Proceedings of the Open Source Software Development Workshop

Newcastle upon Tyne, U.K.
25th - 26th February, 2002

Photograph by David Greathead

Edited by
Cristina Gacek and *Budi Arief*
(University of Newcastle upon Tyne)

Message from the Editors

This is the second in a series of workshops to be supported by the Dependability Interdisciplinary Research Collaboration (DIRC). The focus of the Open Source Software Development workshop is on dependability and open source software development. Dependability is a deliberately broad term which, among others, covers reliability, security, safety and availability.

DIRC starts from the position that society's dependence on computer-based systems continues to increase and the systems themselves (embracing humans, computers and engineered systems) become ever more complex. Our interest is therefore in developing improved means of specifying, designing, assessing, deploying and maintaining complex computer-based systems in contexts where high dependability is crucial.

Within DIRC, the "open source approach" has been reviewed for its potential to contribute to aspects of dependability. One key observation is that there are many, quite different, characteristics of projects which are described as "Open Source". The open source approach is sometimes characterised as 'massively diverse human scrutiny': this both extends the idea of reviews or inspections and introduces a way of confirming final decisions about the inclusion of changes to a system. It poses interesting psychological, sociological and software engineering questions.

Examples of open source projects (e.g. operating systems, development tools, web and mail servers) indicate that a community can be built which can create software that is (claimed to be) highly dependable. It is not entirely clear what determines whether such a community can be built. Addressing such questions requires interdisciplinary research involving people from various backgrounds, including but not limited to sociology and computer science.

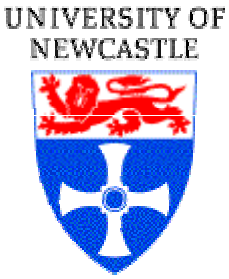
Our two keynote speakers add enormous value to the workshop. We have been fortunate enough to get Graham Button and Peter Neumann for this task. Graham Button is a sociologist, working for Xerox Research Centre Europe. He is well known for his work addressing Software Engineering and its development organisations. Peter Neumann is a computer scientist, working at SRI's Computer Science Lab. He has made a major contribution to the general problem of risks of computer systems. Of specific relevance to this workshop, Peter's more recent interest in Robust Open Source (RoS) has lead to a widely disseminated mailing list.

We have received a considerable number of very good contributions to this workshop. The main areas addressed include: understanding open source, trust and dependability, community, and software engineering and open source. Submissions came from a variety of sources in industry, government and academia, some being personally involved in open source software development, others in using such systems, and still others on doing research on the topic. This is coupled with an interesting diversity of disciplinary backgrounds of the contributors, originating from several continents (America North and South, Europe and Oceania).

This workshop could not have happened without the support and dedication of several people involved. We are very thankful to all of them, including:

- Joan Atkinson
- Denis Besnard
- Angela Birrell
- Diana Bosio
- David Greathead
- Cliff Jones
- Tony Lawrie
- Mark Rouncefield
- Carles Sala-Oliveras
- Claire Smith
- Tim Smith

We are grateful for the support of our sponsors, the Centre for Software Reliability and the Department of Computing Science at the University of Newcastle, and the DIRC project.



February 2002 Cristina Gacek and Budi Arief

Table of Contents

Software Engineering & Open Source

Organisational Considerations In The Work Of Software Engineering

Graham Button, *Xerox Research Centre Europe, Cambridge*

The movement towards 'open source' has mainly concerned what are relatively small undertakings. Yet, some of the problems of large-scale engineering projects done within complex organisational structures and involving scores and even hundreds of software engineers may also be addressed through the open source concept.

Within software engineering organisations, projects that are managed independently of one another are often addressing similar problems and might benefit from sharing code with each other, and past projects may have relevance for current projects. However, the organisational difficulties that often beset engineering projects may be a barrier to the idea of open source. For example, software development is done under strict deadlines and often falls behind schedule, any work not directly involved in moving the project forward is an unaffordable overhead. Also, the preservation of code from one project to another is often jeopardised through lack of documentation and the break up of teams. Further, developments are often done under a regime of SPI which requires code validation.

In this paper, we offer a sketch of some of the organisational issues that surround software and hardware development on middle to large scaled engineering projects. It is based upon ethnographic studies that we have carried out on four such projects. As sociologists, it is not our place to comment upon the appropriateness of open source itself, nor upon the way in which it could scale up to support the sorts of undertakings witnessed in our studies. However, with regard to the latter, we outline some of the organisational considerations that we have witnessed and the way in which they bear upon the actual doing of the engineering that may be relevant to a consideration of open source in these environments. Again, it is not our place to draw conclusions here, but to support others in doing that by providing evidence and data for their subsequent deliberations.

Developing Dependable Open-Source Systems: Principles for Composable Architectures (Carrying Goals to Newcastle?)

Peter G. Neumann
Principal Scientist
Computer Science Laboratory
SRI International
Menlo Park, CA 94025-3493
Neumann@CSL.sri.com
http://www.CSL.sri.com/neumann
Telephone 1-650-859-2375

Workshop on Open Source Software Development
University of Newcastle Upon Tyne
25-26 February 2002

Abstract

- -

The goal of this talk is to explore approaches for developing highly dependable systems and networks that can be readily composed out of subsystems, with predictable behavior. We reconsider classical development principles and their interactions with one another, and principled architectures that could be particularly attractive for developers of open-source software.

This talk is based on our project for Doug Maughan's DARPA's CHATS program (Composable High-Assurance Trustworthy Systems), under Contract N66001-01-C-8040.
Project Web site:

`http://www.CSL.sri.com/neumann/chats.html`

Newcastle, 25-26 Feb 2002

Trust and Trustworthiness

- We seek to develop predictably trustworthy systems and networks.
- Trustworthiness typically addresses security, reliability, high availability, real-time performance, and the ability to survive many realistic adversities.
- Assurance provides evidence of trustworthiness and justifies trust.
- In Europe, trustworthiness is called *dependability*.

- Trust is what you do when you don't know any better, or have no choice. Trusting something that is not trustworthy is foolhardy, but commonplace. We seek to architecturally minimize what must be trustworthy.

Newcastle, 25-26 Feb 2002

Reports on Our Project Web Site

- -

Interim reports:

- Composability Revisited
- CHATS Principles
- Principled Composable Architectures (draft in progress)

- Under a subcontract to the University of California at Berkeley, David Wagner is developing principle-based static-analysis tools using source-code model checking, with ensuing analyses of various targets within a range of modeled principles. Two papers are in progress, authored by Hao Chen, Dave Wagner, and Drew Dean.

Newcastle, 25-26 Feb 2002

Composability Revisited

- Obstacles to composability
- Attaining facile composability
- Paradigmatic mechanisms for enhancing trustworthiness
- Enhancing trustworthiness in real systems

http://www.csl.sri.com/neumann/chats1.html (also .ps, .pdf)

We address composability of policies, architectural subsystems and systems, software modules, protocols, static analysis techniques, etc.

Newcastle, 25-26 Feb 2002

Obstacles to Composability

- Properties outside of specification
- Properties of composition (emergent)
- Scalability issues
- Human issues

Newcastle, 25-26 Feb 2002

Attaining Composability

- Correctness/completeness of specs, code
- Statelessness, state visibility, OO
- Robust dependency strategies
- Robinson-Levitt interlayer consistency (*Communications of the ACM,* **April 1977**)
- Operating systems, programming languages
- Good architecture, software engineering, modularity, abstraction, types, etc.

Newcastle, 25-26 Feb 2002

CHATS Principles

- Principles
- Characteristic flaws and their avoidance
- Roles of formalism
- Caveats

http://www.csl.sri.com/neumann/chats2.html (also .ps, .pdf)

Newcastle, 25-26 Feb 2002

CHATS Principles

- Saltzer-Schroeder security principles: economy of mechanism, fail-safe defaults, complete mediation, separation of privilege, least privilege, least-common mechanism, psychological acceptability, work factor, tamperproof recording of compromises

Newcastle, 25-26 Feb 2002

More CHATS Principles

- Other architecture principles:
Sound architecture, minimization of what must be trustworthy, abstraction, encapsulation, layered/distributed protection, robust dependency, object orientation, separation of policy and mechanism, separation of roles, separation of domains, sound authentication, sound authorization and access control, administrative controllability, comprehensive accountability

Newcastle, 25-26 Feb 2002

Characteristic Flaws

- -

Some of the main types of flaws involve:

• Identification and authentication, authorization, initialization and allocation, finalization, run-time validation, consistent naming, encapsulation, asynchronous consistency, other types of logic errors

Newcastle, 25-26 Feb 2002

Roles of Formalism

- Improving requirements, specs, implementation consistency
- Identifying design and implementation flaws
- Increasing overall assurance
- Formalism is best used sparingly, where most effective.

Newcastle, 25-26 Feb 2002

Caveats on Applying Principles

- Principles are not absolute; they have numerous limitations.
- Principles may be in conflict with one another, and may not be composable.
- The principle of "simplicity" is misleading. Carefully structured architectures can mask complexity.
- Local optimization is dangerous.
- There is no substitute for human intelligence. experience, foresight.

Newcastle, 25-26 Feb 2002

Principled Composable Architectures

- Application of the principles
- Principled architectures
- Implementation and administration

The working draft is at
http://www.csl.sri.com/neumann/
private/chats3.html (also .ps, .pdf)
Feedback is of course welcome on this
and all other documents.

Newcastle, 25-26 Feb 2002

Architectural Concepts

- -

Centralized vs Decentralized

- Physical location
- Logical control
- Trust
- Trustworthiness

Newcastle, 25-26 Feb 2002

Useful Architectural Structures

- Layered trust & trustworthiness
  e.g, as in Multics
- Emphasis on trustworthy servers, as
  in Rushby-Randell's 1980s DSS,
  Proctor-Neumann ISPL architecture
  `http://www.csl.sri.com/neumann/ncs92.html`
- Networks as backplanes
  (but beware of networking risks)
- Trustworthiness enhancement
  `http://www.csl.sri.com/neumann/chats1.html`
- Trusted bootload, trusted paths
- Cryptographic authentication
- Finer-grained authorization
- Traceback abilities
- Trustworthy code distribution
- Alternative hardware bases?

Newcastle, 25-26 Feb 2002

Some Obvious But Nontrivial Conclusions?

- Principled requirements, architecture, design, implementation, and operational interfaces are all vitally important.
- The people involved therein are critical.
- Sensible composable policies, designs, subsystems architectures can lead to huge improvements in trustworthiness.
- When complexity is unavoidable, it can be managed with good architecture and principled software engineering practice.
- Better hardware may sometimes be needed.
- Support tools are helpful. Early analysis has enormous benefits, especially formal.
- Higher assurance can result as a byproduct of principled development.
- Trust only what is trustworthy.
- Foresight has enormous payoffs.

Newcastle, 25-26 Feb 2002

Relevance to Open-Source Systems

Our approach of architecturally principled development is particularly relevant to open-source software. It is ideally suited to collaborative developments. It is also applicable to proprietary systems, but is probably less likely to be applied, because of commitments to legacy compatibility, institutional reliance on minimizing time to market and cost cutting, obliviousness to critical requirements and the long-term value of disciplined software engineering, lack of education, training, and foresight, etc.

Open-source code is not intrinsically more trustworthy than proprietary code, but has great potential — with adequate discipline, wisdom, and attention to principled architecture and development. Symbiotic collaborative efforts can be very beneficial.

Newcastle, 25-26 Feb 2002

Trusted Open-Source Operating Systems Research and Development

Mr. Richard Murphy, Mitretek Systems
Dr. Douglas Maughan, DARPA

U.S. Department of Defense (DoD) computer systems and networks are constantly under attack. Such attacks make systems unusable, degrade performance, lead commanders to make poor decisions due to faulty data, leak valuable secrets, and leave behind code that could provide continuing back-door access or be activated on a predetermined event to take obstructive action. DoD systems are vulnerable due to increased interconnection and connection to the Internet. Additional vulnerabilities exist in common COTS products and can be exploited by anyone in the world to attack DoD systems. The DoD needs to develop focused technologies that support continued system operation in the presence of successful attacks, particularly addressing vulnerabilities and issues expected to arise in DoD's emerging network-centric warfare vision.

During the last several years there has been an increased movement towards the acceptance of community-produced, open-source software. This community-based development model has also found its way into the operating systems development community. DARPA is seeking to develop new security functionality for existing open source operating systems, leveraging the many years of operating systems development, and to demonstrate the value of useful security tools to the open source community. This will gain the cooperation and support of the open source community to move toward a higher assurance open source operating system framework, which can be embraced by the DoD and will be the foundation for future secure products and services.

The U.S. Defense Advanced Research Projects Agency's (DARPA) Composable High Assurance Trusted Systems (CHATS) program addresses this need by supporting development of high assurance open-source operating system technologies to protect DoD systems from constant attack. These technologies will be developed in concert with the unclassified Open Source research and development community to help foster the development of Open Source system technologies that will help to meet the DoD need for a high assurance operating system platform. Additionally, DARPA will engage the open-source community in a consortium-based approach to create a "neutral", secure operating system architecture framework. This security architecture framework will then be used to develop techniques for composing OS capabilities to support both servers and clients in the increasingly network-centric communications fabric of the DoD. These technologies are critical for defensive information warfare capabilities and are needed to ensure that DoD systems of the future are protected from imminent attack.

The CHATS program supports Open Source Operating System development in four broad areas: support for existing Open Source projects, such as file system development; support for new open-source initiatives such as the Linux Security Module program; high assurance cryptography support, such as device driver development; and assurance programs, such as policy modeling and system documentation.

The CHATS research projects are designed to improve the state-of-the-art of Open Source operating systems. There are projects designed to provide additional documentation of the internals of BSD-based operating systems. Another project provides a way for the Open Source community to rate the quality of submissions from software developers; this will help to improve the quality of Open Source systems. Other projects seek to improve the security of Open Source systems by providing support for different access control policies (such as role-

based, labeled security, and so forth), formal analysis of security policy enforcement, and by use of static code analysis tools to search for flaws.

There are projects that enhance existing file systems to improve their security and reliability, and to add additional cryptographic hardware support to Open Source systems. These enhancements include additions to existing Open Source systems to provide auditing support. These improvements are provided across several different operating system platforms in order to help raise the assurance of each system, thus, improving the overall quality of fielded systems.

This paper describes several projects that are part of the DARPA CHATS program, including descriptions of the goals of the projects and the definition of the framework that allows these projects to be combined to build an enhanced open-source operating system that will help to meet the needs of the U.S. DOD and the greater open-source community for systems that meet real security needs.

CHATS Program Areas

Support for existing Open Source projects

The CHATS program seeks to improve the assurance of Open Source systems by supporting several existing Open Source initiatives. These projects are working on enhancements to already existing technology that will provide critical support that other CHATS programs will rely upon as a foundation. The CHATS program seeks to influence the goals of these projects by encouraging them to consider enhanced security as an important design goal.

One of these existing projects is the Reiser4 file system for Linux. Reiser4 is a redesign of the existing ReiserFS file system. The ReiserFS is a journaling file system, based on balanced trees[1], that provides high performance and quick recovery as well as higher space efficiency. CHATS is sponsoring changes to ReiserFS to permit fine-grained security controls to be implemented in the file system as a set of plug-ins that can either be used to implement additional security policies (for example, encryption of file data) or to permit the storage of extended file system metadata (for storage of access control lists). Some of the other CHATS programs will rely upon such file system enhancements in order to implement increased file security.

Another CHATS program is the New York University/Massachusetts Institute of Technology Self-certifying File System (SFS). SFS will provide a secure means of distributing bulk data over a network[2]. SFS provides security by allowing a server to certify (by use of public key algorithms) the data that it supplies to a client. In addition to providing authentication of the source of distributed data, it also provides privacy by encrypting the data and integrity checking by the use of hash functions. SFS provides authentication of the server to the client (so that the client knows that the source of the data is authentic) and allows authentication of the client to the server (so the server can perform access control.) SFS also has a read-only variant that can be used for high performance distribution of such things as software installation kits.

[1] Namesys, *Three Reasons why ReiserFS is great for you*, 2001. http://www.namesys.com/.
[2] fs.net, *Self-Certifying File System: Frequently Asked Questions*, 2001. http://www.fs.net/sfs/

Support for new open-source initiatives

In addition to support for existing Open Source projects, the CHATS program has sponsored some new initiatives that are intended to improve the assurance of Open Source systems. Some of these will lead to new Open Source software, and others will support the Open Source development process. These projects will provide critical components for the CHATS architecture.

The CHATS program has sponsored the Security-Enhanced Bootstrap for Operating Systems (SEBOS) project at the University of Maryland to help build a high-assurance framework for initial operating system bootstrap. Working in conjunction with the LinuxBIOS project, SEBOS will provide a high-integrity environment for operating system bootstrap by using cryptographic integrity checking at several different points during the bootstrap process[3]. SEBOS intends to support cryptographic hardware in order to provide a tamper-resistant, authenticated bootstrap environment for Open Source systems. Support for a tamper-resistant bootstrap process is critical for deployment of a high assurance system. Without such support, there is ample opportunity for an attacker to interfere with the boot process of the operating system to manipulate it so that it does not properly enforce its security policy and can be used by the attacker in the future.

One argument for improvements to Open Source software is that wide availability of source code improves the quality of a software project. The argument is made that "many eyes make all bugs shallow"[4] – that is, with many reviewers, more bugs will be found and fixed. While this is generally correct, the CHATS program would like to see more secure software, not just improved software quality (less bugs). One of the new initiatives in the CHATS program is the Sardonix project at Wirex, Inc[5]. This project is attempting to improve the security of Open Source software by improving the tools being used to build Open Source software and by encouraging behavior that improves the security of Open Source software. The project incorporates three new initiatives – a security auditing portal for measuring the security "goodness" of programs, improvements in the C compiler, and a replacement for a frequently exploited network service daemon.

The Sardonix Security Auditing Portal will be a web site that allows participants in the Open Source community an opportunity to perform an audit of supplied source code. An auditor's identity is associated with the quality of the source that they have audited, allowing the auditor to be rated. This rating scheme will encourage the portal participants to compete with each other to find and fix security problems in the products that are submitted for review. Finding and fixing problems raises an auditor's "karma"; security holes found after an audit lowers the auditor's status. The use of the rating schemes is designed to encourage the community to perform audits and to encourage the development community to submit code for audit and corrections. The Sardonix system will record what software has been audited, when it was audited, and who performed the audit. Lessons learned from successful and failed audits can then be used to help grow the community of experienced code auditors.

Another way in which the Sardonix project hopes to improve the security of Open Source systems is by making improvements to the GNU C Compiler (GCC). The vast majority of Open Source software is built using this compiler – Linux and BSD systems both use the GNU compiler and libraries. The Wirex Immunix project, previously funded by DARPA,

---

[3] SEBOS, *Security Enhanced Bootloader for Operating Systems.* 2001. http://www.missl.cs.umd.edu/sebos/main.html.

[4] Eric S. Raymond, *The Cathedral and the Bazaar: Musings on Open Source by an Accidental Revolutionary*.O'Reilly & Associates, 1999. http://www.oreilly.com/catalog/cb/.

[5] Immunix, *Sardonix: Criticality for Critical* Systems, 2001. http://immunix.org/sardonix/

built a modification to GCC to make programs resistant to buffer overflow attacks. Since buffer overflows are the most common form of security vulnerability, using this GCC modification (called StackGuard) can eliminate several potential vulnerabilities. While StackGuard continues to be maintained by the authors, users must obtain the StackGuard patches for their system then rebuild both the compiler and operating system utilities in order to be protected. Most deployed open-source systems are built from precompiled "distributions" for the sake of convenience; those pre-built systems can't benefit from the StackGuard patches. Sardonix proposes to fix this by working with the GCC developers to move the StackGuard changes into the main line of GCC development, so that any newly built distribution can benefit from the StackGuard protections.

Another project at Wirex is the Linux Security Module project[6], which is seeking to build a framework for extensible security policy enforcement into the Linux kernel. This work is a compromise solution for a long-standing conflict within the Linux community. Given the popularity of the Linux kernel, there is a broad constituency that wants to influence the kernel to meet their particular special interest. Often the various special interest groups are at odds with each other – one group asking for changes that conflict with the needs and desires of another group. Two such interests are the secure system community and the embedded systems community. While the secure systems interests would like to see Linux grow into a flexible, high assurance operating system, the embedded systems special interest groups would like to see all security enforcement removed from the kernel in order to eliminate the perceived overhead. Linus has been receptive to security enhancements in the past and has permitted kernel changes to facilitate them. However, several different security projects have expressed interest in using Linux as a delivery mechanism. This has led to multiple competing demands ranging from completely removing security enforcement (as desired by the embedded systems people) to those who would like multiple enforcement mechanisms to be included with the kernel.

At the Linux 2.5 Kernel Summit, Linus Torvalds proposed a solution for these competing demands [7]. He proposed adding "hooks" to security enforcement functions in the kernel; software that is interested in security enforcement could then utilize those hooks to influence kernel security decisions[8]. The enforcement functions would be separated from the base kernel and would not need to be maintained by the kernel development team. This separation solves many of the problems noted before – the additional enforcement software is not part of the base kernel, thus it does not add to the kernel maintenance burden. The kernel builder can choose whether or not to include any additional policy enforcement software, which allows embedded systems to eliminate the additional overhead. (As will be pointed out later, there is some overhead added, but the design attempts to minimize this.)

The design of the kernel interface uses the existing Linux loadable module interface. The loadable module interface allows a self-contained external module to be merged into the kernel from an external module, or to be statically built into the kernel binary. The LSM modifications allow the policy enforcement software to be built as a loadable module that can be either installed at run-time or built into the kernel. This paper describes a project that is implementing such an enforcement module interface for inclusion into the Linux kernel and an extension that uses that interface

---

[6] Crispin Cowan, *Linux Security Module Interface*, 2001. http://mail.wirex.com/pipermail/immunix-users/2001-April/000063.html

[7] Smalley, Stephen, Timothy Fraser, Chris Vance, *Linux Security Modules: General Security Hooks for Linux*, 2001. part of the Linux Security Module distribution.

[8] Loscocco, Peter, *Be Careful, Please*, 2001. Message on the Linux Security Module mailing list, available from http://mail.wirex.com/pipermail/linux-security-module/2001-April/000084.html.

The Linux Security Module (LSM) work is a joint effort between Wirex, NAI Labs, and others in the open-source community that is working together to build the framework for customizable security enforcement for Linux 2.4. The LSM changes should be released as part of the Linux 2.5 kernel distribution. Several projects plan to take advantage of the LSM work – Immunix, SELinux, SGI IRIX, and Janus[9]. The purpose of the project is to provide a general framework in the kernel for supporting arbitrary access control modules. By itself, the work does not provide any additional enforcement – it only provides the "hooks" that allow an enforcement module to make access control decisions. The "hooks" insert function calls at critical points in the kernel permission logic. The process of installing a security module arranges things so that the kernel calls that module to assist in a policy decision. If no module has been installed, the kernel calls a dummy module that does nothing – while this installation adds some overhead (an additional function call to a function that immediately returns success), this trade off is acceptable. In fact, the LSM patch removes some of the existing policy code (the implementation of the POSIX capabilities logic), so there is a potential for a net performance gain when no security module is installed. Preliminary measurements made by the LSM team are encouraging; it appears that the LSM patch does not impact kernel performance.

Many U.S. DOD programs require the use of systems that meet the Common Criteria (CC) evaluation requirements. Specifically, there is a set of requirements for general-purpose operating systems called the Controlled Access Protection Profile (CAPP). The CAPP requirements are essentially the old Orange Book C2 requirements recast into Common Criteria form. This requirement is significant as many government regulations require the use of evaluated systems (i.e. ones that meet the CAPP requirements). Unfortunately, none of the Open Source operating systems are currently able to meet the CAPP requirements. While this is partially due to the lack of complete OS internals documentation, there are also functional requirements that are not met. One significant functional problem with Open Source operating systems is the lack of secure auditing capabilities. The CAPP requires the system to have a Trusted Computing Base (TCB) that provides the ability to create, maintain, and protect from modification or unauthorized access or destruction, an audit trail of accesses to the objects it protects. One of the CHATS participant projects (at the SPAWAR Systems Center in San Diego) has a goal to develop a kernel-level auditing package for Red Hat Linux that is compliant with the CAPP requirements for security[10]. It will also provide key features needed for the Defense Information Infrastructure Common Operating Environment (DIICOE) certification. This auditing package will allow other CHATS components to meet their information assurance goals, and it will be of immediate benefit to law enforcement in computer forensics. Law enforcement relies on audit and transaction logs when investigating computer crime. Current Open Source operating systems either lack auditing entirely, or log using mechanisms that cannot be demonstrated to be tamperproof. By residing at the kernel level, this auditing package will allow all processes to be monitored and logged in a way that can be demonstrated to meet the assurance requirements of law enforcement. This project will develop a system of kernel processes and system administrator utilities, which meet the CC audit requirements. The implementation will provide assurance that the logs have not been tampered with so that they can be admissible as evidence in a court of law. A final aspect of this project will be a text reader for audit data that can support law enforcement in forensics analysis work.

