Large-scale OSS software engineering using GTKAda

Andrew van der Stock

ajv@greebo.net

In memory of Sang

January 18, 2001

Abstract

Large end-user applications have not been a successful area for OSS development, as it takes altogether a different mindset, as well as long-term dedicated warm bodies to bring a large project to fruition. The "scratch an itch" motivator generally suffices on successful smaller projects, such as Eric S. Raymond's *fetchmail* or *pnm2ppa*, a HP print post-processor for *GhostScript* that the author has been involved with. However, when a small project becomes large, for example the Linux kernel, this model works to the detriment of the project as a whole. End user applications are the tractor applications for an operating system. *KOffice* is doing a sterling job on the main office productivity applications. They have one gap: project management, which is my personal "itch".

1. Introduction

As a test of my ideas on OSS software creation, the author set out to deliberately modify the commonly used – and reasonably successful – "release early and release often" OSS software development model, which is possibly best described in Raymond's seminal work Cathedral and the Bazaar. For this project, the author added or layered software engineering practices, and in particular the rather unfashionable idea of software architecture into a major scale project.

Software is useless if it has no users, or is not available on the platform the user chooses or is forced to use (a legal OSS DVD player anyone?). Project managers would like everyone to have project management software; it reduces their own effort as well as increases the awareness of the project milestones and deadlines among the entire team. At the time of writing, the current project management tool costs over $780, so giving project managers a compelling "free" solution would be a major boon to any organization using project management methodologies.

However, the typical project manager is unlikely to be computer literate in the geek sense. They use tools, resources and methodologies to deliver projects on time and on budget. They are unlikely to install a free operating system to use even the best tool in the world, when everything they must work with is on the other platform (for now). A core requirement for this project suddenly became cross platform delivery, not just interoperability.

The largest break with OSS tradition in this paper is exploring software engineering and software architecture. To provide a meaningful difference, the author chose to use GtkAda, which provides Ada95 bindings to Gtk+, which is the cross platform GUI kit used by another old OSS favorite, the Gimp. By using a modern strongly typed, modular and object-orientated language, the author hopes anyone following the architecture can contribute to the project in a far more structured way, producing desired results far earlier than is otherwise possible.

2. Software Engineering

Why even bother with software engineering? Most projects seem to be currently created or led by volunteers in their early twenties who are reacting against their formal training (if they've had it), or simply ignorant of current best practice. As one diarist on Advogato noted, these "leaders" are like lunatics leading the asylum.

The author is a security architect currently contracted at a large Telco. Recently he introduced UML diagrams to model process flow into the documentation for a major project. Process flow is all-important as semantic attacks are the most likely form of attack, with the highest impact if they succeed. The other young architect working with the author was not even aware of UML, or of the need to model our processes prior to further our investigations. In the author's opinion, this lack of software engineering knowledge that hampers large scale development.

Software engineering can be considered a heavy-weight development methodology, as coding is rarely more than 66% of the total development time.

One or two collaborators devise a complete picture of the project. At least 33% of the time is spent developing specifications, drawn from the goals, requirements, and developing the architecture. By working through the architecture, a complete picture of what needs to be coded is known before the first line of code or first dialog is drawn.

Once coding starts, best practice must accompany good coding. This includes test modules, regression tests, code revision control, bug tracking, documentation, and so on. These skills are the least practiced in real life, and yet make or break commercial projects of

3. What software engineering is not

Software engineering is not just CVS and test modules. Software engineering is not a magic bullet – plenty of large commercial projects fail (at least 25% of them according to recent studies).

4. Users

Users are critical success factors in graphical programs. Too often, open source projects fill the purpose to which their developers' crafted them, with a user base of one. *Human-computer interaction* (HCI) is never considered, even on those projects where the entire purpose is to provide a graphical front end to a currently UI-less or text-based tool (examples include *linuxconf*, *kppp* or *make xconfig*).

Open source developers almost universally develop extremely poor interfaces due to an aggressive lack of HCI skills. This lack of skill ranges from "where's the command line?" or "what's wrong with $EDITOR" through to "GUI's are for wimps" right through "hey, it works for me". There is an obvious disconnect here, one which will hamper deployment of alternative operating systems such as NetBSD or Linux for some time to come.

Using a style-guide will help. Even though both the Gnome and KDE style guides are flawed from a HCI perspective, using them is a step forward for all programs that adhere to a particular style as users have less to learn when introduced to the desktop.

The question open source developers must ask themselves is "would a computer illiterate user be able to use this?" If it's an advanced tool for administrators ask "how can I help prevent mistakes from happening?" Does the tool have stability? Does the user face the least surprise when they do something? Can they always cancel? Does cancel mean cancel? Is there an easier way to do this? Are the most common actions the least difficult and quickest to perform? Does it scale? Does the interface always stay responsive?

