of peer-review is that it helps all engineers contributing to the project to work on and read code from other areas than what they are used to. Peer-review is also a very effective way to help young engineers acquire the wisdom and experience of the older engineers by working more closely with them.

Some argue against such development methods and claim this kind of behavior wastes a lot of valuable developer time. If properly implemented, (that is, if the developpers happily agree that this is a good thing and if they are very rigorous about it), it does save a lot of later debugging efforts and, by improving the technical knowledge of the developers, helps improve their productivity.

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
- Papers:
  - http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf
  - http://alistair.cockburn.us/crystal/articles/alistairsarticles.htm

Conclusion (2)

- I am no crook:
  - I can review two people
  - I can provide ChangeLog generation scripts for C, C++, and Java
  - I can help CVS server configuration (not administration)
  - I answer all emails: lacage@sophia.inria.fr
- DREAM answers all emails too:
  - dream.permanents@sophia.inria.fr