---

[9] Cowan, Crispin, *Linux Security Module Interface*, e-mail announcement sent to the Bugtraq mailing list, April 2001.

[10] SPAWAR Systems Center, *Secure Auditing for Linux*, 2001. http://secureaudit.sourceforge.net

The Monterey Security Enhanced Architecture (MYSEA) project at the Naval Postgraduate School[11] will construct a prototype demonstration of an open source high assurance distributed operating environment for enforcing multi-domain security policies while supporting popular office productivity applications without modification. MYSEA will provide convenient use of popular computing environments and applications while securely accessing multiple domains of information, and the cost savings resulting from the continued use of legacy, proprietary, commercial clients (e.g., WinNT).

MYSEA will construct an innovative high assurance multi-domain distributed architecture that integrates multi-domain support into the OpenBSD operating system. It will provide local and remote trusted path services, single sign-on for access to multiple trusted servers, and integration of security policy management with internal security services. This project will result in new and improved security functionality for existing open source operating systems, and will contribute significantly to the ability of distributed open source components to interoperate securely.

The MYSEA project will implement file security attribute extensions for OpenBSD to support equivalence-class domain assignments for both objects and active subjects. A rule manager will be provided that can support a wide range of policies with respect to these assignments. MYSEA will provide mechanisms to permit remote and local access to multi-domain information at multiple session levels. Policy-aware protocol servers will provide an environment that allows the use of a trusted path (a secured, authenticated connection for security attribute negotiation) between networked systems. The system will provide for single sign-on to the security domain.

High-assurance cryptography support

One of the widely recognized successes in the Open Source operating system arena is the security advances that have been realized by the OpenBSD operating system. OpenBSD has taken an initiative to build a highly secure operating system by performing multiple source code audits of all components of the system. This audit work has led to the elimination of several potentially exploitable bugs in the OpenBSD operating system. The Portable Open Source Security Elements (POSSE) project, managed by the University of Pennsylvania[12], seeks to build upon this work by performing a similar audit of the widely used OpenSSL (Secure Socket Layer) software. As part of this audit, the POSSE team will work on support for higher assurance cryptographic hardware and software for use with BSD-based operating systems. In addition, The POSSE project will work with the core teams from the OpenBSD, OpenSSH, and OpenSSL development groups to try to build a community of developers to help improve the security of OpenSSL, OpenBSD, and other BSD-based operating systems. They will initially work by organizing audits of these systems, delivering an audited version of OpenSSH, OpenSSL, and OpenBSD. The widely used OpenSSH and OpenSSL will be portable immediately to a much wider Open Source system base than just OpenBSD. The POSSE team will work with other system developers to insure that they are aware of any bugs found during the OpenBSD audit so they can fix corresponding systems. Finally, POSSE will work with the development community to help raise the security awareness of the developers so that security bugs are not introduced in the first place.

The BBN Technologies High Assurance Open Source Security Certificate Management System (CMS) will provide a set of components that can be used to implement a complete

---

[11] Naval Postgraduate School, *MYSEA – Monterey Security Enhanced Architecture*, 2002. http://cisr.nps.navy.mil/pages/research/mysea/.

[12] The POSSE Project, *The POSSE Project*, 2001. http://www.cis.upenn.edu/~posse.

Public Key Infrastructure (PKI) certificate management system[13]. The CMS will be built from an existing server that supports X.509 certificate management by adding smart card support and by porting the software to an Open Source operating system (Linux). The resulting CMS will provide a Certificate Server System that can support multiple Certificate Authorities (CAs) and a Registration Agent (RA) that accepts and processes certificate requests. The CMS relies upon the services of a trusted operating system such as those built using the DARPA-sponsored Linux Security Module (LSM).

Assurance Programs

A key requirement for building high assurance software is the use of development methodologies that lead to disciplined software development. While there is no shortage of talented developers willing to work on projects, there has not been a way for new developers to "learn the ropes". This leads to the same mistakes being made over and over during the development of software projects. Also, there is an experienced computer security research community that could be an invaluable resource to the Open Source development community if there was a way to foster interaction between the groups.

The Community-Based Open Source Security (CBOSS)[14] effort will use the resources of the security research community to develop and deploy a set of generalized security architectures that are derived from current research results. These architectures will be developed in conjunction with senior Open Source developers in order to foster a community that is committed to making meaningful and lasting changes in the way the Open Source community develops software. This community will be built with the participation of several of the Open Source community's most influential members. By influencing these senior developers, CBOSS hopes to build an environment where new developers are given the knowledge, tools, and techniques to help them develop software that is sufficiently secure to support the DOD mission.

This approach recognizes that past initiatives to improve the security of Open Source systems have failed because they have been one-time attempts at fixing the problem. While these short-lived attempts have been helpful in fixing individual problems, they have not addressed the root cause of the problem – developers that don't know how to avoid common coding errors that lead to security problems. Each new developer is thus left to learn by making the same mistakes. The CBOSS project proposes to build this community-based approach to help improve the community's development practices by initiatives that will support the transfer of existing security knowledge and technology between the research community and the Open Source developers. CBOSS will also foster technology for implementing kernel security extensions as well as building new technology for high-security applications.

The CBOSS project does not claim that these initiatives will eliminate all the causes of security problems in Open Source software. What CBOSS is intended to do is to eliminate the most common and easiest to correct problems by a combination of education and cooperation. The community fostered by this work will lay a foundation for longer-term research on more comprehensive changes over time. Since the CBOSS project has enlisted the help of many well-known members of the Open Source community, the wider Open Source development community should accept the effort. Through this community-based approach, the CBOSS effort seeks to prevent security problems by improved education of developers, improve the quality of existing code, and foster the development of systems that can be trusted for mission-critical applications.

---

[13] BBN, *Certificate Management System*, 2001. http://www.bbn.com/infosec/cms.html
[14] NAI Labs, *Community-Based Open Source Security(CBOSS)*, 2001. http://opensource.nailabs.com/initiatives/cboss/index.html

Another approach that the CHATS program will use to improve the quality of new and existing Open Source programs is the Code Security Analysis Kit (CoSAK) project[15]. This project will build a tool framework and define a methodology to help Open Source developers produce code that is more robust and secure. The CoSAK tools will provide support for auditing of software by performing an automated analysis of the code in order to gain an understanding of the structure of a program so that routine "contracts" can be derived. A contract is a specification of the invariants that define proper operation for a routine or module; for example, the range of valid values for local variables, requirements on procedure inputs and outputs, and other environmental details such as external variable values. The CoSAK tools then assist in the enforcement of the contract terms. CoSAK will concentrate on externally visible interfaces, as those are the places most prone to exploitation. For each software system to be audited, tools are used to assist in the understanding of the program code. The result of these analysis tools is a set of contract formulations that consist of the conditions that must apply during the execution of the code (timing constraints, utilization of certain system calls, size of the program stack frame, etc.). The CoSAK toolkit includes a run-time environment that is responsible for enforcing the constraints imposed by those contracts. This run-time environment is responsible for taking appropriate action if any of the contract constraints are violated.

Another effort for making software more secure and robust is the Static Security Analysis for Open Source Software project at Secure Software Solutions[16]. This project will build on past experience with automated software analysis tools to build a set of automated program analysis tools that can be used to search project source code for known security flaws. These tools are intended to assist in the audit of Open Source project source code by helping to quickly identify potential flaws so the auditor's time can be more effectively used. The analysis tools, which will be targeted toward C programs, will use a constraint-based static analysis approach in combination with heuristic approaches to provide a highly automated audit of source code. The Sardonix and CBOSS communities will be using these tools during their auditing efforts.

Architectural guidance for the CHATS project comes from the Architectural Frameworks for Composable Survivability and Security project at SRI[17]. The main goal of the work is to provide sound architectural frameworks for composable high-assurance trustworthy distributed systems and networks, explicitly stimulating the development of robust open-source operating systems for applications with critical requirements. The project will research the state of distributed and network system architectures with a view toward building an architecture with high survivability and security, interoperability, composability, and evolvability, with potentials for high assurance, while exploiting the open-source paradigm. The project will provide guidance on how to build trustworthy systems from less trustworthy components (composability), articulate important design principals, and provide an architectural framework for secure, survivable systems. Additional work from this project will be to provide consulting to other CHATS projects and to build tools for static analysis and diagnosis of some types of security flaws.

Another way that CHATS seeks to improve the security of Open Source operating systems is by the use of formal verification. A formal analysis can establish that a particular system

---

[15] Drexel University SERG, *Software Engineering Research Group*, 2001. http://serg.mcs.drexel.edu/cosak.html

[16] Secure Software, *Secure Software Solutions – Projects*, 2001. http://www.securesw.com/Projects/CHATS'.

[17] Peter Neumann, *Architectural Frameworks for Composable Survivability and Security*, 2001. http://www.csl.sri.com/users/neumann/chats.html

enforces a set of security goals, but can be costly, time-consuming, and risky. The Analyzing Security Policies for Security Enhanced Linux (SELinux) project[18] will try to dramatically reduce the level of effort necessary to formally verify that the SELinux system properly enforces the defined security policy. The project will design methods and tools that can be used to simplify the task of verifying complex security policies as defined using the SELinux policy language. These tools are designed for use by the Open Source community to verify that their policy definitions meet their objectives. The Security Policies for SELinux project will begin with a small subset of the SELinux subjects, objects, and types and model the behavior of the system. Initially, the project will use existing analysis tools to model this minimal subset.

The goal of the verification work is to define a methodology that allows the verification tools to be used by a wider audience – specifically designers without past experience in formal model verification tools. The project hopes to improve the assurance of Open Source systems by permitting more system designers to formally verify their designs and models. The approach to be used to permit this is to build an easily understood verification methodology that allows expression of the designer's security goals into a form that can be transformed into input for existing formal verification tools. While this work will begin with the minimal subset policy, the intention is to extend the coverage to incorporate a wider range of security policy goals.

CHATS Architecture

The mission of the CHATS program is to "focus on the development of the tools and technology that enable the core systems and network services to protect themselves from the introduction and execution of malicious code and other attack techniques and methods. These tools and technologies will provide the high assurance trusted operating systems the security services needed to achieve comprehensive secure highly distributed mission critical information systems for the DoD. This program will fundamentally change the existing approach to development and acquisition of high assurance trusted operating systems technology by advancing the security functionality, security services, and the state of assurance in current open-source operating systems and developing a long-term architectural framework for future trusted operating systems."[19] The CHATS projects are designed to support this mission by building a bottom-up set of tools and practices that meet the CHATS goals.

At the bottom layer, there is a foundation of high assurance programs – projects that contribute to the ability of the upper layers to depend upon the foundation to ensure that systems work as they are expected to work. These architectural foundations include the frameworks derived from the Architectural Frameworks task and the formal policy verification work on the SELinux kernel.

Building upon this foundation are the security architectural documentation from the CBOSS and POSSE projects and security analysis tools from the Static Security Analysis project and CoSAK project. These provide assurance that software is using interfaces correctly, operates as intended, and is free of obvious flaws. The Security-Enhanced Bootstrap from the SEBOS

[18] Naval Research Lab, *Analyzing Security Policies for SELinux*, 2001. http://chacs.nrl.navy.mil/projects/selinux/

[19] DARPA Information Technology Office, *Composable High Assurance Trusted Systems*, 2001. http://www.darpa.mil/ito/research/chats/

project provides additional assurance that the operating system can be trusted to enforce the desired security policy.

This architectural foundation provides a basis for the other CHATS projects in the form of an improved, more trustworthy operating system base. Some of the other projects are explicit in their need for this trustworthy base – for example, the CHATS Certificate Management System (CMS) depends upon the trust in the kernel gained by architectural work such as the security policy analysis work. Other cryptographic projects rely upon this architectural layer for proper operation. The POSSE work on OpenSSL and on crypto devices as well as the CMS project both rely on the architectural underpinning. They also provide services to upper layers of the CHATS architecture.

These foundation components provide a basis for additional trustable operating system components. Examples of these components are the ReiserFS and Self-Certifying File System, which provide the ability to provide more expressive local and remote file system access policies.

At the top layer, the CHATS program has projects that are designed to encourage the Open Source community to consider security during the development of their programs. These projects are the Sardonix auditing portal, the CBOSS community, and the source code auditing tools (CoSAK and the static security analysis projects.) Programs like Sardonix and CBOSS are intended to reward developers that build secure software in the hopes that these rewards will provide the incentives that result in more secure software being built.

Conclusion

The wide acceptance of Open Source systems as an alternative to proprietary systems has provided an opportunity for the security research community to provide a positive influence on the development of operating systems. The CHATS program is taking advantage of this opportunity to nurture relationships between security researchers and Open Source developers in order to make security an important consideration for Open Source systems. We are working to ensure that these relationships cover a broad range of operating systems; indeed, we hope to assist in the migration of effective security mechanisms between operating systems. We believe that the CHATS programs will help to foster high impact, short term improvements in security while encouraging the broader community to think about security while building their systems, which will lead to a self-sustaining long-term improvement in assurance over a broad range of Open Source systems.

Advantages of open source processes for reliability: clarifying the issues

D. Bosio, B. Littlewood, L.Strigini
Centre for Software Reliability

M. J. Newby
Department of Actuarial Science and Statistics
City University, London, England

Abstract

Some authors maintain that open source software processes are particularly well-suited for delivering good reliability. We discuss this kind of statement, first clarifying the different measures of reliability and of a process's ability to deliver it that can be of interest, and then proposing a way of addressing part of it via probabilistic modelling. We present a model of the reliability improvement process that results from the use of the software and the fixing of reported faults, which takes account of the effect on this process of the variety of software use patterns within the user community. We show preliminary, interesting, non intuitive results concerning the conjecture that a more diverse population of users engaged in reporting faults may give OSS processes an advantage over conventional industrial processes, in terms of fast reliability growth after release, and discuss further possible developments.

1 Introduction

Many claims have been made about the dependability of Open Source Software (OSS), some of them contradicting each other (OSS is generally better than Closed Source Software (CSS), or vice-versa), some of them presenting a challenge to intuition (OSS is more secure because of the accessibility of its source code to all, including would-be intruders). We find that these arguments often fail to clarify the claims made and the reasoning supporting them.

We propose to add some clarity to some of the issues. Specifically, we wish: to separate different qualities of possible interest, corresponding to

Workshop on Open Source Software Development, Newcastle upon Tyne, 25-26 February 2002

different precisely defined measures related to reliability; then to give formal expression to some of the conjectured laws which support the claims made; to discuss to which extent these conjectures are consistent with the common understanding of the processes that produce software reliability. We do so via an example of probabilistic models of the effects on software dependability of factors in the software production process. In this paper, we use, as examples, models of a class which we developed earlier to weigh claims about the merits of different testing methods [1, 2, 3].

1.1 Different aspects of reliability

One can identify many attributes of a product that are components of its *dependability* (an umbrella word we will use for all the meanings of "reliability" in its common, non-mathematical sense, to avoid confusion with the specialist term "reliability").

For instance, we can discriminate between the ability of a system to deliver continuous correct service, described by "reliability" in its strict technical sense, from its ability to provide a correct service at any given moment. E.g., the probability of surviving a mission is a reliability measure while the average uptime is an availability measure. This distinction is important because the two measures describe requirements whose relative importance varies between users and circumstances; because a system can exhibit a high level of availability without a high level of reliability; and because the best system design for a given application (e.g. using static redundancy vs rollback recovery) varies depending on which requirement is most important.

Other important distinctions centre on the characteristics of the failures of interest, characterised by severity along a unidimensional scale, or in terms of the level of specification that they violate (specifications internal to the development process vs expectations of the users), or in terms of differences of kind: e.g. related to maintaining data integrity or privacy against intentional attacks (two attributes of *security*) or to avoiding failures that cause outcomes classified as hazards or as accidents (attributes of *safety*). It is obvious that these distinctions also matter to a debate about OSS processes, as exemplified by the debate over achieving security via openness vs via obscurity of code or algorithms.

In this paper we choose to talk, for ease of exposition, about a single reliability attribute, measured by the probability of a system completing a demand (e.g. a user session for interactive software) without failures.

We also wish to point out two further important distinctions:

- reliability of a specific release of a product vs the evolution of reliability during the lifetime of the product. One can have software that is highly reliable at release but improves slowly (as an extreme example,

many safety-critical products will exhibit few failures but be subject to exceedingly time-consuming procedures for any update), just as software that is initially unreliable but improves quickly. From the users' viewpoint, in many critical applications it is necessary to know that software is sufficiently dependable when first deployed; in other applications, teething problems are tolerable, especially if the software can be expected to improve rapidly, or to be exempt from reliability deterioration in the long run. Some of the alleged advantages of open processes, e.g. prompt response to problem reports, seem more plausibly to aid fast reliability improvement than reliability at release time;

- average reliability over all users, vs reliability seen by individual users or groups of users. The manner of use of a product determines its reliability; so, the failures affecting different users differ in kind and frequency. Even software that performs well for most users may be ruinously unreliable for some of them. These unfortunate users are hidden in statistical averages, but the risk of becoming one of them should weigh heavily on the mind of anyone planning a software purchase. A software vendor is often interested in the average reliability among all users (and possibly the number of users so wretched that they might endanger the reputation of the vendor). A user is interested in reliability for him/herself. For instance, this concern contributes to the insistence of some military customers on being able to take over maintenance of the products they buy.

  Again, we see a plausible conjecture that some aspects of open processes improve the lot of the least favoured users, e.g. the possibility for minorities of users to modify the source code to fix their own specific problems, possibly leading to great advantages, from a buyer's viewpoint, even if the average reliability to be expected were less than that obtainable from a competing, less open process.

Interestingly, this last conjecture is related to an assumption that seems to underlie many of the claims for the usefulness of "open" processes: their ability to exploit diversity among developers and among maintainers. For instance, the saw "given enough eyes, all bugs are shallow" is obviously wrong if all the eyes have blind spots for the same bugs. What matters is that some eyes naturally see bugs that are hidden to other eyes. Diversity between eyes is a common principle in all development processes: from the use of independent V-and-V staff to "dual programming". The issue is which forms and extents of diversity work better, from the various viewpoints of interest. We will outline some useful research questions and conjectures for shedding light on possible advantages and disadvantages of open processes in this area.

1.2 Probabilistic modelling

We wish to shed some light on the possible contributing factors to the claimed greater (or lower) reliability of OSS. The questions of practical interest we want to answer are of the form "Does factor X in the development process tend to improve dependability measure Y"?

We choose here one aspect of statements like "the greater diversity between participants enjoyed by OSS processes causes better reliability in OSS products", often used either to argue that OSS processes favour dependability or to explain the good dependability observed in some products of OSS processes.

The first advantage of mathematical modelling is that it forces us to explain what we mean by "diversity" and "greater degree of diversity", to express formally which results we wish to compare, etc. After being so specific, we can often check whether it is plausible that the factor alleged to improve dependability actually improves it, and under which additional conditions we should observe this effect.

Some use of modelling is usually necessary for supporting a claimed causal effect between aspects of the software production process and its achieved results, even given some empirical evidence. What we do here is to make the modelling formal and explicit. The only alternative would be appealing to bare statistical evidence of correlation between the two. This could prove to be prohibitively difficult. Checking empirically even a simple statement like "OSS products are more reliable than the others, all things being equal" is difficult in practice, for various reasons: paucity of products with documented reliability, difficulty of choosing "equal" terms of comparison, expected high variability of the effects so that it may be exceedingly difficult to reach conclusions with any level of confidence.

We have started applying this approach to a specific scenario – we model the reliability growth of software while in use after release – and initially to a single conjecture of interest: that the diversity of users involved in fault-reporting in an OSS environment may give it an advantage, from this viewpoint, over comparable software that enjoys less of such diversity.

We do *not* inquire whether OSS products do improve faster than comparable non-OSS products (a worthwhile investigation if the investigator overcomes the difficulties cited above), nor whether OSS products do in general enjoy more diverse fault reporting. Instead we study how this kind of diversity would affect software reliability growth if the plausible assumptions of our model were true. Essentially, we wish to produce conjectural laws that link this kind of user diversity to reliability growth.

These laws are of interest to decision-makers, e.g. project managers who can influence the make-up of the user community engaged in failure reporting, or procurement managers who have to choose between products with visibly different make-ups of this community. Using a mathematical

model also clarifies which statistical evidence would support or refute the idea that these laws are at work in the real world, indicate confusing factors that may affect the measurements, and so on. Even without experimental support, a model that decision makers can recognise as consistent with their experience will allow them to scrutinise the less formal, intuitively appealing arguments proposed to them.

Our modelling is thus not framed in terms of OSS vs non-OSS processes: it applies to comparing any processes whose differences can be described in terms of its parameters.

2 A model of reliability growth during use

2.1 Description and basic assumptions

We start with an intuitive description of the process of finding faults in software while using it, and of removing them. A program has well-identifiable defects ("bugs", "faults"), which may cause it to fail. There is a set of users using the software. Some of them may actually be intentionally testing it, some just using it normally. We do not need at this stage to discriminate between the two sets. What matters is that they use the software, and they may experience failures, as a random process due to their (different) sequences of use of the software. If a failure occurs, the user may notice it and report it. If it is reported, someone may attempt to identify and remove the fault that caused it (and the attempt may succeed or fail).

We wish to describe the reliability growth patterns taking place as a result of all these factors, from the viewpoint of any one user or group of users.

We now go into more formal details of this model, as previously described in [1, 2, 3].

For simplicity, we restrict ourselves to a demand-based model of program execution. A program is given a demand, computes a result and terminates. In other words, we characterise the "extent of exposure" of the program to failure as a discrete variable represented by the number T of executions of (i.e., demands applied to) the software. A demand is characterised by the values of all the input parameters and machine state that determine the behaviour of the program in one execution. This model is very general, applying, for instance, even to interactive programs if we consider a "demand" and a "result" to include the sequences of all user inputs and of all the program's outputs during a session (cf [2, 4]).

The collection of all the possible demands is called the *demand space*. The demands which will cause the program to fail (*failure points)* form its *failure set*, which we describe as composed of multiple, non-overlapping *failure regions*, each a collection of failure points corresponding to a specific defect (or *fault*) in the code. If a failure point is found (through observing

a failure), and if an attempt is made to remove the fault that caused it, then either the failure region to which it belongs is completely eliminated (successful fix) or not at all (this is a simplistic description, commonly accepted in the literature; in reality, even the definition of what is "a fault" is ambiguous, as two different people trying to eliminate the cause of the same failure may well change different parts of the code; cf the discussion in section 2.2, *"So-Called Faults"*, of [2]).

Users use the software in many different ways. The users' ways of using the software are described by their *usage profiles*. A user's usage profile is the set of the probabilities of each possible demand being chosen by that user. It determines the different probabilities of the program failing[1] for that user due to each bug in the software (in a given number of demands by that user), and thus also the reliability of the software for that user.

We also assume that the demands chosen on different executions (by the same or different users) are statistically independent. Making this assumption realistic requires a judicious choice of what is defined as "one demand", and excludes some kinds of testing from our model. These assumptions can model "operational", "statistical" or "stress" testing – each regime is modelled by different usage profiles and different rates of bug reporting – but not partition testing. We will not be modelling the reliability growth that arises from early stages of testing by developers, which is of little present interest to us.

Different users may use the software more or less frequently. Individual users also differ in their probabilities of reporting the failures they observe.

2.2 Parameters of the model

The parameters in the model are

- $q_{i,j}$, the probability of the fault i causing a failure for the user j on a randomly selected demand,
- $r_{i,j}$, the conditional probability of a failure caused by the fault i being reported by user j if it occurs in an execution for that user,
- f_i, the conditional probability of the fault i being successfully fixed given it has been reported,
- T_j, the number of demands applied by user j by the moment in time at which we study the achieved reliability of the software, and total number of demands by all users $T = \sum_j T_j$.

[1] Note that we do not consider the consequence of a failure: the severity of a bug is only given in terms of the probability of selection in operation of a demand from the failure region associated to it.