To assist users, the initial HCI style guide chosen for the program is KDE simply because KOffice is likely to ship well before anything else as useful comes along. The project's architecture will use controller-view-object, which allows the interface to be changed easily, depending on platform, as well as allowing internationalization.

5. Goals

Goals are by far the easiest and hardest portion of a project. Unnecessary goals can cripple a project ("must be written in Java"), but if some goals are not articulated, the project may not satisfy end users. Goals are high level *desires* that help draft requirements that lead into architecture and ultimately design. A goal will help move a project from being just an idea ("there's no competitor to Microsoft Project" or "I'd like to be able to work on projects in Linux") to being a particular brand of reality.

Toolkit-style open source projects have a single goal: *fetch mail from a mail server*, or *translate pixmaps into printer-ready data*. Once they perform their primary goal, they end up having cruft attached until the weight of the additions requires a rewrite or major rework (XFree86 is a good example of this).

Large programs, such as a word processor, require multiple goals as some of the goals compete with each other. For example, KWord started life as a simple *FrameMaker*-inspired word processor. The initial goals might have been to make a frame-based document processor but as time moved on, other high level goals became apparent, such as "must be internationalization-capable".

The goals of this project are manifold. For the author the primary goal is to explore software engineering in open source. But that goal doesn't help project managers migrate away from the only product in the market today. Therefore the revised primary goal of the project is "allow large projects to be worked upon easily on any platform in any human language they choose".

6. Requirements

The requirements document is typically a short bullet list of requirements and assists the architect with including only those features that must be there for the program to function as intended.

In the author's line of work, requirements documents rarely go beyond 10 pages, and this is a fair indication of the high-level nature of requirements documents.

A requirement list for this project includes:

- Must be easily internationalized
- Must be portable
- Must allow projects of over 10,000 tasks to be loaded within two seconds on low-end hardware

These requirements are generally categorized into sub-headings, which lead themselves to be the major portions of the architecture. They do not specify that data must be stored in one-way linked lists or in a database. A compliant program to these specifications may be written in any number of ways and still be a workable program.

7. Architecture

Architecture allows coders and maintainers to look at a program's documentation and quickly ramp up and be productive developing on that program.

A large project such as this requires *conceptual integrity*, a term used by Brooks in his seminal *Mythical Man Month*. Conceptual integrity is key to successful products. Programs with good conceptual integrity do things both internally and externally in a particular fashion that makes it easier for developers and users to perform their work. Programs without conceptual integrity are not only hard to work on; they also typically confuse or confound their users. Typically one person has to be responsible for conceptual integrity, even if many people have been involved in decisions leading up to the time of the final *in* or *out* decision being made by the architect(s).

Good architecture allows and plans for change. No program or system stands still, particularly during development. Good architecture also stands the test of time: does this feature mean anything in three years? Good architecture works through issues that are relevant to the user's needs: can I move this data to another product if necessary,

At the time of writing, the program is still in architecture development.

8. Design

Design is the nuts-and-bolts of specification. Design documents the lowest levels, from module design (ie "the store object manages all persistent objects with the following methods") right through how the different parts of the architecture inter-communicate. Design is vital, and must be conducted not only directly after architecture but also throughout the life cycle of the project to ensure that it is kept in sync with reality.

Good documentation will allow the coders who normally work on open source projects today to come in, program a bit towards the end goal and know that they finished important work, albeit at a much higher quality bar than before, with no wasted effort.

9. Test Plans

Ada is unique among languages that I've studied and programmed in over the years. It's the only language I've come across that the standard reference texts all include test plans, test modules, and testing with their code samples. Imagine "Hello, World!" and add a test or three to it.

This mimics real life, where quality and reliability matter. When you change a tire on the highway, you don't just jump in the car and accelerate to 100 km/h and hope that the tire is firmly fastened. If you're like me, you change the tire, count the nuts to ensure that they are all on, pull and push at the wheel to ensure that the nuts are all done up properly before letting the jack down, pack up and drive off slowly with the CD off until satisfied that the tire is going to make the remainder of the trip. In the fast moving IT world, this rarely happens. Compiles? Ship it.

The sad thing is that SQA is a well-known art, and one that's been with us for more than 40 years. There's nothing difficult about it. Unfortunately, this aspect of programming is not firmly shoved down young programmer's throats at CS courses around the world.

In Australia, there is a wonderful act called the Trade Practices Act. If you sell packaged software, the TPA applies, regardless of what rubbish lawyers stick on the side of the box. The TPA has more than just implied warranties. If your software is broken, the consumer has a range of remedies open to them, most of which involve yet more lawyers, sadly. SQA will become a necessary part of doing business in the future in most countries lacking DCMA-style bad software protections.

