Limits to Collaboration

Mark Könnecke
Laboratory for Neutron Scattering
Paul Scherrer Institut
5232 Villigen–PSI
Switzerland

November 2, 2004

Abstract

The series of NOBUGS conferences has resulted in little collaboration so far. In this contribution I will present some thoughts why this is the case. There are even good reasons why a home grown solution can be superior then a similar external solution. Based on these thoughts some ideas how to improve collaboration will be discussed.

1 Introduction

The original aim of the NOBUGS series of conferences was to enhance the collaboration between IT staff working at the various X-ray and neutron scattering facilities. Now we are holding the 5th installment of the NOBUGS series of conferences and there is little progress in terms of collaboration. There are isolated pockets of collaboration but we still see a lot of duplication of efforts. Why is this so? Are there good reasons for everybody reinventing the wheel again? In order to examine this question some more I started reflecting on my experiences with software collaboration gathered so far.

Before starting let us limit the scope of this paper. This paper is about the consequences of collaboration on major mission critical systems. This is not about small utilities. Most examples in this paper relate to data acquisition systems but many statements are also true for data analysis software.

2 Reflections on SICS

SICS, the SINQ Instrument Control Software, is the instrument control software used for controlling most neutron scattering instruments at the spallation source SINQ at PSI. It is a major system comprising 111 kilo lines of ANSII–C source code plus about 20 kilo lines worth of Java code. SICS is now in operation since 1997 and the author is in the favourable situation of looking at this system with the advantage of hindsight. There is also some experience in teaching SICS to new staff at our facility and at a collaborating facilty, ANSTO in Australia.

The first lesson the author learned quite quickly is that a data acquisition system is a work which is never done. There is more or less constant developments:

- New instruments have to be accomodated for
- New hardware has to be supported
- Error handling tactics have to be adapted to ageing hardware
- New ways of performing measurements are introduced and have to be supported by the software.
- New instrument scientists come with different preferences and these preferences have to be accomodated for.

This also holds true for data analysis software; new methods of analyzing data are invented all the time and different people have different preferences when doing their analysis.

One important text on software engineering, Frederik Brooks, "The Mythical Man Month[1]" recommends to plan to throw your first design away. Well, there rarely is a chance to do this. A working data acquisition systems represents such an investment in work, bug fixes and experiences that a new system only becomes feasible when the current system is either completely broken or the facility undergoes a major refurbishment. This implies that the software has a life time in excess of 20 years! Now, writing a software which is able to accomodate 20+ years worth of changes is a tall order.

The lesson learned from all this is that SICS is more like a programming environment which has to accomodate 20+ years worth of changes. As such is shares many aspects with operating systems, compilers, library frameworks and similar software. The author readily admits that SICS was never designed and documented as a programming environment from the start with. SICS was refactored towards a programming environemnt and the documentation was improved after this realization. I also looked around and failed to find systems which have been designed with the programming environment aspect in mind. But this might be a lack of knowledge on the side of the author. The exception is data analysis, were the field of programming environments is covered by commercial packages like IDL or matlab, to name just a few, or public packages like Scilab or octave.

3 The Need for Software Change And its Implications for Collaboration

If we accept the notion that our software has to be changed all the time, what are the implications for software collaboration? Let us consider the situation which arises when collaboration actually happens and a new system has to be installed at a facility. The job is not done when the software has been compiled and installed (which occasionally already is a major hurdle). Then the adventure of climbing the learning curve begins. And this is not done when the usage of the foreign system has been learned. In fact learning the usage of a system corresponds to climbing the foothills. The need for change requires to become far more intimate with the software. The design of the software has to be studied. Any software system not only encloses functionality and code but also the mindset of the original authors: how they think about programming, how the original authors solve

problems, how they design programs. All this has to be studied when a software is adopted. One has to learn how to think in terms of the system. Data structures and code must be studied in order to adapt the software to local needs, fix bugs or enhance the software. A lot of communication with the original authors is involved, too. At least changes to core components of the adopted systems have to be discussed with the original authors if the co, standing for cooperation, in collaboration is taken seriously. If there is no communication with the original author, chances are that an incompatible change is introduced. This makes it difficult to install a new, enhanced, version of the software from the original author.

Learning a new system thus is a major effort. It can be done but it is hard. And quite often the documentation necessary for understanding a system so intimatetly is missing.

In the light of the need for change it even becomes difficult to choose a system to adopt. Comparing feautures is not enough, the adaptability of the software in question to local needs has to be judged as well. This also requires a careful study of the design and the csource ode of the software. This can be an awful lot of work.