All the probabilities just described, $q_{i,j}, r_{i,j}$ and f_i, are considered constant over successive demands[2]. In other words, we are assuming that the number of users, and their behaviour in terms of software execution and bug reporting, are constant over time. This constrains the generality of the model, but not as severely as it may appear. For instance, we can describe users recruited later as users who were always present but perform no executions until a certain time. We can also describe a step change in a user's usage profile in terms of two virtual users, one of which stops executing the software when the other starts.

The $q_{i,j}$ parameters are determined by each user's usage profile, and in turn they determine both the effects of each fault on the reliability experienced by that specific user, and the probability that that user will be able to report the fault. For a given fault i, larger $q_{i,j}$s correspond to users with higher probabilities of being affected by failures caused by fault i, thus users who are better at finding that fault: if these users are intentionally testing the software, they are better testers, and if they are using the software, they are the less fortunate users.

The model parameters describe measures of practical interest in a debate about software processes. About OSS processes, it is often stated that:

- their products tend to have better (or worse: opinions differ) initial quality than with alternative processes. This can be modelled by sets of $q_{i,j}$ parameters giving lower (conversely higher) $\sum_i q_{i,j}$s for the users whose experienced reliability we consider.
- they enjoy better failure reporting: higher values of [some] $r_{i,j}s$.
- they offer more responsive bug-fixing for at least some faults: higher f_is.

2.3 Reliability improvement as a result of use

This model represents the fact that the reliability improvement process is a stochastic process. The fixing of faults depends on when (and whether) they are found during execution, and their being reported, and the report prompting an action to fix the bug, and the fix actually removing the fault. The model describes statistically how this process will evolve.

For instance, the initial reliability of the program as seen by user j corresponds to $T = 0$, i.e. after no executions of the software, and is described by its probability of failure on demand (pfd) pfd_j

$$pfd_j = \sum_{i \in \{failure\ regions\}} q_{i,j} \ . \tag{1}$$

[2]Note the underlying simplifying assumption that the probability of a user reporting a fault does not depend on how many times that user has observed failures caused by that fault before.

We now consider the case where multiple users have executed the software, each user k having executed T_k demands. The probability of a fault having being removed is the probability of the fault having been reported (at least once) and fixed. The probability of the fault being reported at least once is $1 - P(\text{fault } i \text{ not reported})$. The probability of the fault not being reported by any user is given by

$$P(\text{fault } i \text{ not reported}) = \prod_{k \in \{users\}} (1 - r_{i,k} q_{i,k})^{T_k} \ , \tag{2}$$

Recalling that f_i is the probability of fixing bug i once it has been reported, we have that the probability of the fault i being removed is

$$P(\text{fault } i \text{ removed}) = f_i(1 - P(\text{fault } i \text{ not reported})) \ .$$

Every user will experience an improvement in reliability as a result of the faults fixed when revealed in this multi-user "testing" activity, but this improvement will be different for different users (as, indeed, would be their initial perceived reliabilities before testing). The mean reliability as seen by user j as a result of the multi-user testing, after a total number of executions $T = \sum_k T_k$, depends on the usage profiles of the other users, in the following way:

$$pfd_j = \sum_{i \in \{failure\ regions\}} q_{i,j} \left(1 - f_i (1 - \prod_{k \in \{users\}} (1 - r_{i,k} q_{i,k})^{T_k}) \right) \ . \tag{3}$$

Thus, the associated expected increase in reliability for user j as a result of exposure to the T tests will be

$$I_j = \sum_{i \in \{failure\ regions\}} q_{i,j} f_i (1 - \prod_{k \in \{users\}} (1 - r_{i,k} q_{i,k})^{T_k})) \ . \tag{4}$$

All the following factors will decrease pfd_j (and hence increase I_j, the increase in reliability of the software for user j): adding more users, performing more tests, increasing the value of any of the $r_{i,k}$ or $q_{i,k}$, or f_i parameters. In other words, the more users executing (and thus testing) the software, the greater the benefit for the community; the more likely each person is to report any bug, the more bugs will be exposed, allowing for more bugs to be corrected. The more likely the bug is to be removed if reported, the higher the resulting reliability of the software.

We note that users for whom $r_{i,j} = 0$ for all i are "free riders": they experience reliability improvement without reporting any bugs, and they benefit from bugs being reported by and fixed for others.

All these are somewhat unsurprising properties, simply confirming that the model captures "common-sense" understanding of how people interact

in fixing bugs. It is interesting to see how competing claims about OSS processes can be represented in this model. In a CSS process, one would expect there to be a small community of *special* users, the in-house testers, who put in a big "lump" of executions early on in the life-cycle and after major changes. These users have very high $r_{i,j}$ and aim to have high $q_{i,j}$ as they often *try* to cause failures. Yet theory shows [1, 2, 3] that if their $q_{i,j}$ for certain faults are lower than for "ordinary" users, the latter may see much worse reliability than the in-house testers. There is anecdotal evidence that this happens with many products. In an OSS process, this nucleus of heavy duty testers may well be smaller. On the other hand, it is plausible that "ordinary" users of OSS have $r_{i,j}$s that are often much higher than with "ordinary" users of CSS, due to perceived higher chance of obtaining a fix; visibility of the source code and higher number of potential "fixers" should give higher $f_i s$ than for many commercial products (at least after some time from release).

2.4 Brief discussion of assumptions.

Of the assumptions of this model, the ones that seems most seriously unrealistic are those of a constant r_{ij} for given i, j and of constant f_i. I.e., a user who observes a failure due to a certain fault more than once has the same probability of reporting it each time; and the probability of a fault being fixed depends only on whether it is ever reported. There are many plausible factors that would suggest other behaviours. For instance, users may be unlikely to report a failure that they recognise as similar to one already reported, especially by themselves; however, this does not matter as the model is only affected by a failure being reported *at least once*. For certain failures, users who have not reported them previously may become more willing to report them when they observe them again; but for others (or other users for the same failures) they may be most likely to report the failure at the first observation, and later discount it as a known nuisance. Many such psychologically plausible assumptions are possible. They could be represented in the model at the cost of added complexity, but this is premature during this first investigation. We will need later to examine under which circumstances they would, if true, cause serious changes in the results derived from the model.

We will not discuss here, for brevity, the other, more standard assumptions we made.

3 Diversity is useful

We now illustrate how models of this kind can be used to investigate the plausibility of quite general hypotheses. The hypothesis we investigate here

is: "Diversity is a good thing: all things being equal, it is better for users to have diverse demand profiles than for them to have the same profile."

We want to compare a situation where users have many diverse profiles with one in which all the users have the same profile (we will see below which one), if the total number of executions of the software is the same in both cases: $\sum T_k = T$. Starting with the former situation, we define our "ideal average equivalent user", where "average" refers to the effectiveness in fault reporting, measured by its effect on the potential reliability improvements if all faults reported are fixed. For each bug i, it is a user who has the average of all the considered profiles, weighted with their respective number of executions. Mathematically this corresponds to a user with associated parameters $r'_{i,j}$ and $q'_{i,j}$ such that

$$r'_{i,j}q'_{i,j} = \frac{\sum_{k\in\{users\}} T_k r_{i,k} q_{i,k}}{\sum_k T_k} \ . \tag{5}$$

Let us then consider a situation in which all the users are "average" users: all have identical parameters $r'_{i,j}$ and $q'_{i,j}$ satisfying equation (5) for all is. The theorem of arithmetic and geometric means [5] (see appendix 5) now tells us that, for each bug i, its probability of not being reported is smaller in the case of diverse users than in the case of all users having this ideal average profile

$$\prod_{k\in\{users\}} (1 - r_{i,k}q_{i,k})^{T_k} < \left(1 - r'_{i,j}q'_{i,j}\right)^{\sum_k T_k} \ , \tag{6}$$

unless all the products $r_{i,k}q_{i,k}$ are equal, in which case equality occurs. Combining the effect over all the failure regions i, we obtain

$$\sum_{i\in\{failure\ regions\}} q_{i,j} f_i \left(1 - \prod_{k\in\{users\}} (1 - r_{i,k}q_{i,k})^{T_k}\right) \ > \\ \sum_{i\in\{failure\ regions\}} q_{i,j} f_i \left(1 - \left(1 - r'_{i,j}q'_{i,j}\right)^{\sum_k T_k}\right) \ . \tag{7}$$

What does this tell us? *Any* user (with profile $q_{i,j}$, $r_{i,j}$) would prefer the previous exposure of the software to have been diverse rather than uniform, because diversity gives them higher reliability. There are two conditions here to represent "all things being equal" in our comparison of diverse-profile testing with uniform-profile testing. They are (i) that the same number of demands are executed in *each* case and, (ii) that the uniform profile is, in an intuitively appealing way, the mean of the different profiles used in the diverse-profile testing. Subject to all things being equal in this natural way, we have thus shown that diverse-profile testing is superior to uniform-profile testing, in the sense it can be expected to deliver greater reliability improvement to *every* user (no matter what their profile is).

4 Discussion

4.1 Further implications of the model — conjectures

We consider here a few questions of clear interest, and speculate about which answers the model would give to these questions. I.e., we formulate mathematical conjectures; if in the future we manage to prove them to be consequences of the model, they will gain the status of conjectures about the actual evolution of software reliability. We rely heavily on our understanding of similar models described in [1, 2, 3].

4.1.1 The individual user's viewpoint

Studies in software reliability usually refer to the *average* reliability over all users. In other words, they apply to predictions of the total number of failures observed by the whole population of users. Of course, if users have very diverse profiles, as is the case for many products, even a very good average will not avoid very poor reliability for some users. In other words, just because a product is known to be very reliable on average, I cannot trust that it will be very reliable for me. Our model is a step in the right direction for considering this issue: it refers to the reliability for each specific user, and thus it will allow one to study distributions rather than averages.

In a practical situation, what would the model predict for an individual user, or set of users, who benefit from the collective fault reporting and fixing effort?

A simple (approximate) analogy with the previous work cited is that it is "as though" there were two users, user 1 whose viewpoint we are taking, and user 2 representing all the other users and executing many more demands, $T_2 >> T_1$ (or $T_1 = 0$, as when user 1 is deciding whether to adopt the software). We can compare two scenarios with equal total fault-detecting and reporting efficacies, i.e., two scenarios which, if one tried to estimate the reliability of the program (averaged among all users) by looking at the rate of generation of fault reports, would give identical estimates.

Suppose that user 1's profile is very different from that of all others. We can expect that after any amount of time and use, pfd_1 will be better than if user 1 were alone to report failures; yet much worse than if all users had the same profile as user 1 (this is the standard argument in favour of "operational" testing). Yet the situation is probably more complex. In [2] it was found that if software is tested by user 2 with a constant profile, there is a single profile that delivers the best reliability for user 1, among all profiles that yield the same initial probability of failure per execution, in other words under the constraint

$$\sum_{i \in \{failure\ regions\}} q_{i,1} = \sum_{i \in \{failure\ regions\}} q_{i,2} \ , \tag{8}$$

This optimal profile that user 2 should apply for maximum benefit to user 1 actually depends on the amount of use/testing, T. Its parameters $q_{i,2}$ satisfy the following relation ([2], eq. 28):

$$q_{k,1}\left(1-q_{k,2}\right)^{T-1} = q_{l,1}\left(1-q_{l,2}\right)^{T-1} \quad . \tag{9}$$

for any two faults k and l, under the constraint (8).

What does our "diversity is good" result adds to this previous result? If I (user 1) have to choose, for my software, among testing regimes with constant profile (as though executed by the single other user 2), then equation (9) tells me how to choose this profile for maximum reliability. But on top of that, if I then split my total number T of test cases among many users, and give them more varied profiles, of which this optimal profile is the average according to equation (5), I will get even better reliability for myself. Software that has been used by a diverse community can reach me with better reliability than software that has been used by "the best possible" (from my viewpoint) uniform community with comparable total usage and fault reporting efficiency.

Other interesting questions deserve investigation, e.g.: under which conditions could a user be aware of being too "special" to benefit greatly from diversity in the user community? When is it that an apparently very good fault finding process (large rs and qs) is less useful for me than a "worse" one (generally smaller rs and qs)?

And if "diversity is good", is there a sense in which "more " diversity is better than "less"? We need to characterise what "more" means in terms of the model's parameters before we can ask whether differences between OSS and other approaches matter in this respect. For instance, we may ask under which conditions a law of diminishing returns would apply for diversity.

4.1.2 Evolution over time

All the results so far have been discussed in terms of reliability at a certain, arbitrary moment in the history of use of the program. All results contain parameters T_k, or their sum $T = \sum_k T_k$.

We are really interested in how the program's reliability evolves over time. We showed in [3] a phenomenon whereby testing with a profile similar to one's usage profile yields better reliability growth in the short term, but in the long term different profiles, with some emphasis on the "less important" bugs, are more beneficial (i.e., in the long run, the "important" bugs will have been found and fixed no matter what; but the other ones are difficult to get rid of).

In our model we can therefore conjecture two main contributing factors to the reliability growth as observed by user j. In the short term, corresponding to initial rapid reliability growth, it is affected mostly by those users with similar profiles to user j's own. In the long run, the profiles

which differ from user j's will contribute more to improving reliability as seen by j.

4.1.3 Predictability of results, dependability of process

We have so far referred to the average, or expected value of reliability measures for a given user. This acknowledges that reliability growth is a stochastic process: for instance, a fault i with high $q_{i,j}$s is likely to be discovered early on, but it may well (with low probability) go undetected for a long time. The probabilities of different histories of reliability growth are determined by our model parameters. The averages that we have been discussing are defined over all the possible histories of reliability growth. So, they are useful indicators, but they may be misleading as they hide the potential variation between different histories.

In reality, what matters in a project is the reliability growth history that actually takes places. When I am stuck with an unreliable product, it does not matter much that, if I consider all other possible histories, the average of all the reliability levels that I *could* have obtained would be much better than the one I actually see. So, in project decisions the probabilities of "bad" reliability growth histories - thus, the probabilities of histories that are "much worse" than average - matter.

In [3] we studied probability distributions of reliability growth histories, accounting also for the fact that the initial set of faults is unknown. We could show some counterintuitive examples of how comparatively bad failure reporting rates, from usage with a user's own usage profile, would be better defences against the risk of very poor reliability growth than even much higher reporting rates based on someone else's, different profile.

We would like to explore how this translates into predictions, for a new prospective user, of the risk of very bad reliability growth after adopting a new product, and how the degree of "openness" of the process and diversity of the user base should affect them.

4.2 Limits of this model and possible extensions

This model is not exhaustive: there are many aspects of OSS processes that we have not described. Some of these could be studied through extensions to the model, some require different methods. To give some examples:

- in OSS an individual user might make available to the other users a fix which would work only in his particular profile, which is often commercially unviable in CSS;
- having additional users reporting bugs may be generally useful as it will increase the chances to improve the reliability in any case. However, in this model there is no explicit representation of any resource

bottleneck, like scarcity of bug fixing staff, or simply delays due to the need to coordinate many fixes. As an example, consider the situation when bug fixing is a competing activity. Bug fixers may need to abandon one bug for another, so that increasing the likelihood of user k reporting bug i implies a higher probability of fixing bug i, but at the expense of fixing bug l. Mathematically this can be described by saying that increasing $r_{i,k}$ increases f_i, which in turn decreases f_l.

- the model lacks any notion of consequence of a failure, or failure "severity", yet it seems likely that users will be influenced by this (as well as their perception of its frequency) in deciding whether or not to report it. It may well be that attitudes to failure severity are different between OSS and commercial developments.

5 Conclusions

We started this work to clarify to ourselves some open issues about the relationship between software processes and software dependability, using the tools we know, i.e. a simple, formal, probabilistic approach. We were motivated by interest in open source processes, but a reader may well say that this paper is not about"open source". Our considerations about dependability attributes, and the model we propose, apply to any scenario in which the important factors at work can be described by the model's parameters. It so happens that some of these parameters are plausibly influenced by "openness" factors. E.g., if a development community appears responsive to failure reports, we may well expect users to report failures more often than they would otherwise. Such thorough reporting of failures might of course be produced by appropriate management decisions or cultural factors even without open source development.

So, we have not studied "open source" per se. Yet this way of modelling does seem useful for decisions involving the choice between more or less open project styles, both in managing development projects and in software procurement. The model's universe includes many of the factors that seem important: the initial quality of the software, various factors affecting its growth, the different reliability levels experienced by different users, etc.

In fact, we would argue that only questions of the kind we have chosen, "how does factor X affect dependability attribute Y" are likely to produce interesting general answers. Questions of the form "how reliable is open source software?" may elicit highly specific, useful answers of the form "product X achieved an average time between crashes of y days for user community Z", but no generalisation without the support of theories – i.e., of models.

As to useful answers that this model can give to our questions, we have only proven a first theorem. This theorem does resolve an intuitively unclear

issue, but does not seem especially useful in the practical situations, in which two processes are likely to differ in all the $r_{ij}q_{ij}$ in arbitrary ways. However, even this modest theorem has some practical implications. E.g., given a community of similar users who are not likely to report all failures, it would pay off to encourage different users to focus on reporting different kinds of failures, provided their reporting frequencies, taken together, still satisfy equation (5).

We will continue to interrogate this model, e.g. about the relationships between reliabilities observed by different users, as outlined in the previous sections.

We plan to obtain clarifications of what "one should believe" given plausible assumptions, and thus about the consistency of various claims about the differences between processes that differ in the extent of some aspect of "openness". We shall investigate which predictions can be checked empirically to decide whether these models, however stylised, are useful approximations for important, dependability-related aspects of real, complex processes.

We have chosen to study aspects of the software life-cycle for which our modelling approach seems appropriate. We are sure that there are other aspects for which it would be ineffective. For instance, some will argue that a prestigious open source project will produce high initial dependability, or long-term maintainability, simply because it will attract highly skilled and dedicated contributors. In our model, this result would be represented by low values of the sum in (1) for all js. The process it describes is not represented in the model, nor would we expect any advantage from trying to represent it. To investigate how this conjecture should affect one's decisions, e.g. in setting up an open-source project, one would need to use knowledge from psychology, sociology or economics, and still would expect rather imprecise answers. For someone intent on procuring a software product, the conjecture is probably irrelevant, as what matters if whether that specific product *has* been built by especially skilled individuals, and/or *is* very reliable.

We expect our style of modelling to complement such other methods of investigating software engineering problems, in terms of general laws or of specific cases.

Acknowledgements

This work was supported in part by EPSRC, within the Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems, DIRC.

Appendix: theorem

In this appendix we describe the theorem of arithmetic and geometric means. Let us consider two set of non-negative numbers $a_1, a_2, \ldots, a_n$ and $p_1, p_2, \ldots, p_n$. We will call the ps weights. The weighted arithmetic means of the as is defined as

$$A(a,p) = \frac{\sum_{k=1}^{n} p_k a_k}{\sum_{k=1}^{n} p_k} \ ,$$

whereas the weighted geometric means is defined as

$$G(a,p) = \left(\prod_{k=1}^{n} a_k^{p_k} \right)^{1/\sum_k p_k} \ .$$

The theorem ([5], p.17) says that $G(a,p) < A(a,p)$ unless all the as are equal. Raising both sides to the power $\sum_k p_k$ the theorem gives the equation

$$\prod_{k=1}^{n} a_k^{p_k} < \left(\frac{\sum_{k=1}^{n} p_k a_k}{\sum_{k=1}^{n} p_k} \right)^{\sum_k p_k} \ . \tag{10}$$

Replacing a_k with $(1 - r_{i,k} q_{i,k})$ and the weights p_k with the number of executions T_k in equation (10) yields equation (6). Indeed, we find

$$\prod_{k=1}^{n} (1 - r_{i,k} q_{i,k})^{T_k} < \left(\frac{\sum_{k=1}^{n} T_k \, (1 - r_{i,k} q_{i,k})}{\sum_{k=1}^{n} T_k} \right)^{\sum_k T_k} \ .$$

In the right hand side term we can recognise our definition of the "ideal average user". Indeed, by the definition of "ideal average user"(5), we have

$$\left(\frac{\sum_{k=1}^{n} T_k \, (1 - r_{i,k} q_{i,k})}{\sum_{k=1}^{n} T_k} \right) = 1 - \frac{\sum_{k=1}^{n} T_k r_{i,k} q_{i,k}}{\sum_{k=1}^{n} T_k} = 1 - r'_{i,j} q'_{i,j} \ .$$

References

[1] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Choosing a testing method to deliver reliability. In *Proceedings 19th International Conference on Software Engineering ICSE'97*, pages 68–78, Boston, USA, 1997. IEEE Computer Society Press.

[2] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, SE-24(8):586–601, 1999.

[3] M. Pizza and L. Strigini. Comparing the effectiveness of testing methods in improving programs: the effect of variations in program quality. In *Proceedings 9th International Symposium on Software Reliability Engineering, ISSRE '98*, pages 144–153, Paderborn, Germany, 1998. IEEE Computer Society Press.

[4] L. Strigini. On testing process control software for reliability assessment: the effects of correlation between successive failures. *Software Testing Verification and Reliability*, 6(1):36–48, 1996.

[5] G. Hardy, J.E. Littlewood, and G. Polya. *Inequalities*. Cambridge University Press, 1952.

A Business Case Study of Open Source Software

Carolyn A. Kenwood
The MITRE Corporation
(kenwood@mitre.org)

Abstract

Based on The MITRE Corporation research project "Open Source Software in Military Systems,"[1] this paper analyzes the business case of open source software. It is intended to help Program Managers evaluate whether open source software and development methodologies are applicable to their technology programs. The paper explains open source, describes its significance, compares open source to traditional commercial off-the-shelf (COTS) products, presents the military business case, shows the applicability of Linux to the military business case, analyzes the use of Linux, discusses anomalies, and provides considerations for military Program Managers. This paper was prepared for the 2002 Open Source Software Development Workshop (Newcastle, UK) and highlights the importance of reliability and availability to the military business case.

What Is Open Source?

Open source, by definition, means that the source code is available. Open source software (OSS) is software with its source code available that may be used, copied, and distributed with or without modifications, and that may be offered either with or without a fee. If the end-user makes any alterations to the software, he can either choose to keep those

---

[1] The MITRE Corporation is a not-for-profit corporation working in the public interest. MITRE addresses issues of critical national importance, combining systems engineering and information technology to develop innovative solutions that make a difference. MITRE received a Leadership Award from the non-profit Potomac Forum for investigating the technology and economics of open source software in its research project "Open Source Software in Military Systems."

changes private or return them to the community so that they can potentially be added to future releases[2]. An open source license is certified by the Open Source Initiative (OSI), an unincorporated nonprofit research and educational association with the mission to own and defend the open source trademark and advance the cause of OSS. The open source community consists of individuals or groups of individuals who contribute to a particular open source product or technology. The open source process refers to the approach for developing and maintaining open source products and technologies, including software, computers, devices, technical formats, and computer languages.

Although OSS has recently become a hot topic in the press, it has actually been in existence since the 1960s and has shown a successful track record to-date. Examples of popular open source products include Emacs, GNU toolset, Apache, Sendmail, and Linux. The development of Perl is an example of the open source process.

Emacs was one of the first open source products. It is a text editor that is widely used for software development. As a software tool, many developers (including defense contractors) use Emacs to develop their (non-open source) applications.[3] The success of Emacs led to the GNU program. GNU stands for "Gnu's not Unix." The GNU project consists of an operating system kernel and associated Unix tools. The GNU tools have been ported to a wide variety of platforms, including Windows NT. Again, they are widely used by software developers to produce both open source and proprietary software.[4]

The Apache web server is a freely available web server distributed under an open source license. Apache developers form a voting committee, and votes from this committee set the direction for the project. The Apache Software Foundation provides organizational, legal, and financial support for Apache projects. Apache web servers are known for their functionality and reliability. They form the backbone infrastructure running the Internet. Today, Apache comprises over 60 percent of the web server market and continues to grow.[5]

---

[2] There are several licensing models for Open Source. Some require that all changes made to the source must be freely distributed with the modified product. Other licenses permit an organization to make changes and keep the changes private.

[3] For more information on Emacs, see http://www.gnu.org/software/emacs/emacs.html.

[4] For further information on GNU, visit the GNU Project web server at http://www.gnu.org/.

[5] O'Reilly, Tim, Linux eSeminar Series, 1999. For more information on Apache, refer to the Apache Software Foundation at http://www.apache.org/.