10. Why Ada?

Ada is well off the beaten path in terms of language in the open source world. Many successful projects use alternative languages, such as perl or python, or even Modula-3 (cvsup). This project uses Ada for the same reasons that many programmers choose to avoid C – reliability, speed of development, and supporting post-1980 software engineering practices, like objects, inheritance, code re-use, and concurrency.

The author chose Ada because it is:

- Robust;
- Reliable;
- Object-orientated;
- Extremely typed;
- Has native threads;
- Portable;
- Relatively fast;
- Enables programming by contract

among many other features. The project will simply recompile on NetBSD/alpha from the native Win32/x86 version.

Running speed on this project is secondary to portability; the author does not have the time necessary to develop for the primary platform of the users (Win32) and his preferred platform (NetBSD). Ada is not necessarily slow, however, *gnat*, the free Ada implementation, is far from properly optimized compared to commercial compilers.

Ada has been used for many years as the preferred embedded language for command-and-control weapons systems where reliability is the key before any other consideration. The output from most Ada compilers is very equivalent to the best C backend emitters in the business and can even use JVM's to run the code.

However, the main attraction to using Ada is there is not a large pool of developers. Therefore all the old bad habits of C programming which seem to infest C++ and Java programmers would be difficult to take across to a Pascal-style language. Programmers would learn Ada and pick up good coding habits as they went along.

The learning curve of Ada is about the same as Java (the author has at various times tried to learn both languages). From past experience, it takes a decent programmer about a month to become usefully proficient in a new language. If the ramp up time coming to the project can be reduced by this amount, the project gains decent programmers, and the programmers learn new skills that they can re-use when returning to their preferred languages. With some luck, they may fall in love with Ada and continue to use it. ☺

11. Why GTK+?

Selection of a graphical toolkit came only after requirements had identified portability as key, which led to language selection. For various reasons, using Ada seemed a sensible choice to avoid a range of programming issues, and Ada had a limited number of graphical toolkits (in fact, the language is devoid of input/output to assist with embedded uses). GTKAda is the most advanced and satisfied the portability key.

GTK+ is implemented in C, and is billed as objected-orientated. Sadly, this is not borne out by further inspection. The GTK+ framework takes none of the responsibility for object management away from the developer, nor does it really implement the basics of object orientation: inheritance, object reuse and strongly typed objects. For these reasons, and its association with very poorly implemented software (the Gimp is a HCI nightmare) the author was initially reluctant to use it at all.

However, GTKAda implements a thin shim over the top of GTK+, and relieves some of the object management tasks away from the developer. Use of GTK+ through Ada is made partially object orientated:

```
if Dial.Adjustment /= null then
   Gtk.Handlers.Disconnect (Dial.Adjustment, Dial.Changed_Id);
   Gtk.Handlers.Disconnect (Dial.Adjustment, Dial.Value_Changed_Id);
   Unref (Dial.Adjustment);
end if;
```

In addition, GTKAda is safe with Ada's inbuilt thread implementation, allowing multiple user interface threads to provide implied responsiveness (think BeOS-style at least one thread per window model).

The author inspected WINE, Java, and KDE as alternatives, but discarded them for various reasons, including portability and OO features. Java was the closest match to GTKAda in terms of features, but typically produces applications that do not look like anything else on any host platform. GTK on Windows can use Win32 widgets, but this project chose not to use them in favor of fast and easy portability.

12. The Project

Project seeks like-minded software engineering geeks for good time. Your place or mine.

The project is a project management tool, similar to Microsoft's actually rather excellent Project. There are surprisingly few commercial competitors, and no comparable free tool available. The author made contact with a few interested developers earlier this year, but found to his dismay that one wanted to be the project manager, and the other wished to push Java but not much else. That effort quickly dissolved.

The status of the project at the end of 2000 is that the project is in architecture. Once that has been completed, the "build every day, release-early-release often" phase will initiate.

13. Other Projects

Perl 6 is the only other project the author is aware of that has a large amount of software engineering dollopped on top. After many years, Perl is in desperate need of a complete rewrite, and this is a fine effort. Kirrily Roberts' talk on Perl 6 will be an eye-opener for many of you, I encourage you to attend her fine session.

If you are aware of other software-engineering led projects, I'd be very happy to hear about them; mail me at ajv@greebo.net

14. References

Due to a death in the family, I am without access to my technical library. This is a truncated form of the references section. A future version of this paper, with full references will be available from my web site

http://www.greebo.net/

1. Feldman, Michael B., Koffman, Elliot B., *Ada95: Problem Solving and Program Design*, Third Edition, © 1999 Addison Wesley, ISBN 0-201-36123-X
2. Harlow, Eric, *Developing Linux Applications with GTK+ and GDK*, © 1999 New Riders ISBN 0-7357-0021-4
3. Ada web site: http://www.adapower.com/
4. GTKAda web site: http://www.adapower.com/gtkada/
5. GNAT ftp site: ftp://cs.nyu.edu/pub/gnat/