4 Time and its Implications on Software Collaboration

Another problem is time. Software technology and methodology advances at a rapid pace. This implies is that a software system designed five years ago looks old by todays standard, a system designed ten years ago may even look archaic. And who wants to burden himself with the maintainance of an outdated system? This is another reason for rejecting collaboration.

In the course of time a series of programming languages became fashionable:

- Scientific software all started with VAXtran. From this era some useful fortran code was salvaged.
- Unix brought ANSII–C. C ist still going strong due to its portability.
- The invention of object oriented programming brought us Java and C++.
- Now a lot of python code is bein developed.
- Today WWW-services are in fashion, the WWW carries php and Perl on its wires.

These programming languages are not only languages but they also enclose different ways of thinking about programming. Especially the step from procedure based programming (fortran, C) to object oriented programming is a difficult one. This can pose another challenge to software collaboration: to have to learn a new programming language or programming paradigm.

5 More Hurdles on the Path to Collaboration

People generally underestimate the amount of work required to implement a major data acquisition or data analysis package. Rarely the problem domain is fully understood on the outset of a new project. People thus are mislead that they can do a better job quickly.

Sometimes, lack of information about available software can stop collaboration on software already in its ealy stages. This problem should have been solved by the NOBUGS series of conferences.

The reliance of an available software package on a commercial foundation can also be a good reason to reject a package. The expenses for software licences are the least of the problems involved. Commercial packages become a problem when the vendor goes out of business or reimplements her software in an incompatible way. Who can gurantee a software vendor to exist for 20 odd years? No one.

Peoples egos are also a problem. There are just people who just think they can do better then others.

Another problem are manpower constraints. The main business purpose of the organisations the NOBUGS crowd is working for is to crank out knowledge about properties of matter. Software is only a means to an end. There will never be enough people around to attend to software how it should be. This is in the first place an incentive to collaboration. But colloboration also requires a lot of communication. And the amount of communication increases exponentially with the number of people involved. If there is not enough time for this, lack of manpower can become an obstacle on the path to collaboration.

Software collaboration also may cause legal problems, especially with the advent of software patents in recent legislation. Both software producers and adaptors can be at risk from rogue lawyers. Management may thus decide not to collaborate.

There are also problems of software craftmannship (But we are excused to be sloppy, see above). One major concern is the lack of documentation. Then there are portability problems: for each programming language there is a portable subset. This is the subset of the language which is implemented across a wide range of compilers. Portable programs have to use that subset. Many people are not aware of this and use non portable language feautures. C++ is a language especially prone to this problem. Often such problems cannot easily be resolved.

6 Invented Here is Good for You!

Usually people who fall victim to the Not Invented Here syndrome are frowned upon when they reinvent the wheel. May be this is wong, given the discussion above, there is a clear business case for rolling your own system:

- You get an up to date design
- You know the design
- You know the code structure
- You know the programming language
- You can fix bugs or extend your software quickly
- Nobody introduces incompatible changes or bugs
- You are not bothered with communication with codevelopers
- Your vendor does not go out of business

All this is important when a software has an estimated lifetime of 20 years or more.

7 How to Improve?

Given the discussion above, how can we make collaboration on software more likely? The author must admit that there seems to be no silver bullet. But there is room for improvement with the technologies of today.

Quite obviously, new software should be designed and documented with the aspect of changeability and portability in mind. This includes:

- Algorithms should be packaged as libraries with defined and documented interfaces. Thus they can be used as building blocks in other systems.
- At least provide documented hooks for those sub systems which are notorious for change:
  - Raw data input
  - Result output
  - Coordinate transformations
  - Raw data corrections
  - Hardware interfaces
- Modularize your code
- Document
  - The usage of the software
  - Describe the design ideas and algorithms which went into the software
  - Document the way the code has been modularized
  - Describe data structures
- Write readable code in the portable subset of a popular and well supported programming language. This implies: compilers or interpreters must be freely available.

8 Conclusion

It can be shown that there are actually good reasons not to collaborate which in the end means that a software system developed at another site is adopted. The main reasons for a reimplementation are the need to modify and maintain the software for 20+ years. For this an intimate knowledge of the software system is needed. Another reason is the low half–life of software system designs because of the fast paced developments in both software technology and methodolgy. However, as people regularly underestimate the amount of work required to build a new system, it often is wiser to adopt a system developed at another site and to face up to the learning curve.

Many of the issues raised in this paper may not be new; many might have a gut feeling about this. One aim of this paper is to pull the problems in collaborations on software into the open in order for them to be acknowledged and addressed.

9 References

The Mythical Man-Month Frederic Brooks, Addison–Wesley, 1995