Sendmail is a platform for moving mail from one machine to another. The Sendmail Consortium, a nonprofit organization, runs the open source program and maintains a website to serve as a resource. Sendmail is estimated to carry nearly 90 percent of e-mail traffic.[6]

Linux is an open source Unix-like operating system (OS). The kernel is maintained by the Linux community, led by Linus Torvalds, the creator of Linux.[7] Torvalds has appointed delegates who are responsible for managing certain areas of the project and, in turn, these delegates have a team of coordinators. Linux has multiple uses; it can be used as an OS for a server, desktop, or embedded environment. There are over ten million Linux users worldwide. According to an InformationWeek survey, Linux comprises about 4 percent of all operating systems, and that number is expected to rise to 15 percent in two years.[8] Linux is the fastest growing server operating environment, increasing from 16 percent of the market in 1998 to 25 percent in 1999.[9] In the embedded market, Linux is also expected to play a significant role. [10] (An embedded device is a piece of microprocessor-based computing hardware, usually on single circuit board, which has been built to run a specific software application. The term *embedded* refers to the fact that these devices were originally used as building blocks in larger systems.)

While Emacs, GNU toolset, Apache, Sendmail, and Linux are examples of open source products, the Practical Extraction and Reporting Language (Perl) is an example of an open source process. Perl is a system administration and computer-programming language widely used throughout the Internet. It is the standard scripting language for all Apache web servers, and is commonly used on Unix. Perl is managed on a rotating basis by the ten to twenty most active programmers. Each takes turns managing different parts of the project. There are an estimated one million Perl users today.[11]

---

[6] O'Reilly, Tim, and Ether Dyson, "Open Mind, Open Source." For more information on Sendmail, see http://www.sendmail.org/.

[7] Linus Torvalds' homepage can be found at http://www.cs.Helsinki.FI/u/torvalds/.

[8] Ricadela, Aaron, "Linux Comes Alive," InformationWeek, January 24, 2000.

[9] "The Future of Linux," CNet 2000, cites IDC data, no date provided.

[10] For further information on Linux, visit The Linux Home Page http://www.linux.org/, Linux International http://www.li.org/, and MITRE Linux Resources Page http://w030nt.mitre.org/users/terry/pub/linux/.

[11] For more information on Perl, visit http://www.perl.com/pub.

Significance of Open Source

The open source development process differs sharply from the traditional commercial off-the-shelf (COTS) model. In the corporate or traditional COTS model, whereby a corporation produces and sells proprietary software, COTS products tend to be driven by time-to-market considerations. Development is done under the aegis of the COTS vendor, who views the source code as valuable intellectual property, and controls when versions of the software are released. Eric Raymond likens this corporate model to a cathedral and the open source model to a bazaar.[12] In the corporate model, Raymond depicts individuals or small groups of individuals quietly and reverently develop software in isolation, without releasing a beta version before it is deemed ready. In contrast, the open source model relies on a network of "volunteer" programmers, with differing styles and agendas, who develop and debug the code in parallel. From the submitted modifications, the delegated leader chooses whether or not to accept one of the modifications. If the leader thinks the modification will benefit many users, he will choose the best code from all of the submittals and incorporate it into the OSS updates. The software is released early and often.

Benefits and Risks of Open Source Software Compared to Traditional COTS

Due to the different development models, Program Managers can achieve many benefits over traditional COTS by using OSS. Popular open source products have access to extensive technical expertise, and this enables the software to achieve a high level of efficiency, using less lines of code than its COTS counterparts. The rapid release rate of OSS distributes fixes and patches quickly, potentially an order of magnitude faster than those of commercial software. OSS is relatively easy to manage because it often incorporates elements such as central administration and remote management. Because the source code is publicly available, Program Managers can have the code tailored to meet their specific needs and tightly control system resources. Moreover, Program Managers can re-use code written by others for similar tasks or purposes. This enables Program Managers to concentrate on developing the features unique to their current task, instead of spending their effort on re-thinking and re-writing code that has already been developed by others. Code re-use reduces development time and provides predictable results. With access to the source code, the lifetime of OSS systems and their upgrades can be extended indefinitely. In contrast, the lifetime of traditional COTS systems and their upgrades cannot be extended if the vendor does not share its code and either goes out of business, raises its prices prohibitively, or reduces the quality of the software prohibitively. While traditional COTS typically depends on monopoly support with one company providing support and "holding all the cards" (i.e., access to the code) for a piece of software, the publicly available source code for OSS

---

[12] Raymond, Eric, "The Cathedral and the Bazaar," O'Reilly Associates, 1999.

enables many vendors to learn the platform and provide support. Because OSS vendors compete against one another to provide support, the quality of support is potentially increased while the end-user cost of receiving the support is decreased. Open source can create support that lasts as long as there is demand, even if one support vendor goes out of business. For government acquisition purposes, OSS adds potential as a second-source "bargaining chip" to improve COTS support.

OSS can be a long-term viable solution with significant benefits, but there are issues and risks to Program Managers. Poor code often results if the open source project is too small or fails to attract the interest of enough skilled developers; thus, Program Managers should make sure that the OSS community is large, talented, and well-organized to offer a viable alternative to COTS. Highly technical, skilled developers tend to focus on the technical user at the expense of the non-technical user. As a result, OSS tends to have a relatively weak graphical user interface (GUI) and fewer compatible applications, making it more difficult to use and less practical, in particular, for desktop applications (although some OSS products are greatly improving in this area). Version control can become an issue if the OSS system requires integration and development. As new versions of the OSS are released, Program Managers need to make sure that the versions to be integrated are compatible, ensure that all developers are working with the proper version, and keep track of changes made to the software. Without a formal corporate structure, OSS faces a risk of fragmentation of the code base, or code forking, which transpires when multiple, inconsistent versions of the project's code base evolve. This can occur when developers try to create alternative means for their code to play a more significant role than achieved in the base product. Sometimes fragmentation occurs for good reasons (e.g., if the maintainer is doing a poor job) and sometimes it occurs for bad reasons (e.g., a personality conflict between lead developers). The Linux kernel code has not yet forked, and this can be attributed to its accepted leadership structure, open membership and long-term contribution potential, GNU General Public License (GPL) licensing eliminating the economic motivations for fragmentation, and the subsequent threat of a fragmented pool of developers. The small amount of fragmentation between different Linux distributions is good because it allows them to cater to different segments. Users benefit by choosing a Linux distribution that best meets their needs. Finally, there is a risk of companies developing competitive strategies specifically focused against OSS.

When comparing long-term economic costs and benefits of open source usage and maintenance to traditional COTS, the winner varies according to each specific use and set of circumstances. Typically, open source compares favorably in many cases for server and embedded system implementations that may require some customization, but fares no better than COTS for typical desktop applications. Indeed, some literature sources generalize that open source products are no worse than closed source, but our findings indicate that the scale measuring the value derived from open versus closed source software can be heavily tipped

in one direction or the other depending on the specific requirements and runtime environment of the software.

A decision between OSS and traditional COTS is based on three factors: (1) costs – both direct (e.g., price of software) and indirect (e.g., end-user downtime); (2) benefits (i.e., performance); and, (3) other, more intangible criteria (e.g., quality of peer support). Direct costs are largely understood and have traditionally comprised most of the total lifecycle costs of a system. However, indirect costs as well as operational and performance benefits (e.g., scalability, reliability, and functionality) play a most influential economic role in today's more mature software market. Other, more intangible criteria are difficult to quantify, but can also impact the effectiveness of open and closed source software. Because indirect costs and operational and performance benefits play a much larger role in OSS compared to traditional COTS products, traditional lifecycle cost models and other COTS software tools can no longer be relied on for optimal mission-oriented and IT investment decision-making involving a choice of OSS.

To understand how indirect costs should be incorporated into the analysis, Program Managers must understand what these costs mean to their programs. Since the salary and other labor costs associated with an employee are direct costs, only the labor costs that are "wasted" and could be used in more productive ways should be included as indirect costs. In other words, although there is no additional direct cost to the organization, not as much output was received from the employee due to inefficiencies in the process or system. To a profit-making organization it would be hoped that this improved productivity increases profits. For example, time wasted could be spent bringing in more business. Within a Department of Defense (DOD) organization, the concepts of bringing in more business and increasing profits do not apply, and these lost productivity costs could be viewed as justification for force structure cuts. If, for example, an organization migrates to a new solution and experiences improved productivity, the organization could perform the same job with fewer people.) Data collection efforts to understand these metrics are viewed negatively by employees for this reason. Unless a direct cause-and-effect link can be established, it may be that some indirect influences are best viewed as relative costs rather than as absolute costs in support of IT investment analyses.

Program Managers need a complete taxonomy of lifecycle costs, benefits, and other, more intangible criteria to account for hidden costs and benefits that they might otherwise have overlooked. With this taxonomy, Program Managers can make software-purchasing decisions being fully aware of their economic, performance, and mission implications. The following table represents a cost element taxonomy for OSS developed by this research investigation.

Table 1. OSS Cost Element Taxonomy[13]

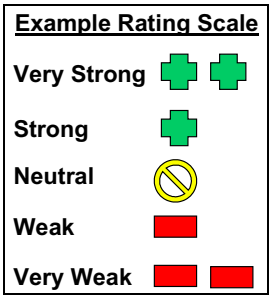
Direct Costs
Software and Hardware
Software
Purchase price
Upgrades and additions
Intellectual property/licensing fees
Hardware
Purchase price
Upgrades and additions
Support Costs
Internal
Installation and set-up
Maintenance
Troubleshooting
Support tools (e.g., books, publications)
External
Installation and set-up
Maintenance
Troubleshooting
Staffing Costs
Project management
Systems engineering/development
Systems administration
Vendor management
Other administration
Purchasing
Other
Training
De-installation and Disposal
Indirect Costs
Support Costs
Peer support
Casual learning
Formal training
Application development
Futz factor
Downtime

[13] Futz factor is included by GartnerGroup as an indirect cost. GartnerGroup describes this term as the labor expense when the end-user exploits corporate computing assets for his own personal use during productive work hours.

In addition to a taxonomy of lifecycle costs, Program Managers also need a taxonomy of benefits and risks along with an example rating scale to compare the costs, benefits, and other, more intangible criteria of OSS and traditional COTS software. This research developed a taxonomy of benefits and risks for OSS and an example rating scale, and these are presented in Table 2 below.

Table 2. OSS Taxonomy of Benefits and Risks

Qualitative Attributes
Ability to customize
Availability/reliability
Interoperability
Scalability
Design flexibility
Lifetime
Performance
Quality of service and support
Security
Level of difficulty/ease of management
Risk of fragmentation
Availability of applications



The above taxonomy comprises a list of qualitative attributes. For each attribute, Program Managers should compare the relative strength or weakness for OSS versus traditional COTS products. A relative strength would indicate a benefit, and a relative weakness would indicate a risk. An example rating scale is shown above for comparing the relative value of OSS versus traditional COTS. This example scale presents five ratings – very strong, strong, neutral, weak, and very weak. Since the ratings will differ depending on the specific use and environment of the software, Program Managers should customize their ratings according to their particular circumstances.

Compared to traditional COTS products, OSS provides more options to Program Managers for life-cycle supportability. The maintenance burden of OSS can be similar to

pure COTS ("buy"), custom code ("build"), or lie somewhere in between. Unmodified OSS can be considered similar to pure COTS. Thoroughly modified and owner-maintained OSS is comparable to custom code. "Modifiable COTS," or OSS that relies on short-term modifications yet attempts to re-merge with newly released OSS updates, takes advantage of the benefits of both pure COTS and custom code. The following diagram illustrates this spectrum and points out differences between the above scenarios.

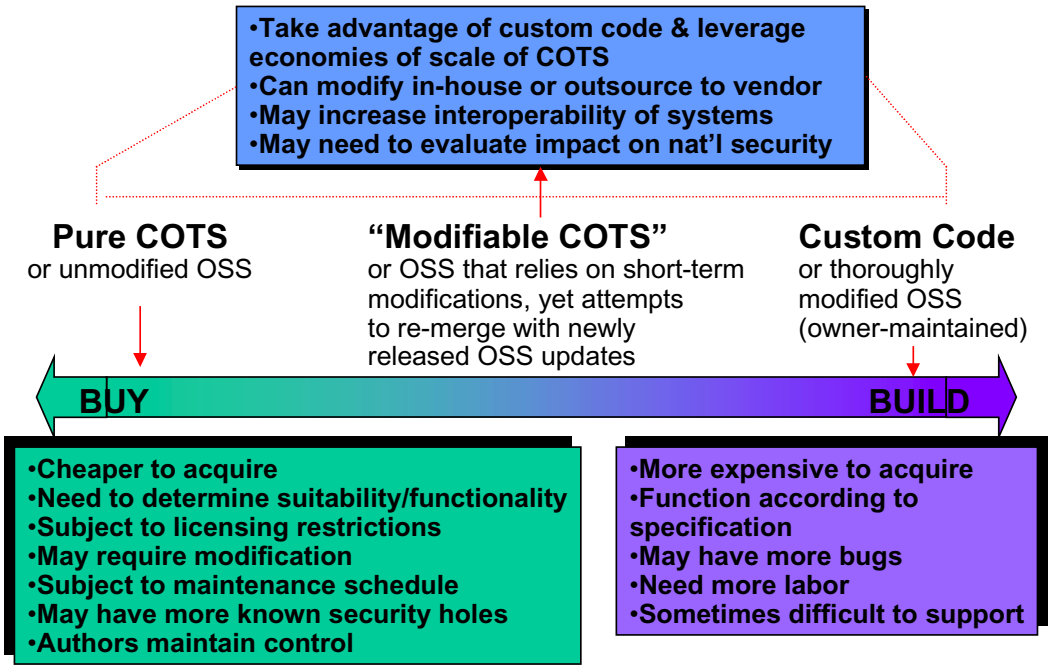


Figure 1. OSS Provides Several Maintenance and Support Options

Program Managers should evaluate the relative advantages and disadvantages of the pure COTS, "modifiable COTS," and custom code maintenance models for their specific use and set of circumstances. Pure COTS is advantageous because it is cheaper to acquire. However, Program Managers need to assess the suitability and functionality of the software to their specific needs. The software may require modification, and Program Managers are subject to licensing restrictions and set maintenance schedules. Pure COTS may have more known security holes, and control is maintained by the authors of the software. "Modifiable COTS" takes advantage of customer code while leveraging the economies of scale achieved by COTS products. The software can be modified in-house or by a vendor. The interoperability of systems may be increased with "modifiable COTS." The impact on national security may need to be evaluated. Custom code is more expensive to acquire, functions according to specification, may have more bugs, requires more labor, and is sometimes difficult to support.

Open source will benefit the government by improving interoperability, long-term access to data, and ability to incorporate new technology. Interoperability increases because open source enables the same code, documentation, and data formats to be used in every system component. (However, the downside risk of exposure should be evaluated; if the security of an open source system is compromised, interoperability could also be compromised.) Long-term access to data gives the user full access to its own systems. It is possible to contract out maintenance development work to support vendors, who have the same information as the original supplier. Open source can allow the government to more easily adopt new technology because it reduces the cost and risk of change. Open source projects tend to be evolutionary and less disruptive to operations.

The Military Business Case

The military has different software needs than the commercial sector because of its unique mission and environment. Software attributes most important to the commercial sector include application choice, ease of use, service and support, price, reliability, and performance. Most operationally significant attributes for software used in the military include reliability, long-term supportability, security, and scalability. Additional attributes of highest programmatic significance to the military include cost or price, availability or multiple distribution sources, and popularity or brand/reputation.

While both the commercial and government sectors are concerned about price and reliability, certain commercial customers generally have less stringent requirements for security, availability, and long-term supportability. However, these features are becoming more important in the private sector. E-commerce companies must have high levels of security to protect personal financial information and transactions. Availability of software from multiple sources increases competition, resulting in higher quality at low prices. Long-term supportability is important to businesses needing to access legacy data. If a commercial product or process, such as open source, is deemed suitable and offers the required functionality, the military can take advantage of these to achieve significant cost savings. There are other potential benefits to leveraging commercial products or processes, including faster deployment time, improved quality and reliability, reduced development risks, and a support system already in place.

Applicability of Linux to the Military Business Case

Linux has attracted a large group of highly trained developers, and "given enough eyeballs, all bugs are shallow."[14] Over 120,000 programmers contribute to Linux,

---

[14] Raymond, Eric, "The Cathedral and the Bazaar," O'Reilly Associates, 1999.

volunteering about 2 billion dollars worth of labor.[15] This massive amount of technical expertise could not be afforded by providers of traditional COTS products. As a result of the open source process, highly reliable and stable software is produced.

Reliability and availability are essential to the military. Reliability refers to the ability to produce the same acceptable result on successive trials, and availability is the readiness of the system for immediate use. In the case of Linux vs. Windows, reliability is a primary objective of the Linux community and one of the greatest weaknesses of Windows. Since so many programmers work to improve the Linux code, bugs are more likely to be discovered and fixed to improve the software's stability. Also, the Linux kernel uses a virtual memory management system that shares memory across all active programs. It gives each program a separate virtual address space, reducing the effect of one program on another. This management system also prevents programs from overwriting critical areas of memory (i.e., areas where Linux kernel is stored). GNet reports that the computer usually must be restarted when Windows NT incurs reconfiguration or software loading problems; this is usually not necessary for Linux. Benchmarking studies agree that Linux is more reliable than Windows. The Bloor Research benchmarking study measured the uptime/downtime of Linux and Windows NT over the period of one year. Over that time, the Linux machine in the study crashed once because of hardware fault (disk problems), and it took four hours to fix. Windows NT crashed 68 times due to hardware problems, memory, file management, and a number of miscellaneous problems, all of which took 65 hours to fix. Thus, the availability of Linux was 99.95 percent and the availability of NT 99.26 percent. In a similar benchmarking study, Giga Information Group determined the availability of Unix as 99.8 percent and the availability of Windows NT as 99.2 percent.[16]

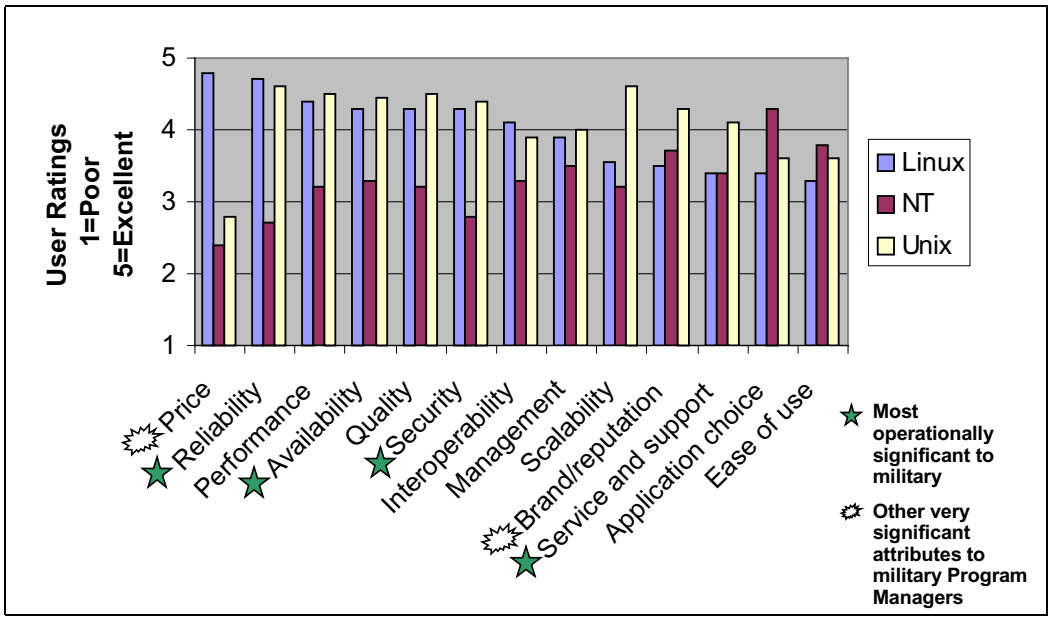
This comparative advantage in reliability and availability, along with its perceived low price, enables Linux to attract a large user base worldwide. The following graph compares user ratings of Linux, NT, and Unix.[17] While Linux is used because of its perceived low price and reliability, NT is preferred for its choice of applications and ease of use. Users select Unix for its performance, availability, quality, security, management, scalability, brand/reputation, and service and support.

---

[15] Orzech, Dan, "Linux and the Saga of Open Source Software," Datamation, February 1999 and Dan Kaminsky, "Core Competencies: Why Open Source is the Optimum Economic Paradigm for Software," March 2, 1999.

[16] DiDio, Laura, cited by Derek Slater, "Deciding Factors - Operating Systems," CIO Magazine, February 1, 2000 and Frans Godden, "How do Linux and Windows NT Measure Up in Real Life?" GNet, January 2000.

[17] US Linux user ratings by server OS from Michelle Bailey, Vernon Turner, Jean Bozman, and Janet Waxman, "Linux Servers: What's the Hype, and What's the Reality," IDC, March 2000.



Source: US Linux user ratings by server OS from Michelle Bailey, Vernon Turner, Jean Bozman, and Janet Waxman, "Linux Servers: What's the Hype, and What's the Reality," IDC, March 2000.

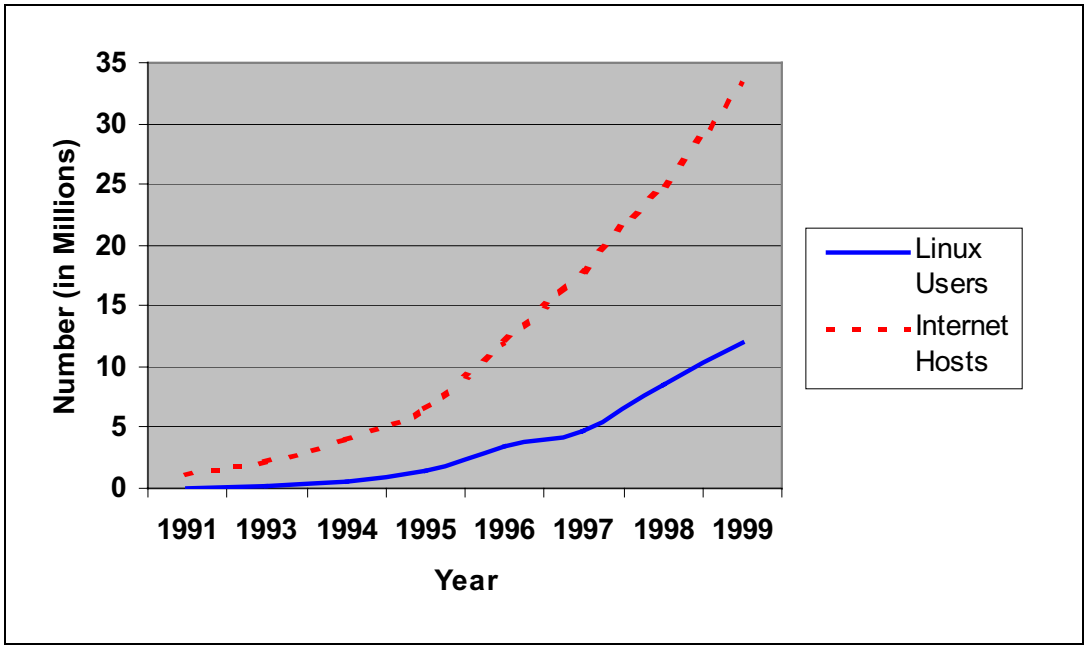
Figure 2. Military and Commercial User Benefits of Linux

Use of Linux

The number of Linux users worldwide has grown from 1 user (Linus Torvalds) in 1991 to an estimated 12 million users in 1999. The following graph plots the number of Linux users worldwide against the number of Internet hosts worldwide, and shows that the number of Linux users has been growing with the number of Internet hosts. As the Internet expands, the number and productivity of open source development teams increase and attract more users.[18]

---

[18] Linux estimates derived from GartnerGroup, IDC, and Red Hat market research. Internet estimates based on research from Bruce L. Egan, 1996. Data based on year-end estimates.

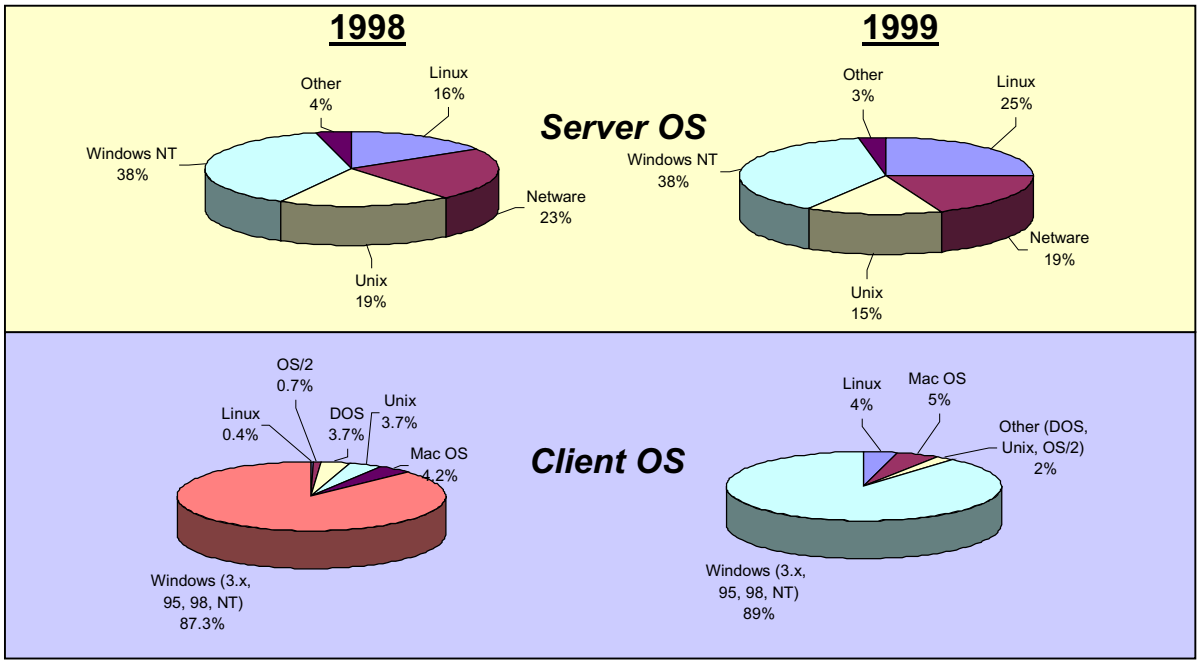


Source: Linux estimates derived from GartnerGroup, IDC, and Red Hat market research. Internet estimates based on research from Bruce L. Egan, 1996. Data based on year-end estimates.

Figure 3. Worldwide Success of Linux in the Marketplace

More Linux installations are expected in the server market than the client OS market. Significant investments in areas such as ease of use and configuration are needed for Linux to achieve success on desktops. The following pie charts shows the Linux market share for the server and client OS market in 1998 and 1999.[19]

[19] "The Future of Linux," CNet, 2000 cites IDC, no date provided.



Source: "The Future of Linux," CNet, 2000 cites IDC.

Figure 4. Server and Client OS Market Share in 1998 and 1999

Although Linux deployments are widening, they are not deep. Between 1998 and 1999, the Linux server OS market share grew from 16 percent to 25 percent and the Linux client OS market share grew from 0.4 percent to 4 percent. It appears that most of this growth came from Unix users who switched to Linux.

Discussion: Windows/NT, Unix, and Linux

Although the open source development process offers many benefits over traditional COTS, Microsoft Windows continues to dominate the server and desktop markets. There are several reasons for this. First, Microsoft has invested significantly in marketing Windows to developers. Second, the NT platform enables servers from different vendors to work on NT. In fact, there are over 100 NT server vendors.[20] Third, users often choose Windows because of the large choice of compatible applications and its ease of use. There is an affinity between the desktop and server environments when Microsoft products are used. Fourth, Windows NT systems have historically had a much lower initial cost of entry compared to

---

[20] Deate Hohmann, GartnerGroup, phone conversation, December 2000.

Unix systems. Hardware and software costs are lower when using NT because the system runs on commodity components and standard chipset and storage devices. For the above reasons, Windows is perceived as a less risky choice by IT management. Industry analysts further add that "no one ever got fired for buying Microsoft."[21]

Despite these pro-Microsoft observations, GartnerGroup has concluded that one cannot generalize whether NT or Unix offers the least expensive long-term support. Instead, the least expensive choice depends on the specific application, environment, and current skill base of the organization.[22] It should also be noted that Windows does not scale as well as Unix, and this can turn the tables on the relative total costs of Windows versus Unix. NT is not as powerful as Unix and, according to GartnerGroup tests, NT can only support up to 1,000 concurrent users.[23] Smaller organizations that grow into larger ones must correspondingly add more boxes to support its larger user base. In some instances, five-times as many boxes of NT may be required to get the same performance as a Unix box. Organizations that do not plan for growth often choose Windows for its low initial cost of entry, while organizations that plan for aggressive growth upfront may choose Unix. Therefore, the optimal choice of Windows versus Unix depends on the number of users the system supports. As the number of users increase to over 1,000, Unix becomes the most effective platform, or optimal platform choice.

Since the recent surge in online use that has helped to fuel the maturation of Linux, there have been small migrations to Linux. Some users of Unix have shifted to Linux, a Unix-like OS. In addition, some start-up businesses with little capital choose Linux because it runs nicely on older computers. If more Program Managers compared OSS to traditional COTS for their specific business case, it is likely that there would be many more users of OSS today.

Considerations for Military Program Managers

OSS provides more options than traditional COTS for life-cycle supportability, particularly for long-lived systems. It can be used in the form of pure COTS, "modifiable COTS," or custom code. Program Managers' requirements for operating systems differ considerably depending on their particular environmental and mission requirements.

Command and Control (C2) Program Managers are operationally-driven. For these managers, the cost of failure is very high. Reliability and performance are essential. C2

---

[21] Garvey, Martin J., "The Hidden Cost of NT," InformationWeek, July 1998.

[22] Hohmann, Deate, GartnerGroup, phone conversation, December 2000.

[23] Hohmann, Deate, GartnerGroup, phone conversation, December 2000.

Program Managers use traditional COTS unless the system requires more customization, and system upgrades tend to be frequent. C2 Program Managers should consider using Linux because it provides the highest level of reliability with good performance. The Windows family is weakest for both of these metrics.

Information System (IS) Program Managers are driven by costs, quality of support, and application choice. Systems are generally replaced every five to seven years. If application choice is important, IS Program Managers should consider NT. Otherwise, Program Managers may find more service and support options with Unix and Linux. Tapping into the "modifiable COTS" option with Linux could provide very valuable additional features without the added maintenance burden associated with them.

Embedded/Weapon System Program Managers are driven by portability, ruggedness, and hard real-time requirements. System upgrades are typically expensive endeavors. Embedded/Weapon System Program Managers will likely find Linux most appealing. Its design flexibility enables the kernel to be either pared down to eliminate unnecessary features or expanded to include additional features. Linux is portable to many central processing units (CPUs) and hardware platforms. It is stable and scalable over a wide range of capabilities and easy to use for development. The software can dynamically reconfigure itself without rebooting. Linux can isolate faults and processes. Processes can load and remove kernel modules, device drivers, and custom modules based on available resources and dynamic application needs. The applications are also modular with well-defined interfaces. Furthermore, hard real-time capabilities are available from the Linux kernel extension RTLinux.

Conclusion

OSS is a viable long-term solution that merits careful consideration because of the potential for significant cost, reliability, and support advantages. However, these potential benefits must also be carefully balanced with a number of risks associated with OSS approaches and products. The optimal choice of OSS versus traditional COTS varies according to the specific requirements and runtime environment of the software. OSS is often a good option for products relevant and interesting to a large community with highly skilled developers. It typically compares favorably for server and embedded system implementations that may require some customization, but fares no better than traditional COTS for typical desktop applications. When making a decision about whether to use OSS or traditional COTS, it is recommended that Program Managers follow the five steps presented below.

1. **Assess the supporting OSS developer community (e.g., Linux, Apache).** Look for communities that are large, talented, and well organized.
2. **Examine the market.** Is there a strong and increasing demand for the specific OSS product? To what extent have vendors and service providers emerged in the commercial

marketplace to provide complementary services and support not available from the community?

3. **Conduct a specific analysis of benefits and risks.** The MITRE effort has developed a taxonomy of OSS benefits and risks (see Table 2) that can be used to compare candidate OSS products to your specific economic, performance, and mission objectives.
4. **Compare the long-term costs.** Use the MITRE-developed OSS Cost Element Taxonomy (see Table 1) to compare the long-term costs associated with usage and maintenance of OSS versus traditional COTS relative to your specific objectives.
5. **Choose and execute your strategy.** Following the previous four steps will provide enough information and detail to choose and then execute the most effective option combination of OSS, traditional COTS, and proprietary development to support objectives.

In conclusion, open source methods and products are well worth considering seriously in a wide range of government applications, particularly if they are applied with care and a solid understanding of the risks they entail. OSS encourages significant software development and code re-use, can provide important economic benefits, and has the potential for especially large direct and indirect cost savings for military systems that require large deployments of costly software products.

List of References

1. Bailey, Michelle, Vernon Turner, Jean Bozman, and Janet Waxman, “Linux Servers: What’s the Hype, and What’s the Reality?” IDC, March 2000.
2. Be, www.be.com. (Note that Be was sold recently to Palm.)
3. Bryar, Jack, “How Much Does Free Cost?” The Andover News Network, March 15, 2000.
4. Caldera, www.caldera.com.
5. Clark, Tim, “Network Associates Adds Linux Product,” CNET News, February 8, 1999.
6. Chime-Net; Medzilla, 1999; and Wageweb, 2000.
7. CoolLogic, www.coollogic.com.
8. Corel, www.corel.com.
9. Covey, Jeff, “A New Business Plan for Free Software,” Freshmeat, January 22, 2000.
10. Datapro, February 1999.
11. Debian, www.debian.org.
12. D. H. Brown Associates, “Linux: How Good Is It?” 1999.
13. Embedded Linux Consortium, www.embedded-linux.org.
14. Epplin, Jerry, “Linux as an Embedded Operating System,” October 1997, wysiwyg://4/http://www.embedded.com/97/fe39710.htm.
15. “French Ministry Adopts Open-Source Culture, Linux,” InfoWorld.com, February 8, 2000, http://www.infoworld.com/articles/ec/xml/00/02/08/000208eclinparis.xml.
16. “The Future of Linux,” CNet, 2000.
17. “Getting to Know Linux,” Colorado Business, July 2000.
18. Gillen, Al, and Dan Kusnetzky, “Linux Overview: Understanding the Linux Market Model,” IDC, February 2000.
19. Gutfraind, Alexander, “Introductory into the World of Linux,” The Linux World, http://www.tht.net/~gutfrnd/linux/intro/linworld.htm, 1998.
20. Harmon, Paul, “Linux and Architecture,” Cutter Consortium, Feb. 9, 2000.

21. Hohmann, Deate, GartnerGroup, phone conversation, December 2000.
22. Hontañón, Ramón J., UUNET, "Building a Robust Linux Security Solution, Network Magazine, 2000.
23. "Join the Freeware Revolution?" CIO, March 19, 1999.
24. Jordan, Peter, "Nibbling Away at UNIX," VARBusiness, January 14, 2000.
25. Kaminsky, Dan, "Core Competencies: Why Open Source is the Optimum Economic Paradigm for Software," March 2, 1999.
26. Kirch, John, "Microsoft Windows NT Server 4.0 Versus UNIX," August 7, 1999.
27. Lauriston, Robert, "The Un-Microsoft Office," ComputerCurrents.com, March 23, 1999.
28. Lerner, Josh and Jean Tirole, "The Simple Economics of Open Source," National Bureau of Economic Research, March 2000.
29. LinuxDevices, http://www.linuxdevices.com.
30. "Linux Is Biggest Focus Shift Since TCP/IP, Says IBM," Network News, August 30, 2000.
31. "National Security Agency Selects Secure Computing to Provide Type Enforcement on Linux OS," January 14, 2000.
32. "Open Source Code and the Security of Federal Systems," Report of the Ad Hoc Working Group, DARPA, GSA, NIST, and NSA, prepared for the National Coordinator for Security, Infrastructure Protection, ,and Counter-Terrorism, June 1999.
33. O'Reilly, Tim and Ether Dyson, "Open Mind, Open Source."
34. O'Reilly, Tim, Linux eSeminar Series, 1999.
35. Orzech, Dan, "Linux and the Saga of Open Source Software," Datamation, February 1999.
36. "PC DOS," IBM, http://www-3.ibm.com/software/os/dos/.
37. Quinlan, Daniel, "The Past and Future of Linux Standards," Linux Journal, June 1999.
38. Raymond, Eric, "The Cathedral and the Bazaar," O'Reilly Associates, 1999.
39. Raymond, Eric, http://www.opensource.org.
40. Ready, Jim and Bill Weinberg, "Leveraging Linux for Embedded Applications," LinuxDevices.com.
41. RedHat, http://www.redhat.com.

42. Ricadela, Aaron, “Linux Comes Alive,” InformationWeek, January 24, 2000.

43. Schmidt, Jürgen, “Mixed Double,” c’t (German technical computer magazine), 1999.

44. Secure Computing Corporation, “Reconsidering Assurance for Open Source Software,” November 30, 1999.

45. Seiferth, C. Justin, Major US Air Force, Deputy Chief, Global Air Traffic Operations Division, “Opening the Military to Open Source,” COTS Journal, November/December 1999.

46. Seiferth, C. Justin, Major, US Air Force, Deputy Chief, Global Air Traffic Operations Division, “Adoption of Open Licensing,” COTS Journal, November/December 1999.

47. Seiferth, C. Justin, Major, US Air Force, “Open Source and the United States,” Air Command and Staff College, Air University, June 11, 1999, http://ceu.fi.udc.es/GPUL/articulos/varios/US_DoD_and_OSS.txt.

48. Slackware, www.slackware.com.

49. Slater, Derek, “Deciding Factors - Operating Systems,” CIO Magazine, February 1, 2000.

50. Stoltz, Mitch, “The Case for Government Promotion of Open Source Software,” A NetAction White Paper, 1999.

51. SuSE, www.suse.com.

52. United States Air Force Scientific Advisory Board, “Ensuring Successful Implementation of Commercial Items in Air Force Systems,” SAB-TR-99-03, April 2000.

53. Valloppillil, Vinod, Microsoft, “Open Source Software,” edited by Eric Raymond as Halloween I, 1998.

54. Valloppillil, Vinod, and Josh Cohen, Microsoft, “Open Source Software,” edited by Eric Raymond as Halloween II, 1998.

55. Vaughn-Nichols, Steven J., “TripWire Delivers Open Source DDoS and Security Answer,” Sm@rt Reseller, March 1, 2000.

56. Weiner, Bruce, Mindcraft, “Open Benchmark: Windows NT Server 4.0 and Linux,” Mindcraft, June 30, 1999.

57. Weiss, G., “The Competitive Impact of IBM’s Linux Announcement,” GartnerGroup, February 8, 2000.

58. Weiss, G., “The GartnerGroup Server Operating System Forecast,” GartnerGroup, March 26, 2000.

59. Weiss, G., “How the Open Source Movement Will Affect Users,” GartnerGroup, January 26, 1999.

60. Weiss, G., “Linux Adoption Best Practices: A 10-Point Program,” GartnerGroup, February 8, 2000.

61. Weiss, G., “Updated OS Evaluation: Linux vs. Unix and Windows 2000,” GartnerGroup, July 25, 2000.

62. White, Walker, “Observations, Considerations, and Directions,” Oracle, May 8, 2000 cites Frederick Brooks in “The Mythical Man Month.”

63. Williams, Tom, “Linux Catches the Embedded Wave,” February 2000, http://www.embeddedsystems.com.

Interdisciplinary Insights on Open Source

Cristina Gacek, Tony Lawrie, and Budi Arief
Centre for Software Reliability
Department of Computing Science
University of Newcastle
Newcastle upon Tyne NE1 7RU
United Kingdom
{Cristina.Gacek, A.T.Lawrie, L.B.Arief}@ncl.ac.uk

Abstract

The term "open source" is widely applied to describe some software development methodologies. This paper does not provide a judgment on the open source approach, but exposes the fact that simply stating that a project is open source does not provide a precise description of the approach used to support the project. By taking a multi-disciplinary point of view, we propose a collection of characteristics that are common, as well as some that vary among open source projects. The set of open source characteristics we found can be used as a tick-list both for analysing and for setting up open source projects. Our tick-list also provides a starting point for understanding the many meanings of the term open source.

1 Introduction

We started looking into *Open Source* to try to determine how using this approach actually impacts the dependability of the software systems being developed. Our intention was to spend some minor effort to understand what is meant by the term "open source", and from there perform various studies and experiments to support or to oppose dependability claims in the area. Much to our surprise, understanding what open source is turned out to be a much more complex task. The term "open source" has been widely used to describe a software development process that relies on the contribution of its geographically dispersed developers by the means of the Internet. Amongst other criteria, one basic requirement of open source projects is the availability of its source code [1], without which the development or evolution of the software is very difficult if not impossible. But apart from these characteristics, there seems to be some confusion on what actually makes a project an open source project.

The aim of this paper is therefore to provide a clearer description on what is meant by "open source". To achieve this aim, we investigated several well-known open source projects such as Linux [2], Apache [3] and Mozilla [4]. We also did literature studies on published materials about open source, notably *The Cathedral and the Bazaar* [5], *Rebel Code* [6], *Open Sources* [7] as well as work by other people interested on open source (for example, [8-12]). We have also used several on-line resources dedicated to various open source projects [13, 14] and interviewed both individuals working on open source projects at their free time and individuals involved with open source as part of their job in large corporations. From there, we tried to dissect open source further by determining the characteristics that open source projects should or usually have. We determined a set of characteristics that are almost always present and others that vary among open source projects, and this serves as the core of this work.

The rest of this paper is structured as follows: Section 2 presents a brief history of open source, which is important for understanding its motives and directions; Section 3 describes some open source characteristics that can be used in determining whether a project is or not open source; Section 4 provides some initial conclusions of our work; and Section 5 outlines areas that can be researched further.

2 A Brief History of Open Source

2.1 How it started

The idea of building software within a cooperating community, where the source code was made available so that everyone could modify and redistribute it began with the GNU project at MIT in the early 1980s. The intention was to provide *freedom* relating to software systems. In 1985 the *Free Software Foundation (FSF)* was pioneered by Richard Stallman to generate some income for the free software movement, not restricting itself to GNU.

Free software, as defined by the FSF, is a program that grants various freedoms to its users. A free software program provides its users with [15]:

- Freedom to run the program for any purpose
- Freedom to study and adapt the code for personal use
- Freedom to redistribute copies of the program, either gratis or for a fee
- Freedom to distribute improved or modified versions of the program to the public

The discourse used by the FSF tends to be confrontational and against proprietary (closed) software, since they view anyone producing this kind of software as big obstacles to the four basic freedoms mentioned above. This is reflected in the restrictive viral nature of some of their licenses (see section 3.3).

2.2 Free Software and Open Source Movements

In the early 1998, the term *Open Source* was coined as a response to the announcement made by Netscape on its plan to give away the source code of its web browser. The new term came out of a strategy meeting in which people present realised that:

> *"…it was time to dump the confrontational attitude that has been associated with 'free software' in the past and sell the idea strictly on the same pragmatic, business-case grounds that motivated Netscape." [16]*

Immediately afterwards, the *Open Source Initiative (OSI)* was set up to manage and promote the *Open Source Definition (OSD)*. The OSD was composed as a guideline to determine whether a particular software distribution can be called open source or not. OSD asserts nine criteria that open source software must follow; the main three are:

- The ability to distribute the software freely
- The availability of the source code, and
- The right to create derived works through modification.

The rest of the criteria deals with the licensing issues and spell out the "no discrimination" stance that must be followed [1]. They are:

- The integrity of the author's source code must be preserved, making the source of changes clear to the community
- No discrimination against persons or groups both for providing contributions and for using the software
- No restriction on the purpose of usage of the software, providing no discrimination against fields of endeavour
- The rights attached to the software apply to all recipients of its (re)distribution
- The license must not be specific to a product, but apply to all sub-parts within the licensed product
- The license must not contaminate other software, permitting the distribution of other non-open source software along with open source one

The Open Source and Free Software movements can be compared to two political parties within a community. While two political parties agree on the basic principles but disagree on practical issues, the Open Source and Free Software do exactly the opposite. They disagree on the basic principles (commercialism, licensing, etc.), but agree on (most of) practical recommendations (availability of source code, ability to modify the code, etc.). They even work together on many specific projects to achieve the same goal: to provide software that is free (in terms of liberty) for all [17].

2.3 Commercialisation of Open Source

Open source is often seen as a marketing ploy to make Free Software more attractive to business users since it allows greater liberties with its licenses (see section 3.3). This means that the open source licenses are more accommodating to people or companies to make profit from the software, as long as the source code remains available and can be modified freely.

The most prominent way of commercialising open source is by providing service and distribution packages for software developed in an open source fashion. This is due to the fact that open source software is usually more difficult to install since it was originally aimed for the hacker community. Another way of making money out of open source is by using the relevant open source as a platform, upon which commercial (often proprietary) application software can be built.

More and more computing corporations turn their attention to open source as a business opportunity. What they are looking for in this new development method is *innovation*, and sharing source code is perceived to be a good way for facilitating creativity. Commercial organizations are also attracted to contributing to open source projects as they see a strategic opportunity to undermine (more powerful/dominating) competitors. On the down side, they are afraid that maintaining control of an active open source project can be difficult. They are particularly concerned with the risk of code forking – the evolution of two (or more) separate strands of work from the original code base, which threatens compatibility. This fear prevents some individuals and many companies from active participation in open source developments [7].

Although this code forking risk is always present, it is usually overcome by the novel attitude that the open source community has. Instead of basing their reputation on "what they have", they measure it against "what they give". This "gift-culture" encourages people to contribute more and binds people together in the same strand of work. More information on the "gift-culture" is available from Eric Raymond's paper, *Homesteading the Noosphere* [18].

2.4 The Open Source Approach compared with Others

To provide a clearer picture on where open source (free) software stands in relation to other software, we provide some comparisons (mostly in licensing and distribution terms) among several categories of software. For simplicity, we could say that the two main categories are the "free" software (meaning open source as well) and the "proprietary" software.

There are two kinds of software within the "free" category: *non-copylefted free software* and *copylefted software*. Non-copylefted free software comes from the author with permission to modify and redistribute, and in a legal term it means "not copyrighted". On top of that, it is allowed to add more restrictions to the modified version, which means that some copies (modified versions) may not be free at all. Anyone can compile the program and redistribute the binary as proprietary software. *Public domain software* is a special case of non-copylefted free software. On the other hand, with copylefted software, it is not allowed to have additional restrictions to be added when someone redistributes or modifies the software. As a consequence, every copy of copylefted software, even after modification, must be a free software. The most prominent distribution terms for copylefted software are covered in the *GNU GPL (General Public License).*

Proprietary software is *closed* software in that the source code is not available to the public. It has very restrictive terms on its condition of use, and its redistribution or modification is prohibited. There are two special cases within this group of software: *shareware* and *freeware*. Both allow people to download, use and redistribute the software for free, but modification is (almost) impossible because they are usually released in executable (binary) format only. The difference is on the limit of usage, if someone wants to keep using a shareware, he/she must pay a license fee. One important note is that freeware must not be confused with free software, especially because modification of a freeware is not possible (since the source code is not available).

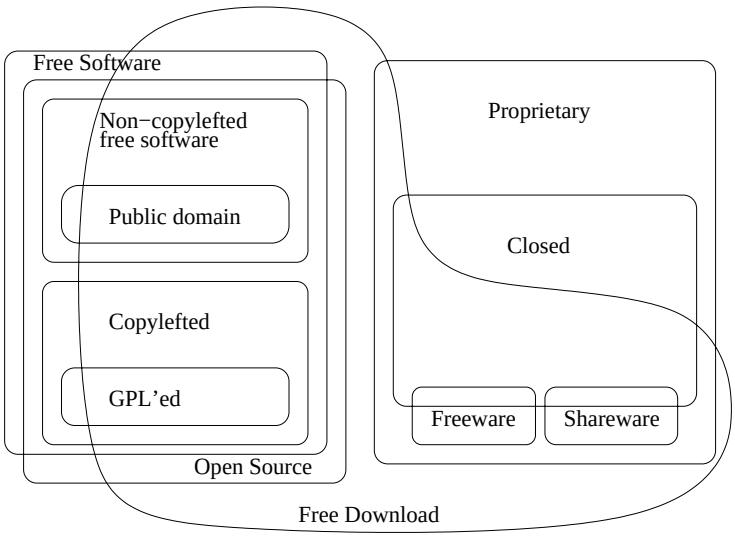


Figure 1: Categories of software

The classification of software in the manner above can be seen diagrammatically as Figure 1, which was adapted from the software categories based on the Free Software Foundation view [19]. Table 1 below summarises the main comparisons between the characteristics of those software categories.

There are subtle differences between open source and free software, in particular around licensing issues. For example, open source software may use proprietary library (e.g. the KDE project [20] was using a proprietary library called Qt until September 2000), which is unacceptable in free software. Further investigation surrounding these differences could provide better understanding, as highlighted in section 5.

Table 1: Comparisons of different kinds of software

	Open Source (Free) Software		**Proprietary Software**		
	Non-copylefted	*Copylefted*	*Closed*	*Shareware*	*Freeware*
Availability of source code	Y	Y	N	N	N
Permission to					
• redistribute	Y	Y	N	Y	Y
• modify	Y	Y	-	-	-
• add restriction	Y	N	N	N	N
Modified version always free	N	Y	-	-	-
Free Download	Y	Y	N	Y	Y
Time Limit in usage	N	N	N	Y	N
Possibility of making money	Y	Y	Y	Y	N

3 Characteristics of Open Source

By exposing the characteristics that open source projects usually have, we hope to be able to develop a clearer picture on what it really means for a particular project or software development to be an open source project[1] or not. The idea is to have a "tick-list" of open source characteristics, against which the characteristics of the project in question can be compared. Additionally, these characteristics highlight the fact that just stating that a project is open source does not necessarily provide a precise definition.

3.1 Disciplines to consider

In the spirit of DIRC[2], a research project that we are working on, it is important to highlight that software development is a very complex process that draws upon knowledge/expertise from many scientific disciplines. Therefore, to understand it better, it is necessary to emphasise its interdisciplinary nature. It appears that open source software development is no exception, and in order to determine the relevant open source characteristics, there are several disciplines that we would like to consider:

- Computing Science
  Covering the technical aspects that need to be considered to engage in an open source project.
- Management Issues
  Dealing with managerial issues and how they relate to open source projects.
- Social Sciences

[1] The term 'project' is used loosely in this paper, as it is doubtful whether OSS projects fulfil the more generic management definition of a unique/novel activity with explicit/finite timescales. Should the use of this term create conflicts of definition, for readers, they can interpret the term 'project' as 'undertakings' or 'initiatives'.

[2] DIRC is a UK EPSRC project based on a Dependable Interdisciplinary Research Collaboration (DIRC) on computer-based systems (see http://www.dirc.org.uk/).

Addressing areas related to the communities involved in open source projects and their behaviour.

- Psychology
  Accounting for the characteristics of the individuals involved in open source projects.
- Organisational Aspects
  Dealing with aspects such as organisational structures.
- Economics
  Looking into economic models that underlie open source projects and/or corporations with respect to their involvement in open source projects.
- Law
  Focusing on legal issues.

Clearly, the OSI definition for the term open source does address legal issues extensively, and encompasses some economic aspects. On the other hand, it hardly touches on computing science areas; it also completely ignores the areas of management, psychology, social sciences and organizational aspects. Furthermore, there is no guarantee that a given project, by simply adhering to the OSI definition of the term open source, benefits from the positive effects that are usually related to the term open source (e.g. being reviewed by many people). The open source software characteristics proposed by Wang and Wang [11] address some technical aspects, and in less depth, legal and managerial aspects.

In our attempt to understand open source, we determined a set of characteristics that occur under that umbrella term, while considering the various disciplines mentioned above. Some characteristics are common to all efforts we were able to investigate, whereas others vary between projects. The set of characteristics we deem relevant for discussing open source are described below, section 3.2 covering those that are common throughout open source projects and section 3.3 addressing those that vary between projects.

3.2 Common characteristics

Open source projects have many common characteristics. All items listed under the OSI definition of open source, OSD (see section 2.2), are the basic requirements for projects to qualify as open source. Moreover, *active* open source projects rely upon several other characteristics. We have identified six characteristics that are present in successful open source projects, these are addressed below.

Community

All active open source projects have a well-defined community with common interests that are either involved in continuously evolving its related products and/or in using its results. Anecdotally, the community, in its vast majority, is composed by men. Communications tend to be constructive, at times becoming confrontational.

Motivation

The biggest question surrounding the open source phenomena is *why do people do it?* What is the explanation behind having people providing contributions for free? The answer to these questions is not as straightforward as one might have thought. There are *different types of contributors*, individuals and corporations. Individuals usually

contribute for personal satisfaction; some have really strong philosophical beliefs others do not care as much about such issues. Corporations usually get involved with the aim to gain market share, undermine their competitors, or simply rely on products generated by open source without having to build a fully equivalent product from scratch.

Peer recognition also plays a role on motivating contributions. By having their contributions recognized as appropriate and of good quality by the community involved, both individuals and corporations have their status raised within the given project. Consequently, their opinions are considered more carefully with respect to project related decisions and their reputation may even improve outside the project boundaries.

Developers' profile

The set of people that contribute code to specific open source projects is always composed of those that are also users of the code produced. This means that open source developers are a subset of the open source user community, i.e. all open source developers are users, but not all users are developers (Figure 3).

This characteristic explains the fact that there are normally no precise specifications or requirements documents clarifying what is to be achieved in the project. It also highlights that it is quite unrealistic to expect the open source community to start developing arbitrary kinds of software. Software developers are usually not expert users of medical systems, nuclear plant control systems, or air traffic control systems.

Process of accepting submissions

An open source project evolves by receiving submissions from various sources to address various aspects of the project. The most common submissions are those of bug reports and source code, others include documentation and test cases. Furthermore, open source projects often post the areas in which they are interested in receiving submissions. As a consequence, multiple concurrent submissions may be received addressing the exact same area. Therefore, open source projects have in place processes for accepting various types of submissions, also making it clear on how to handle multiple concurrent submissions.

The process of accepting submissions is composed of two main parts: the *decision making process* and the *process of disseminating information on submissions*. How these two parts get implemented varies from one open source project to another (see section 3.3).

Development improvement cycles

Product improvement in the open source software development process can manifest in both breakthrough and continuous improvement modes. Breakthrough improvement involves dramatic and relatively impromptu changes [21]. Evidence of this form of product improvement in open source development was provided by Raymond [5] in the development of Fetchmail. He notes that:

> *"The real turning point in the project was when Harry Hochheiser sent me his scratch code for forwarding mail to the client machine's SMTP port…this SMTP-forwarding concept was the biggest single payoff I got…The Cruftiest*

parts of the driver vanished. Configuration got radically simpler…the only way to lose mail vanished…and performance improved." (p. 47-50)

Continuous improvement involves an increased frequency of change but in smaller and more incrementally consolidating stages [21]. This philosophy of product development recognises that small improvements build up to larger improvement overtime, but with the added advantage of being far easier to implement. Incremental product improvement through bug finding and fixing is a development hallmark of the open source paradigm and is embodied in Eric Raymond's original characterisation "release early, release often" [5] The idea is to get quick feedback, which can then be incorporated back into the product.

More recently such anecdotal claims have been further reinforced by the research findings of Aoki et al. with the open source *Jun* project [22]. They tracked the evolution of the software over 360 versions and identified both incremental improvements within single version updates followed by significant functionality increases requiring major modification to the existing architecture. Both of these forms of product improvement are generically shown in Figure 2 below.

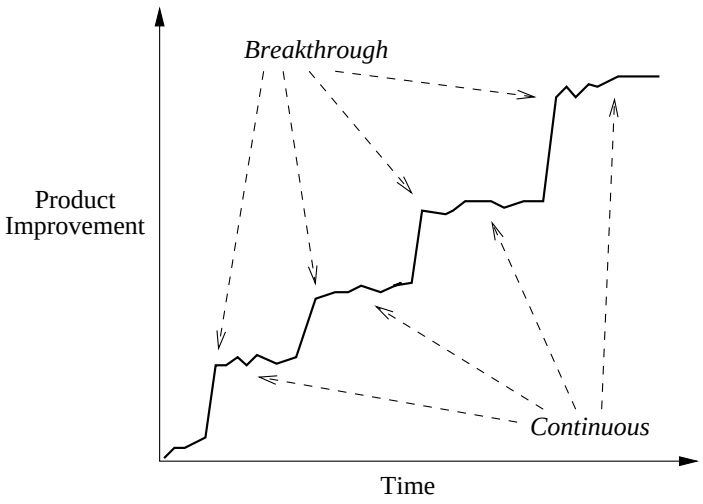


Figure 2: Open source product improvement over time.

Modularity

The benefits of modular design are well established in all engineering disciplines, as it supports increased understanding during design and concurrent allocation of work during implementation [23]. However, due to the globally distributed nature of open source development, well-defined interfaces and modularised source-code are a prerequisite for effective remote collaboration [24].

3.3 Variable characteristics

The areas in which open source projects vary are much more numerous than those that they have in common. Below is a discussion of some of those.

Choice of work area

As previously mentioned, open source projects often request contributions to the areas in which they are interested in receiving submissions. Some open source projects will

process both solicited and spontaneous contributions, whereas other open source projects may be prone to ignoring spontaneous contributions.

Balance of centralisation and decentralisation

The communities within various open source projects are organised differently. Some have a very strict hierarchy differentiating among various levels of developers (see Figure 3), whereas others have a much looser structure. The strict hierarchies bring with them a more centralised power structure, for example, the *core* developers have more power than ordinary (co-) developers in making executive decisions. In some open source projects (e.g. Apache), it is even possible to have more than two levels of developers. But not all open source projects have multi-level developer groups. Looser organisational structures have all their developers on the same level, which implies decentralisation of decisions, at times being based on full consensus for approving decisions.

Meritocratic culture

The basic model underlying open source projects is that knowledge shown by means of contributions increases the perception of merit, which in turn leads to power. Exactly how this transition takes place varies from project to project in terms of timing and the obstacles that must be overcome, and depends on the actual organisational structure of the project. For example, Figure 3 shows the possible transition from passive to active users when they start contributing to the project. If they could then show their ability (or they could gain respect from the community), they might be invited into the developer group, where they would have greater rights over the code (e.g. to incorporate their own modifications into the code base). In some projects, there is also a possibility of promotion from the co-developer to the core developer group. The transitions can also go the other way, e.g. a core developer might wish to resign and become a co-developer instead (or even leave the project completely) due to other commitments or personality clash.

Business model

Depending on the domain that an open source project addresses, different business models may motivate the involvement of commercial corporations, researchers, individual developers and end-users. The business models we have identified so far are: own use, packaging and selling, and platform/foundation for commercial or research software development.

Decision making process

The decision making process relies on four dimensions that vary from open source project to project. These are the *quality goals*, the *acceptance criteria* enacted, the *cognitive abilities* of the decision group, and the *social structure* within the project. Quality goals vary widely from one open source project to another; this can be observed even in the same application area (e.g. one focusing on performance and another on portability). The acceptance criteria used also vary among open source projects. It can be the best solution out of the first *n* submissions, some form of aggregation of multiple submissions (even by requesting that someone changes their solution to add some other aspect seen elsewhere), some memory of previous submissions by the same person, the first submission received, etc. Additionally, the

ability to recognise better solutions is highly dependent on the cognitive abilities of the decision group. This implies that the decision making process on accepting submissions varies among projects and potentially within projects as well, unless the same people are involved in all decisions.

The social structure inherent to an open source project may be a defined hierarchy where different groups of people get to evaluate different submissions (e.g. by focus area) and/or some people exercise greater power, or a monolithic group composed of all developers. The social structure impacts directly on the decision making process. If the group is monolithic then the acceptance of submissions may be achieved by consensus or majority voting. If there is some other form of social structure, the same consensus or majority voting may apply, at times with the votes of some of the members counting more than others.

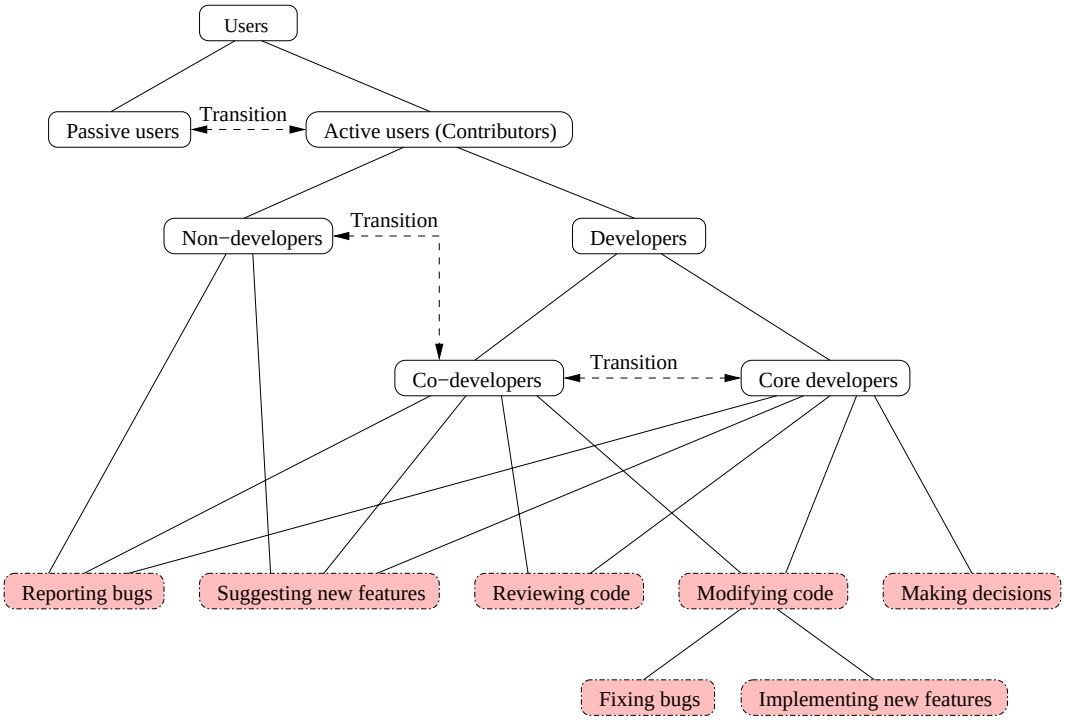


Figure 3: The classification of open source users and developers

Submission information dissemination process

The information on submissions and their acceptance may be passively disseminated by the means of newsgroups or comments in the code itself, it may be actively disseminated by using emails and mailing lists, or there may be some dedicated web space for statistical information.

Project starting points

Open source software projects may start from scratch or from existing closed source software systems, either commercial or research. From the various projects that we studied we could only find examples of projects that transitioned the full package from closed to open source at once. Nevertheless, one can envision some closed source software making a gradual transition to open source, one part (e.g. a subsystem) at a time.

Visibility of software architecture

The software architecture of a computing system depicts its structure(s) and comprises its software components, the externally visible properties of those components, and the relationships among them [25]. The architecture of an open source software system may be itself open or closed. The "closedness" may occur intentionally or accidentally. Having an intentionally closed software architecture means that the core group will consciously not reveal the structure to the general public. An unintentionally closed software architecture suggests that the structure exists in some people's minds only.

Documentation and testing

Documentation and testing are important aspects of the software development process. Good documentation allows people to use – and more specifically in open source projects, to understand and modify – the software. Thorough testing enables the users (and the developers) to have confidence that the software they are using (or developing) is going to function as expected.

These two areas are often overlooked or vary widely in the open source development process. Open source contributors tend to be more interested in coding than documenting or testing. This is probably due to the nature of open source that tries to replace the formal testing process with "many eyeballs" effect in eliminating the bugs. Also, adding comments in the source code is often perceived as sufficient for documentation. There has been some effort in addressing the problem of lack of documentation (e.g. the *Linux Documentation Project* [26] and *Mozilla Developer Documentation* web page [27]), but this is still a rarity for smaller open source projects. We have yet to find some sort of testing strategies for open source projects. They might exist, but implicitly and not open to the outside the project.

Licensing

The basic freedoms of open source software and how they differ from other software distributions were discussed in section 2.1 and 2.4 earlier. Here we consider the main varying features of OSD and FSF qualifying licenses[3]. Whether the software is viral or can become closed (proprietary) reflects the two main varying features of free and open source software.

Table 2 illustrates this with some of the more popular public licenses conforming to the OSD/FSF definitions. Viral licenses ensure that if any of the software code is used in other software developments then this will cause all of the software to come under the terms of that original license. The other varying feature

Table 2: Varying characteristics of open source licenses

Licenses	Is it viral?	Can it be closed?
GPL	Yes	No
LGPL	No	No
BSD	No	Yes
Q Public	No	No
IBM	No	Yes
Netscape (i.e. Mozilla)	No	Yes

[3] The term 'qualifying' refers to the four fundamental freedoms that both the OSD and FSF agree on.

concerns whether the license allows any of the original source code to be distributed in binary form only in future derived software products.

Operational support

In order to facilitate concurrent software development and fast controlled evolution, all open source projects implement some form of configuration management. This is enacted by using CVS, other tools, or even an ad-hoc solution using some web-based support.

The communication within communities related to specific open source projects is done almost exclusively by electronic means, which are also used to organise their work. The electronic means most commonly used are dedicated mailing lists, newsgroups, and web site. The exact structure and usage of web sites, mailing lists and newsgroups vary among open source projects.

Size

Size is not a distinctive measure in open source projects. Both involved-community and code base sizes vary widely from project to project.

4 Conclusion

The term open source is being used within the computing science community at large in a vague manner, consequently creating confusion and misunderstandings. In our efforts to understand open source we have done an extensive literature review, explored several web sites related to the topic, and interviewed some individuals and

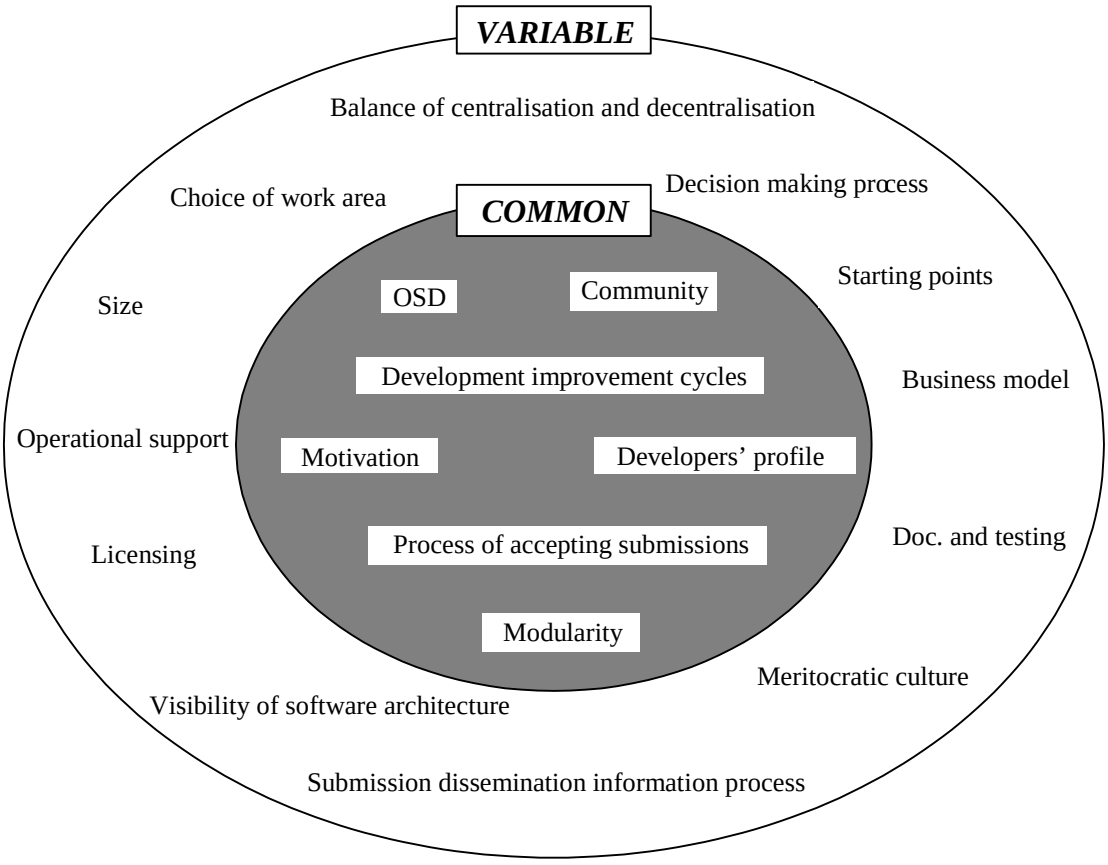


Figure 4: Open source characteristics – common and variable

corporations involved with open source. Our work was performed bearing multiple disciplines in mind.

We have determined many project characteristics that are relevant for open source. Some of these characteristics are common to all efforts, whereas others vary among open source projects (Figure 4).

How the various characteristics relate to the disciplines discussed in section 3.1 is highlighted in Table 3.

The set of open source characteristics we found can be used as a tick-list both for analysing and for setting up open source projects. We understand that there is no way that an absolute tick-list can ever be generated due to the variations that exist from one open source project to another, so additional variable characteristics may exist. Our proposed tick-list provides a starting point for understanding open source and its many meanings.

Table 3: Open source characteristics and disciplines considered

	Computing Science	Management Issues	Social Sciences	Psychology	Organizational Aspects	Economics	Law
OSD	√					√	√
Community			√	√			
Motivation		√	√	√		√	
Developers' profile	√		√		√		
Process of accepting submissions	√	√			√		
Development improvement cycles	√	√	√				
Modularity	√	√		√			
Choice of work area	√	√		√			
Balance of centralisation and decentralisation		√			√		
Meritocratic culture			√		√		
Business model						√	
Decision making process	√	√	√	√			
Submission information dissemination process		√	√				
Project starting points	√	√				√	
Visibility of software architecture	√	√	√	√	√		
Documentation and testing	√	√	√				
Licensing						√	√
Operational support	√	√	√		√		
Size	√		√		√	√	

Our work has led us to understand that it would be unreasonable to try to discuss open source software in general. There are as many differences among open source software projects as among non-open source software projects. Furthermore, many of the characteristics present in open source software projects are not restricted to open source software environments, they may also be found in some proprietary environments. Simply using the term open source is not enough, just as using the term proprietary software does not suffice.

Consequently, discussions comparing software project processes and approaches ought to occur at a lower level of granularity, at the individual characteristics level, in order to be fruitful. Whether projects are more or less successful, or exhibit a lower or higher expected quality, depends on the characteristics of the development and maintenance environment that they are in.

5 Future Work

There are many issues still left to be investigated with respect to understanding and exploiting the open source approach. Future work should further clarify the exact differences between open source and free software, as well as generate a table relating various existing open source and free software projects to the characteristics we set forth, while describing how each of these projects implement the variable parts.

There are also *dependability* issues that need to be addressed. We shall be looking into statistical information, such as bug density, fixing time, hacking incidents, etc., regarding open source software, free software, and proprietary software. This shall be done by grouping software packages according to their individual characteristics, rather than by grouping them under the labels that we have just used above (open source, free and proprietary software), with the aim of determining which openness characteristics foster more dependable systems or not.

6 Acknowledgements

This paper has been funded by the UK EPSRC project on Dependable Interdisciplinary Research Collaboration (DIRC – http://www.dirc.org.uk/). We would like to thank the volunteers – in particular, Julian Coleman, Stuart Wheater, and Mike Ellison – that spent their time while sharing their experiences with us. We would also like to thank our colleagues from the DIRC project involved in the Open Source activity for various fruitful discussions contributing towards this paper.

7 References

[1] "The Open Source Initiative: Open Source Definition", http://www.opensource.org/docs/definition.html.

[2] "The Linux Home Page at Linux Online", http://www.linux.org/.

[3] "The Apache Software Foundation", http://www.apache.org.

[4] "mozilla.org", http://www.mozilla.org/.

[5] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates, 1999.

[6] G. Moody, *Rebel Code: Linux and the Open Source Revolution*, The Penguin Press, 2001.

[7] C. Dibona, M. Stone, and S. Ockman, *Open Sources: Voices from the Open Source Revolution*, O'Reilly & Associates, 1999.

[8] A. Mockus, R. T. Fielding, and J. Herbsleb, “A Case Study of Open Source Software Development: The Apache Server,” Proceedings of ICSE 2000, pp. 263-272, 2000.
[9] M. W. Godfrey and Q. Tu, “Evolution in Open Source Software: A Case Study,” Proceedings of International Conference on Software Maintenance (ICSM'00), 2000.
[10] B. J. Dempsey, D. Weiss, P. Jones, and J. Greenberg, “A Quantitative Profile of a Community of Open Source Linux Developers”, SILS TR-1999-05, 1999.
[11] H. Wang and C. Wang, “Open Source Software Adoption: A Status Report,” *IEEE Software*, March/April, pp. 90-95, 2001.
[12] J. Feller and B. Fitzgerald, “A framework analysis of the open source software development paradigm,” Proceedings of 21st International Conference on Information Systems, pp. 58-69, 2000.
[13] “SourceForge”, http://sourceforge.net/.
[14] “Geocrawler”, http://www.geocrawler.org/.
[15] “The Free Software Definition - GNU Project - Free Software Foundation (FSF)”, http://www.fsf.org/philosophy/free-sw.html.
[16] “The Open Source Initiatiative: History of the OSI”, http://opensource.org/docs/history.html.
[17] “Why Free Software is better than Open Source”, http://gnu.metagensoft.com/philosophy/free-software-for-freedom.html.
[18] E. S. Raymond, “Homesteading the Noosphere”, http://tuxedo.org/~esr/writings/homesteading/homesteading/.
[19] “Categories of Free and Non-Free Software”, http://www.gnu.org/philosophy/categories.html.
[20] “K Desktop Environment Home”, http://www.kde.org/.
[21] N. Slack, S. Chambers, C. Harland, A. Harrison, and R. Johnston, *Operations Management*, 2nd ed, Financial Times Pitman Publishing Series, 1998.
[22] A. Aoki, K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A.Takashima, and Y. Yamamoto, “A Case Study of the Evolution of Jun: an Object-Oriented Open-Source 3D Multimedia Library,” Proceedings of 23rd ICSE Conference, Toronto, Canada, pp. 524-532, 2001.
[23] R. N. Britcher, *The Limits of Software: People, Projects, and Perspectives*, Addison Wesley, 1999.
[24] T. Bollinger, R. Nelson, K. M. Self, and S. J. Turnbull, “Open-Source Methods: Peering Through the Clutter,” *IEEE Software*, July/August, pp. 8-11, 1999.
[25] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison Wesley, 1998.
[26] “Linux Documentation Project”, http://www.linuxdoc.org.
[27] “Mozilla Developer Documentation”, http://www.mozilla.org/docs/.

Rebel Code? The open source 'code' of work.

Adrian Mackenzie[1], Phillipe Rouchy[2] and Mark Rouncefield[1]

[1]Department of Computing, Lancaster University
[2]Blekinge Institute of Technology

Abstract:
This paper is concerned with understanding the character of open source (OSS) project work. Using data from interviews, email communications and the code itself we describe how the orderliness of such projects is achieved in contrast, perhaps, to stereotypical views of 'hacker' projects. We use this data to explicate the ways in which OSS projects are accomplished and the ways in which the various participants observably, reportably, accountably orient their actions, their contributions to the project as a whole. The contrast we draw is not with idealised views on software projects but with ethnographic and ethnomethodological studies of software production that emphasise the project as a practical, ongoing achievement. The paper moves beyond idealised versions of the open source project towards understanding OSS as a practical sociological phenomenon.

· Introduction:

As society's dependence on computer-based systems increases, the systems themselves become ever more complex and achieving dependability in these systems, and demonstrating this achievement in a rigorous and convincing manner, is of crucial importance. One of the attractions of Open Source Software Development (OSSD) approaches, at least as suggested by its advocates, comes in terms of the improvements in reliability dependability, and flexibility for the process of software development and the quality of the end product:

> "Open source promotes software reliability and quality by supporting independent peer review and rapid evolution of source code ···Mature open-source code is as bulletproof as software ever gets." (1)*

The OSS approach, characterised as 'massively diverse human scrutiny', or peer-reviewed software, extends the idea of review and introduces a way of confirming final decisions about the inclusion of changes to a system. Examples of open source (e.g. operating systems, development tools, web and mail servers) indicate that a community can be built which can create software that is highly reliable.

However, even studies that might be regarded as broadly supportive of OSS development have pointed to the scarcity of what might be regarded as conventional attributes of orderly software development. Mockus et al (2) for example, use email archives to develop quantitative measures of dependability attributes such as defect density and problem resolution but suggest that "there is no project plan, schedule, or list of deliverables" and that OSS "lacks many of the traditional mechanisms used to coordinate software development, such as plans, system level design, schedules, and defined processes". This links with popular, if fanciful, conceptions of open source software development as the product of 'hacking.' Even when 'hacking' is distinguished from

'cracking' (attempting to breach the security of computer systems), it still often implies unplanned, improvised work. The great attention open source software has attracted in technical and mainstream press fosters that view. It has focused on relatively small bands of highly motivated, even visionary, programmers working in geographical isolation. They seem to be engaged in a brilliantly productive yet free-wheeling production of new software artifacts. Elaborate operating systems and major pieces of software infrastructure (e.g. Apache) seem to flow from their fingertips.

Background

1.Software engineering vs hacking

How different is open source software produced through hacking from the software produced by software engineers? Answering this question decisively is difficult. Therefore, most accounts actually stress differences in engineering process rather than differences in the software itself as a outcome of the process. From the perspective of software engineering, [Vixie, 1999] argues:

> "Open Source developers often succeed for years before the difference between programming and software engineering finally catches up to them, simply because Open Source projects take longer to suffer from the lack of engineering rigor".

Vixie bases his conclusions on a comparison between the formal textbook methods of software engineering (e.g. [Sommerville, 2001]) and what he labels, somewhat derogatively, 'programming'. If professional software engineers are eager to point out the 'lack of rigor' of open source programming, open source programmers have been even quicker to distance themselves from conventional software engineering. The famous hacker, Eric Raymond, writes:

> "What I saw around me was a community which had evolved the most effective software-development method ever *and didn't know it*! That is, an effective practice had evolved as a set of customs, transmitted by imitation and example, without the theory or language to explain why the practice worked" [Raymond, 1999]

The contrast with Vixie's position could not be greater. Instead of an absence of method, Raymond is suggesting that the development practices involved in open source software was so radically new that there was no way to explain it.

We propose that much shared ground runs between these two diametrically opposed positions. However, this shared ground is not highly visible. Instead of a deficiency in methodical rigor (Vixie) or an effectiveness almost too novel to be explained (Raymond), we would like to describe a habitually ignored middle-ground between the highly formalized vision of software engineering and the myth of collective improvisation. There are important continuities between the two types of activity which neither account recognise. A scarcely visible infrastructure of practices and contrivances is woven through both open source and professional software engineering. The analysis of a significant case study, the Cocoon project (http://xml.apache.org/cocoon), will allow us to show how open source projects are grafted onto practices developed in software engineering.

2.Making things orderly

Following Button & Sharrock (1996), we suggest that the continuities and some important differences between OSS and conventional software projects consist in the *ordering practices* commonly found in open source software projects.

> ".. much effort is expanded on contriving devices which will provide 'orderliness' in the conduct of work and in ensuring that such devices can be implemented and enforced. These devices are meant to enable the achievement of orderly work where it requires the collaborative participation of many individuals, and may, crudely, be characterised as devices which are designed to create and support teamwork". (1996: 373**)**

These practices are intricate and fine-grained, and, as we will show in the case of the Cocoon project, criss-cross every level of project work, ranging from end-user documents down to source coding. Button and Sharrock also highlight the importance of 'the project':

> It is commonplace to refer to engineering projects and the easy way in which this term is used can detract from the recognition that the project is a prominent way in which engineering work is socially organised so as to confront the sorts of contingencies that face software engineering that we have alluded to such as the threatened curtailment because of, for example, drastic slippage, or such as the pressures to abandon good practice." (Button & Sharrock, 1996, 372)

While the contingencies may be different, the notion of the project retains strong relevance to open source software. They too involve social and technical organisation, albeit now directed towards the contingencies of geographical dispersion, fluctuating teams of participants, and open-ended timelines.

1.Method - studying Cocoon as geographically dispersed, fluctuating and open-ended project

Our research focuses on an OSS project, 'Cocoon,' which itself is related to the much larger and more well-known Apache OSS project. Cocoon is described by its originator as:

> " .. a software project that I started to "ease" the task of writing documentation, creating tools that allowed publishing to be easier and more specific for their needs." (email from Steffano Mazzochi)

but describes itself as:

> ".. a 100% pure Java publishing framework that relies on new W3C technologies (such as XML and XSL) to provide web content. The Cocoon project aims to change the way web information is created, rendered and delivered. This new paradigm is based on the fact that document content, style and logic are often created by different individuals or working groups. Cocoon aims to a complete separation of the three layers, allowing the three layers to be independently designed, created and managed, reducing management overhead, increasing work reuse and reducing time to market."(Cocoon website)

The goal that Cocoon is setting itself is to refine the management of web pages at every stage ranging from creation to maintenance. It is a device that seeks to co-ordinate the collaborative work involved in web-sites. As a mode of ordering a certain kind of documentary work, Cocoon conforms to what Button and Sharrock describe as an ordering device. Cocoon as a device is designed to create and support teamwork in the domain of the creation of web pages. It focuses on ordering the conduct of work so that 'management overhead' is reduced.

Our observations are drawn from interview, source code and email archive data. We use this data to explicate the ways in which OSS projects are accomplished and the ways in which the various participants observably, reportably, accountably orient their actions, their contributions, their emails to the project and to notions concerning 'good' or 'elegant' code, ideas about 'ownership' and so on. Much in the way that Wieder (8) describes the 'convict code' - as a resource that is drawn upon to account for and understand action rather than a simple normative stipulation and explanation for behaviour - so the OSS Cocoon community uses sets of ideas about coding, about participating in an OSS project and so on as resources for their accounting practices in the course of contributing to the project itself.

Our interest is primarily in understanding the character of the OSS project as a 'project' - how it actually 'gets done'. The contrast we draw is not with idealised views on software projects or open source development but with ethnographic and ethnomethodological studies of 'realworld, real time' software production (Button and Sharrock 1996). These emphasise the project as a practical, ongoing achievement and concentrate on the everyday, mundane aspects of keeping a project going. We place particular emphasis on various kinds of 'ordering work' that occurs at a number of levels throughout the project and draw attention to the set-up of the website, coding, email correspondence and the project archive. As an instantiation of 'virtual teamwork' Cocoon as a project necessarily needs to attach considerable importance to issues of distributed coordination; plans and procedures; and developing an 'awareness of work'. The concept of the 'virtual team' (Zimmerman, 1997; Siebel 1998) is intended to denote an organisational form consisting of networks of workers and organisational units, linked by information and communication technologies, that flexibly co-ordinates activities, skills and resources to achieve common goals without traditional hierarchical modes of central direction or supervision. Such teamwork, 'less fettered by the constraints of traditional hierarchies and spheres of responsibility, engenders a heightened sense of empowerment, commitment and collective responsibility' (Casey, 1995: 45). Whilst with conventional software projects understanding the organisational context is vital - "software engineering is often carried out within an organisational environment which threatens to overwhelm the project"(Button and Sharrock, 1996) - with OSS the position is more complicated. While the OSS may be likened to a 'virtual organisation' there are manifestly real problems both connected to the organisation within which the code contributor ordinarily works (for example in time constraints), and within the virtual team itself to do with communication and awareness.

2.Achieving the orderly character of OSS project work

In their salutary paper on the organisation of collaborative design and development in software engineering, Button and Sharrock (1996) point to 'the project' as a formatted organisational arrangement within which software engineers typically coordinate their design and development work and make their work mutually and organisationally

accountable. They carefully document how engineers achieve the formatted arrangements of the project and how they display an orientation to these arrangements in the way they order and accomplish their work. In project work the organisation of the work itself can be a source of troubles that is accommodated through the organisation and re-organisation of work. Ordering work as a project does not in itself ensure the orderliness of work or provide remedies for all contingencies, instead the project structure and plan is an achievement of everyday work and a response to and recognition of the contingent nature of such work. In these circumstances a number of devices are noticeable for ensuring the orderly character of work. 'Phasing' ensures that necessary tasks are adequately completed and provides for the interdependence of activities and the recognition of uncompleted stages. The 'methodic handling of tasks' provides for some kind of system in the confrontation and elimination of problems. 'Orienting to the project as a totality' provides a method for project teams to keep each other's progress in view and make it visible to others. 'Measured progression' refers to procedures and devices - organisational metrics - for documenting how much of the project has been done and what remains; checking work against schedules and so on. Finally they note how 'making sure the documentation gets done' is regarded as 'dirty work' not an integral part of job and superfluous to engineers practical needs.

1.The website as an ordering device.

Quite clearly the Cocoon website can be viewed as an ordering device orienting both 'newbies' and established project members to features of the project through devices such as the menu-bar(Fig.1), the 'to do list, requests for help (Fig 2.), advice for contributors (Fig 3) and so on. The advice on making a contribution for example describes a number of stages through which a 'typical contribution' may go and how any contribution is treated once submitted. The 'to do' list prioritises requirements for code, documentation, samples and design from 'high' (Fig 4) - "upgrade Turbine-pool" - to low and a 'wish' list. The list also assigns particular tasks to named individuals.

Fig 1.

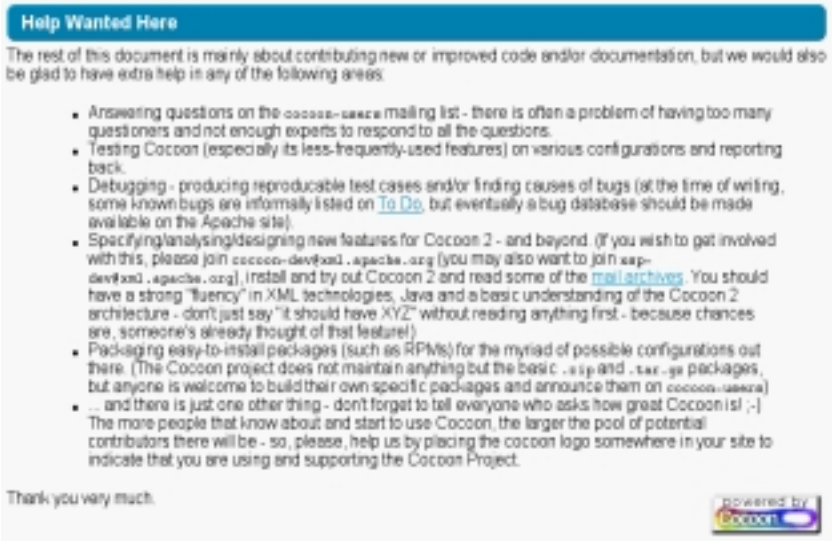


Fig 2.

One way of understanding the working of the website, as effectively the 'desktop' or 'front-office' of the project, is in terms of 'affordances' of knowledge (Anderson and Sharrock (1993). The website provides for project members knowledge of the state of the project, where they are up to what needs to be done etc - and it was evidently designed with this possibility in mind. The website is both the public focus for work and a visible, a publicly available, record of work that has been done or remains to be done. In other words, what these representations do, among other things, is make the work 'visible' so that it can be 'taken note of', 'reviewed', 'queried', and so on, by others involved. They put the work on display so that others may be aware of it.

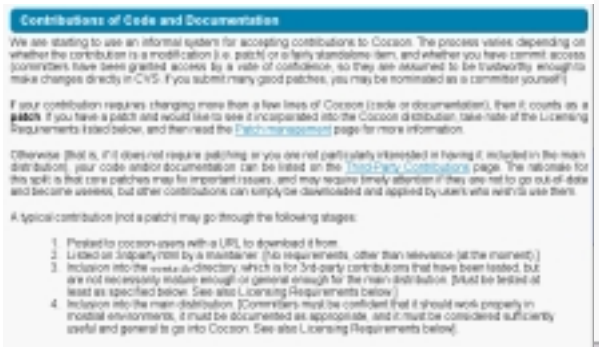


Fig 3.

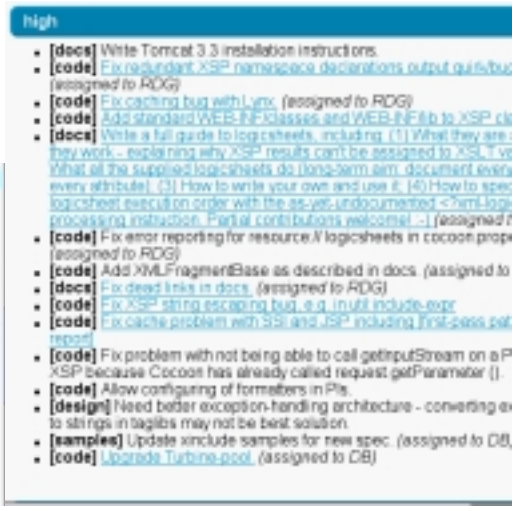


Fig 4.

Our interest is how the different features of the website are constructed so as to 'afford knowledge' as to the working division of labour by which the various tasks on Cocoon are performed. The notion of affordance used here treats perception as resolutely embedded in particular cultural practices. Just being fully enculturated members of the Cocoon project means being able to use website and associated email system, to see unproblematically what needs to be done urgently, what is less important, what the next phase of the project is and what progress they are making. The website (and the email system) provides the project team with the means to see at a glance, and recognise immediately what is going on in the project. The website thereby also acts as a 'technology of accountability' (Suchman 1994; Bowers, Button and Sharrock 1995) enabling members to see, at a glance, the status of the project and calculate whereabouts they might be in the organisational and temporal cycle of events.

2.Order by email: Finding order in the archive

Examination of the email archive is also instructive of the various ways by which order is accomplished in an open source project. Although the richness (and occasional vehemence) of the exchanges is difficult to adequately captured here what is evident is the way in which email communication provides for the administration, voting and scheduling of the project as well as orienting to the project as a whole. Despite the opinion that ""scheduling" is ultimately impossible: we are talking about volunteers that spend their free time. How can you tell when you'll finish planting your garden? or when you'll finish your WW2 tank model? When you do it. Period" (email correspondence) it is clear that a lot of communication through email is about the scheduling of activity. Thus:

> " Okay, how about this for a schedule: (too formal, I know!) If anyone wants to change it, better make it quick!
>
> * I'll commit what I've done so far on the FAQ tomorrow (Saturday), plus some other minor changes. ..
>
> * Feature freeze 00:00 GMT (not BST) Monday - i.e. no new features, only minor bugfixes and doc improvements

> * Around the same time I'll send emails to cocoon-users and cocoon-dev asking for testers to download from CVS and test, and report back what configuration they have and whether there were any problems."

Scheduling is also affected by the 'lazy consensus' system of voting as the following email makes clear:

"Are we all agreed to implement content aggregation in the way specified by Stefano in his RT? I've been pondering it for the last few weeks and playing some thought experiments, and I'm definitely +1. How about the rest of you?"

What also comes over is an orientation to the Cocoon project as a whole both in terms of the management and administration of the project as well as some notion of a 'code' or orientation to the ethos of open source in general. For example the following email:

> "I got alittle issue here. I voted -1 on the engine synch. patch since I thoughtthat we shouldn't put in a patch that messes around with cocoon that deeply shortly before a new major release. according to the terms of the asf project constitution: my veto is binding unless you convince me otherwise".

Brought this response:

> "So, for the sake of the "dignity" of the Cocoon project overall, including myself, I'd like to get it in a more shipshape condition. Now you may say that's about image and PR and marketing etc. which we are things we should stop getting hung up about - but it's not just image, it's about code quality (and quality of the docs).
>
> While I can see the sense in being cautious just before a release, as a general principle - if I were a complete outsider I expect I'd _still_ think that leaving these known simple bugs in was... odd, for an open source project".

What becomes apparent in these discussions is a clear orientation to the project as a whole rather than a collection of tasks. The email system has become a way of keeping each other's progress in view and making their own progress visible to others through activities such as involving themselves in others activities and tasks through talking them through; and knowing where their worked impacted on others and informing them.

3.The open source 'code' of work.'

Finally, the development and orientation to some notion of an open source code regularly appears in the email discussions on 'good' or 'elegant' code, design philosophy and the principles of open source. Perhaps the best example came in the various responses to the following upset contributor:

> "> removed that unportable (and useless) ASCII art along with (slow) system out (logs >/are there for a reason) and clean up messy code..
>
> I'm sorry that you think my coding is messy, and I would prefer that you tell me first, being its my code..
>
> I would appreciate you all to refer to the author of the code first before spreading bullshit.."

That brought the following reply (amongst many):

> "I think it is important to recognise that we are working on an open source project. I know that there are "code ownership" political issues in many companies, but I would sincerely hope that those attitudes would not bleed into this project. Once the code has been committed, it is no longer 'your code' it is 'our code', and we are all committed to making that code as good as possible. It's one of the strengths of open source."

Without necessarily following Edwards (2000) suggestion of 'epistemic communities' what comes over in this email exchange - too lengthy to fully document here - is the outline of some idea of a 'code' that 'governs' open source. This is depicted in even more detail in accounts of the 'hacker ethic' and is used to provide some kind of explanatory account - 'why hackers do it'. In these approaches compliance to the 'code' explains behaviour. The open source community seen as governed by set of rules. Our argument is rather different since we are not interested in offering explanatory or motivational accounts of open source but of understanding how these projects 'get done'. We are interested in examining how the OSS community both construct and make use of the code in the course of their mundane interactions where the code is used by parties to the interaction as displays, or accounts of what those actions are. Orienting to the code in an email can be used for changing topic, defending or defeating a proposed course of action and for accounting for one's actions in an acceptable way. As Wieder comments (on a very different kind of code):

> "The code then, is much more a method of moral persuasion and justification than it is a substantive account of an organized way of life.' Wieder: 175.

3.Working the 'code'

4.Ordering devices at work in the source code

Can we find ordering devices embedded in the source code itself? The availability of the source code is one of the most salient attributes of the open source phenomena, yet in some ways it is also the most difficult kind of observational data that an ethnographic study has to deal with. The reasons for this are complex. While as ethnographers, we can read the source code for open source projects (something that is more difficult to negotiate in conventional industry software projects), by virtue of its formality and rule-governed nature, source code tends to hide the traces of its own development. Similarly, because open source projects are de-centralised and nearly always involve multiple sites, tracking the interactions between different kinds of reading and writing the code can be difficult.

Presumably the code itself should bear the marks of the ordering devices since they afford orderliness in the conduct of work. They allow a *project* to take place, even if its documents, its 'deliverables' and the relations between developers and users looks quite different to that envisaged by accounts of fully-equipped software engineering projects. The practices and ordering devices surrounding open source code are concerned with regulating how it is read, and channeling how it is written and re-written. Open source programmers are often encouraged to read the code, as well as reading the documentation that accompanies the code. In this domain, we expect to find ordering devices concerned with reading and writing code.

Some source code for a part of the Cocoon system is shown below. This code defines a part of the system that manages the caching of web-pages processed by the Cocoon framework. It helps the system decide whether a particular item (such as a web page, or an xml file) should be kept in system memory ready for another page request, or shunted back onto secondary storage, such as a hard disk, because it is not being frequently requested.

```
package org.apache.cocoon.store;

import java.io.*;
import java.util.*;
import org.apache.cocoon.framework.*;

/**
 * This class implements a memory-managed hashtable wrapper that uses
 * a weighted mix of LRU and LFU to keep track of object importance.
 *
 * NOTE: this class is _HIGHLY_ un-optimized and this class is _CRITICAL_
 * for a fast performance of the whole system. So, if you find any better
 * way to implement this class (clever data models, smart update algorithms,
 * etc...), please, consider patching this implementation or
 * sending a note about a method to do it.
 *
 * @author <a href="mailto:stefano@apache.org">Stefano Mazzocchi</a>
 * @author <a href="mailto:michel.lehon@outwares.com">Michel Lehon</a>
 * @version $Revision: 1.12 $ $Date: 2000/05/16 21:11:51 $
 */

public class MemoryStore implements Store, Status, Configurable, Runnable {
    /**
     * Indicates how much memory should be left free in the JVM for
     * normal operation.
     */
    private int freememory;

    /**
     * Indicates how big the heap size can grow to before the cleanup thread kicks in.
     * The default value is based on the default maximum heap size of 64Mb.
     */
    private int heapsize;

…

    class Container {
        public Object object;
        public long time = 0;
        public int count = 0;

        public Container(Object object) {
            this.object = object;
```

Some of the ordering devices present in this code are common in software engineering today. Some are somewhat specific to open source style projects. A combination rderings are present in this example. This particular code shows evidence of being ordered for at least two distinct kinds of readers.

Firstly, it affords reading by programmers and developers. Their ‘reading’ is associated with modifying the program. What is known to programmers as ‘code style’ - the restricted line lengths, the use of nested indentation to represent something about the flow of execution of the program, and the use of blank space to show separations between different components of the code – allows people reading the program on screen to begin to interpret the code as a set of operations and structures. For such readers, particular zones of the text are marked out for different modes of reading. Any line beginning with an asterisk will attract attention as a comment, something programmers particularly addresses to other people or themselves. This may be an explanation, an apology or a request (e.g. “So, if you find any better way to implement this class (clever data models,

smart update algorithms,etc...), please, consider patching this implementation or sending a note about a method to do it."). More than half of the source code text in this example consists of comments. By contrast, any line that begins with a keyword like 'class', 'public' or 'private' will stand out to a programmer since it signals an important boundary in the organisation of the program. Reading these lines involves separating out keywords, operators and syntax marks from the proper names that the programmer(s) have used to designate elements of the program. Words such as 'freememory' or 'getStatus' describe designate places where an important values is stored, or places where significant operations will be specified. On these lines, the reader is alerted that they must read the code as naming something specific to this program.

Secondly the source code affords processing by other programs such as compilers, document generators (e.g. javadoc will read certain lines), and configuration management systems (e.g. Concurrent Versioning System, CVS). Programming languages constitute large scale and complex ordering device for people to working with information systems. This is both a trivial and significant point. Programmers assume that by virtue of such things as the termination of lines by semicolons, the use of brackets of various kinds - {}, [], and (), - and the presence of keywords such as 'public' and 'class', the compiler will be able to parse the source code file into an executable file containing instructions that can for the Java Virtual Machine. Programmers compile source code so frequently that it becomes just a routine habit. Yet source code is a formal representation (governed by rules operating on a 'vocabulary' of written characters) that can be directly processed by other programs *and* by programmers. The trivial point that programs are read/written by programmers and read by compilers covers the crucial point that the programmer and the compiler must share the same set of rules. (By contrast, other types of computer files such as an MPEG video file will only ever be 'read' by programs.) OSSD, as a collective activity, only makes sense because of this shared set of rules.

Do these ways of ordering source code indicate anything specific about OSSD? The formatting of the code, the use of Java, and the method of documenting the source (using javadoc) are all textbook academic or industry standard. Almost identically formatted and commented code can be found on any Java-related industry web-site, or in any Java programming textbook. At the level of the reading and writing practices carried out by programmers using text editors or integrated development environments, the ruling conventions in this open source project come from well beyond the domain of open source software projects. There is no evidence of a specific style of coding.

Some minor differences show that this code belongs to an open source project, although even these are somewhat ambiguous data. Firstly, there is a request to anonymous readers to contribute a better algorithm or data structure for part of the system that is said to be '_CRITICAL for a fast performance of the whole system. … please consider patching this implementation.' It is unlikely that a critical component of a professional software system would publicly acknowledge that it is '_HIGHLY un-optimized.' The source code itself, as well as the API documents, solicit contributions and involvement in developing the software. Secondly, the authors' email addresses are provided suggesting the possibility of responding to the source code itself. Again, making source code available for reading is linked to providing an address for responses arising from that reading. The

Cocoon project keeps going only so long as it manages to enroll contributors who are prepared to read and amend the source code and other documents.

In any case, the real significance of the source code lies elsewhere. The theme running through our study has been the clustering of ordering devices around OSS projects. Our argument has been that as ordering devices, none of them are entirely novel. It would very strange if the source code was an exception to this. On the contrary, perhaps we could regard the very familiarity and readability of the source code as an important affordance in OSS.

5.Where are the differences between OSS and professional software development?

One final ordering device confirms this point. Almost every open source software development project currently active, including Cocoon, makes use of a single important ordering device, a program called 'cvs', Concurrent Versioning System. This device is profoundly enabling for open source development in several respects. Itself the product of an open source project (http://www.cvshome.org), the CVS program makes it possible forl arge number of people to access, read and write copies of the same source code files, and amalgamate the results. CVS's developers claim both that "its client-server access method lets developers access the latest code from anywhere there's an Internet connection" and that "its unreserved check-out model to version control avoids artificial conflicts common with the exclusive check-out model." It is so crucial that certain institutions in the open source arena, such as SourceForge (www.sourceforge.net), a large website that hosts thousands of open source projects, can basically be interpreted as a public interface or 'frontend' to a vast CVS repository.

In the source code quoted above, revision numbers show on how many occasions a source code file has been modified. For instance, revision 1.12 implies that this file has been edited at least 12 times, although it may have been read many times before. The symbols shown in a line we have already cited are relevant here: * @version $Revision: 1.12 $ $Date: 2000/05/16 21:11:51 $. The $ characters are put there by another reading and writing device, the versioning system, in this case CVS.

Talk about the status of the CVS repository is a major feature of the email communication amongst developers. Many email messages describe events in the CVS repository. For instance, in describing a milestone release of Cocoon, the developer responsible writes to the developer list:

> On Wed, 6 Jun 2001 22:23:06 +0200 (CEST), giacomo <giacomo@apache.org> wrote:
>
> > > *Now the CVS stuff:*
> > > *- I tagged the beta with cocoon_20_b1*
> > > *- I checked in the build.xml with the new version 2.1-dev*
> > > *- I made a branch of cocoon_20_b1 with the name cocoon_20*

> > > *- I checked in the build.xml with the new version 2.0b1-dev under*

> > > *the branch cocoon_20_branch.*

> > > *So the HEAD is the 2.1 version and the 2.0 is a branch.*

The developer describes in detail the operations that had to be carried out so that the software source was named in an orderly way, and accessible to other readers and writers of the code. The developer' s descriptions of their actions within CVS render the contents of the archive manageable for other developers. If for instance, a particular ' build' or version of the project does not have a commonly agreed upon name, then the team of developers cannot reliably synchronize their editing of the source code. Agreeing on what the name of the version will be is sometimes not enough. It may still leave open the question of where in the CVS repository further changes will take place. Another developer replies to the preceding message:

> > *Yes, that good. I assume all the new development will happen only on*

> > *the HEAD and bug fixes will be applied to both HEAD and 2.0b1-dev*

· > *branch. Is this the common understanding?*

Again, negotiations around how source code will be named, stored and retrieved are taking place here, but in this case about future changes to the code. Without these negotiations about how to name the place where changed code is to be stored, the project would start to fragment. Rather than look for some OSS specificity in the code itself, we can regard the practices of open source as taking existing coding practices and configuration management techniques into a different form of organization.

Open Source and 'Epistemic Communities' - " IF ANYONE THINKS MY CODE SUCKS...TELL ME IT SUCKS...BUT TELL ME WHY".

This paper uses a study of the Cocoon Open Source project to explicate some of the ways in which Open Source projects are accomplished and how participants observably, reportably, accountably orient to the project as a whole. Of particular interest is an examination of how some sense of the Open Source 'community' manifests itself in and through the project. This requires moving beyond idealised versions of Open Source - as exemplified in the "Hacker Ethic" (Himanen 2001) or "Rebel Code" (Moody 2001) - towards understanding open source development as a sociological phenomenon. Our analysis suggests we transcend the simplistic motivational or economic approaches that characterise much of the debate and move toward a praxiological understanding of open source. Such an approach focuses on some of the practical ways in which a project is developed and sustained and 'codes of practice' are routinely displayed, achieved and maintained as features of everyday work.

In particular we are interested in the notion of the Open Source community as an 'epistemic community' (Edwards 2001). Edwards (2001) argues that the notion of epistemic communities provides an understanding of how open source software develops under difficult circumstances; and that the four characteristics of an epistemic community; shared normative and causal beliefs, notions of validity and common policy enterprise; provide some insight into their working. It is evident that the development and

orientation to some philosophy or notion of an open source 'code' regularly appears in Open Source discussions on 'good' or 'elegant' code, design philosophy and the principles of open source. This is depicted in detail in sociological accounts of the 'hacker ethic' where compliance to the 'code' is often used to provide some kind of explanatory account of behaviour. The Open Source community is seen as being governed by a publicly available and documented set of norms.

Edwards (2001) suggests that OSS can be characterised as an 'epistemic community', consisting of participants from various disciplines with various previous experience, but having "the following four characteristics:

- A shared set of normative and principled beliefs, providing a value based rationale for the social action of community members;
- Shared causal beliefs, which are derived from their analysis of practices leading or contributing to a central set of problems in their domain and serving as the basis for elucidating the multiple linkages between possible policy actions and desired outcomes;
- Shared notions of validity – that is, intersubjective, internally defined criteria for weighing and validating knowledge in the domain of their expertise;
- A common policy enterprise – that is, a set of common practices associated with a set of problems to which their competence is directed, presumably out of the conviction that human welfare will be enhanced as a consequence."

Outline of the 'code' that 'governs' OSS - depicted in even more detail in accounts of the 'hacker ethic' etc are used to provide some kind of explanatory account in that behavious is explained by compliance to the code. The OSS community seen as governed by set of normative rules and values. Analysis of the email archive facilitates an examination of some aspects of the OS 'epistemic community'. From numerous examples one will suffice - , a series of emails outline some aspects of the Open Source 'code' relating to what is 'good' code and 'ownership' of code within open source projects. As Edwards suggests this is a central value within the OSS community: " The open source software community in general (not just speaking for a single project) shares a strong disbelief in software patents and closed source software as mechanisms, which restrict the freedom of the users. Closed source software is perceived as barriers that hinder people's free choice and ability to improve software. It is very explicit in the community that sharing code is positive and that contributing code to a project is a way of getting status within the community."

The first email is a comment from a commiter relating what changes have been implemented:

Simplified Extract 1.

Modified: src/org/apache/cocoon Tag: xml-cocoon2 Notification.java
Notifier.java

Log:

removed that unportable (and useless) ASCII art along with (slow) system.out (logs \ are there for a reason) and cleanup the messy code

This evoked the following, hurt, (with appropriate emoticons) response:

Simplified Extract 2.
I'm sorry that you think my coding is messy, and I would prefer that you tell me first, being it my code, if it's not too much of a fuss >:-(
Anyway, why is it messy? Would you like it if I go round saying _your_ code is messy? Don't think you are perfect, we _all_ have to learn.
It is important that I get feedback on my code.
I think these remarks are _unfair_. :-(
The " unportable (and useless) ASCII art " was there to make errors more evident and not pass unnoticed. No problem if it annoys you, I don't care if you take it away, but it was not put there just for fun.
The "(slow) system.out" is rather important to me and to many other programmers IMO, and if it's slow who cares, it should not fire if all is ok. "(logs are there for a reason)"... yes, I KNOW, I already said I knew logging had to be kicked in, and the "(slow) system.out" is there just till C2 gets finished. Alpha 2a? Get real.

I would appreciate you all refer to the author of the code first before spreading _bullshit_ because:
1. nobody is perfect, this is for the users and the coders.
2. if you didn't write the code maybe you don't understand it. Some things that seem stupid to you have been thought of.
3. if the author doesn't get notified he cannot improve.
4. it's not nice to refer to other's people code as sloppy, you are not perfect.
5. if the code is not ok, who wrote it is the first one who has to make it ok.

I am astonished by this lack of sensibility.
I hope it will not happen again.
Nobody would like to contribute if this is done behind their backs.
If you don't trust me, tell me.
If you don't like my code tell me.
If you don't think I'm good enough, don't accept my work.

But please, don't make fun of me.

The response that followed contains a number of pointers as to the nature of the 'code' that governed the OS community:

Simplified Extract 3.
I was rather surprised to see a response like this to a simple code change.

>I'm sorry that you think my coding is messy, and I would prefer that you tell me first, being it '>my code, >if it's not too much of a fuss >:-(
I think it is important to recognize that we are working on an open-source project. I know there are "code ownership" political issues in many companies, but I would sincerely hope that those attitudes would not bleed into this project. Once the code has been committed, it is no longer *your* code...it is *our* code, and we all are committed to making that code as good as possible. It's one of the strengths of open source.
>I would appreciate you all refer to the author of the code first before spreading _bullshit_ >because:
>1. nobody is perfect, this is for the users and the coders.
The users want high-quality code. The coders want to be able to change it. A "code ownership" attitude prevents both.
>3. if the author doesn't get notified he cannot improve.

> You have been notified. You got CVS commit mailings just like everyone else, and instead of taking the opportunity to improve or make inquiries into how to improve or why improvement was necessary, you took offense to it.
> >4. it's not nice to refer to other's people code as sloppy, you are not perfect.
> We could play a touchy-feely dance where we try and get the words right so we don't offend anyone at all, or we can call it like we see it. Personally, I'd rather know that someone thought my code was sloppy instead of never finding out because they don't know how to tell me without offending me. Since the issue has been brought up, I'd like to state the following to all Cocoon developers:
> IF ANYONE THINKS MY CODE SUCKS...TELL ME IT SUCKS...BUT TELL ME WHY.
> >5. if the code is not ok, who wrote it is the first one who has to make it
> ok.
> I completely disagree with this statement. Whoever wrote it, released it to an open-source project. Anyone can (and will) make it ok. I can't make this point strongly enough...there is NO PLACE for code-ownership on an open-source project.

The final email in the series attempted to summarize the lessons learned for the project as a whole. In so doing it indicates and reinforces some of the central working norms of the OS community.

> **Simplified Extract 6.**
> Now that testosteron storms are cleared, I would like you to know that Nicola and I talked a lot on the phone and he apologized for what he now considers a misbehavior.
> I would like to repeat to all of you what I told him:
> 1) nothing to apologize for: everyone of us makes mistakes, sometimes technical, sometimes human. I like people that make mistakes more than people that don't: the first have something to learn, they have a much more interesting life to live.
> 2) development is done by committing first and asking later. This is about release early and often since code is a thousands time more expressive than words. CVS is there *exactly* to know who did what, why and to be able to rollback at any time.
> 4) open source is about working under the eyes of hundreds of the best developers in the world. There is no place where the quality of you work is judged more strictly than in an open source environment, but code is never important, the community is much more.

In Edwards' view the notion of epistemic community and the normative values that bind them together are assumed to explain the whys and wherefores of the actions of the community. The norms, the 'code' is both a descriptive and an explanatory device, providing simple ways of interpreting actions. Our argument is rather different and subtler since we are not interested in offering explanatory or motivational accounts of open source but instead of understanding how these projects 'get done'. Drawing on Wieder's (1974) account of the 'convict code', we are interested in examining how the OS community construct and make use of 'the code' as a resource in the course of their mundane interactions. Here 'the code' is used as displays, or accounts of actions. There is, as Wieder (1974) suggests, another way of understanding the 'code' - not as somehow external to the setting but constitutive of some of the ways in which people act within the setting - ", part of the seamless fabric of action which make up the activities". In this view 'telling the code' - as outlined in the extracts above, is not simply reciting rules but sharing joint actions. This approach is not concerned with producing sociological explanation of behaviour in terms of compliance with code. Instead it is more interested in examining how OSS community both construct and make use of the code in the course of their interaction. The idea of some OSS code that is invoked in the various e-mail discussions is one that enables the OSS community to define and perform actions by making reference to the code. This includes

accounting for their own actions in terms of conformity to the code, and as such it is used by parties to the interaction as displays, accounts of what those actions are. The code can thereby be used for stopping a conversation, defending or defeating a proposed course of action and for accounting for one's actions in an acceptable way. As Wieder writes of the 'convict code': "The code then, is much more a method of moral persuasion and justification than it is a substantive account of an organized way of life.' Wieder: (1974: 175). Thus the phrase " Once the code has been committed, it is no longer *your*code...it is *our* code" formulates what has just happened; provides an account of and a motive for what is said and attempts to direct the email 'conversation' along familiar lines - that there is no place for code-ownership issues in OSS.. As Heritage (1984) argues, this analysis, "vividly demonstrates that where sociological research encounters institutional domains in which values, rules or maxims of conduct are overtly invoked, the identification of these latter will not provide an explanatory terminus for the investigation. Rather their identification will constitute the first step of a study directed at discovering how they are perceivedly exemplified, used, appealed to and contested."

4.Conclusion: Many-eyed bugs: co-ordination amongst the team of readers and writers

Eric Raymond's notorious 'The Cathedral and the Bazaar' argues passionately that open source development substitutes a potentially huge crowd of people for the small number of expert debugging engineers found in conventional modes of software development:

> "No quiet, reverent cathedral-building here -- rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from *anyone*) out of which a coherent and stable system could seemingly emerge only by a succession of miracles". (Raymond, 2000)

This is a very commonly cited difference between professional software engineering and open source development. This contrast is framed in terms of the difference between the monumental 'cathedral' style of software construction associated with traditional software engineering and the buzzing hive of 'bazaar' style of activity out of which open source software emerges. What is less often emphasized is how this contrast could actually work. How has a great babbling bazaar of potential readers and writers of the source code been drawn together? This paper suggests that the 'bazaar' takes place along very specific lines and highly organized lines. It is an achievement that requires a good deal of communication, and the implementation of contrivances that afford certain kinds of reading and writing centred on source code, but also circulating through emails, websites, configuration management systems and release/configuration documents.

In this paper we moved beyond idealised versions of the OSS project - as exemplified in the "Hacker Ethic' or 'Rebel Code' - towards understanding OSS as a sociological phenomenon. Our analysis suggests we transcend the simplistic motivational or incredible economic approaches that characterise much of the debate on OSS. Instead we offer a understanding of OSS in terms of the everyday practicalities of software projects. Standing outside debates about motivation allows us to concentrate on understanding

exactly how and in what ways an open source project is accomplished as practical work in which participants' pressures are usually egological - "what do I do next" - rather than motivational - "what's my motivation here". Such an approach sees OSS less in terms of a lifestyle choice, though this may well be individually important, but instead focuses on some of the practical ways in which a project is developed and sustained and 'codes of practice' are displayed, achieved and maintained as features of everyday work.

5.Acknowledgement

This work is funded by the EPSRC/ESRC Dependability Interdisciplinary Research Collaboration (DIRC).

6.References

- Anderson, R and Sharrock, W. (1993) 'Can Organisations Afford Knowledge?' in Computer Supposted Cooperative Work. Vol1. No. 3. Pp 143-162.
- Button, G and Sharrock, W. (1996) "Project Work: The Organisation of Collaborative Design and Development in Software Engineering" *Computer Supported Cooperative Work: The Journal of Collaborative Computing* 5: pp 369-386,
- Edwards, K. (2001) 'Epistemic Communities, Situated Learning and Open Source Software Development' paper prepared for the 'Epistemic Cultures and the Practice of Interdisciplinarity' Trondheim. Avaliable at http://www.opencontent.org/openpub/
- Himanen, P. (2001) The Hacker Ethic and the Spirit of the Information Age. Random House.
- Mockus, A., Fielding, R, and Herbsleb, J. (2000) 'A case Study of Open Source Software Development: The Apache Server.' in Proceedings of ICSE 2000, Limerick, Ireland.
- Moody, G. (2001) Rebel Code: Linux and the open source revolution. London. Penguin.
- Raymond, E. S. (1999), "The Revenge of the Hackers", " eds. Chris DiBona, Sam Ockman & Mark Stone, *Open Sources: Voices from the Open Source Revolution*, O'Reilly Books, 1st Edition January 1999
- Raymond, E. S. (2000) "The Cathedral and the Bazaar迫 http://tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/
- Vixie, P. (1999) "Software Engineering" eds. Chris DiBona, Sam Ockman & Mark Stone, *Open Sources: Voices from the Open Source Revolution*, O'Reilly Books, 1st Edition January 1999
- Wieder, D.L. (1974) Language and Social Reality, The Hague, Mouton.,

What is in a Bazaar?
A Model of Individual Participation in an Open Source[†] Community

Haggen So, Nigel Thomas and Hossein Zadeh
School of Business Information Technology
RMIT Business
Royal Melbourne Institute of Technology
239 Bourke Street
Melbourne 3000
Australia
haggen@seven.bf.rmit.edu.au, hossein@bf.rmit.edu.au, nigelt@rmit.edu.au

Abstract

The "Open Source approach", which is usually portrayed using the Bazaar metaphor, was criticised as being too simplistic. The model presented here as a contribution designed to represent the relationship between individual developers and an Open Source/Free Software community more completely. This model includes a four-layer (4C) model of the Open Source/Free Software community, the motivations and barriers when a developer decides to join an Open Source/Free Software community, the positive and negative results which occur after interaction with an Open Source/Free Software community. The advantages and disadvantage of the model are examined and its relevancy to the research of dependability is also discussed.

Introduction

The practice of making the source code freely available exists nearly as long as the invention of the computer itself and turning source code into proprietary software and distributing only the compiled binary is a relatively recent concept (Levy, 1984). There were written accounts on the some of the most open systems in history, such as ITS, the Incompatible Time-sharing System (Levy, 1984; Turkle, 1984), but the software development process associated was seldom investigated in depth. Therefore, when

[†] The term "Open Source" in this context actually means Open Source/Free Software. The term "Open Source/Free Software" is used throughout this particular paper except at instances referring to a specific movement in order to show that the authors here maintains a political view that is neutral to both movements in this paper.

Linux started to gain popularity, people searched for words to explain this phenomenon. One of the most highly acclaimed attempts was made by Eric Raymond with his metaphor of the Cathedral and the Bazaar (Raymond, 2000). This metaphor then became the most frequently used explanation for Open Source/Free Software and it had an influential impact on the decision of Netscape to open up the source code of its browser (Moody, 2001; Hamerly, Paquin & Walton, 1999) and started one of the most famous commercial Open Source project, Mozilla.

In Raymond's article of the Cathedral and the Bazaar (Raymond, 2000), he explained how a collective effort of co-developers over the Internet could possibly produce quality software with better reliability and more useful features in a shorter time. However, critics on the Bazaar metaphor suggested that the model provided "too few data points" to construct a picture of the approach (Eunice, 1998a) and extended interpretations to fill the gaps for the cathedral metaphor can sometimes be found in literature. Examples of those are "The Cathedral represents a monolithic, highly planned, top-down style of software development" (Eunice, 1998b), "All alternative models (considered to be one and called the "Cathedral model")" (Bezroukov, 1999a) and "The paper essentially ignored contemporary techniques in software engineering, using the Cathedral as a pseudonym for the waterfall lifecycle of the 1970s (Royce, 1970)" (Johnson, 1999). On the other hand, for the bazaar metaphor, most interpretations did not go beyond the boundary of Raymond's article but terms such as "somehow results in high quality software" (Pavlicek, 2000, p. 11) and "for some mysterious reason" (Bezroukov, 1999a) could be found in literature. This implied a yearning for more detail explanation in the exact mechanism of the software development process in Open Source/Free Software.

After the conceptualisation of the metaphor, different researches were done to explain the mechanics of Open Source/Free Software further by employing certain social or economic theories (Kelty, 2000; Kuwabara, 2000; Edwards, 2001), obtaining statistics from Open Source/Free Software archives and mailing lists (Dempsey, et al., 1999; Ghosh & Prakash, 2000; Moon & Sproull, 2000; Yamauchi, et al., 2000; Kienzle, 2001) and describing the detail processes from a software engineering viewpoint (Fogel, 1999; Vixie, 1999). The model presented in this paper is developed from yet another direction which would possibly introduce another viewpoint to the understanding Open Source/Free Software.

Relevant areas of interest to the topic of Open Source/Free Software

Before introducing a new model to explain Open Source/Free Software, it is useful to understand the relevant areas of interest to the topic of Open Source/Free Software. It can be proposed that there were three areas of interest that are related to Open Source/Free Software, namely, the contextual, technological and socio-economical aspects. The three aspects proposed are not mutually exclusive and they all overlap with each other (figure 1).

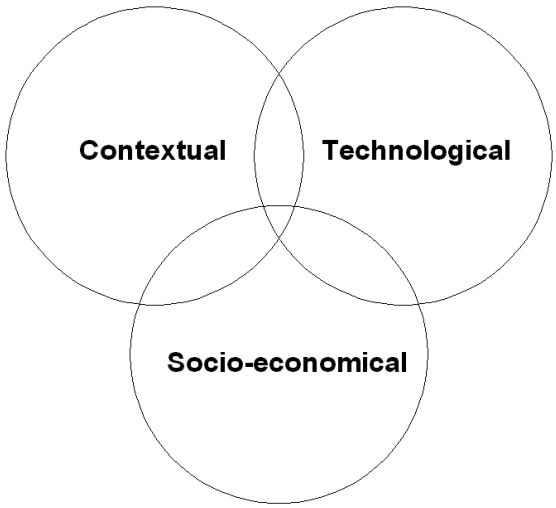


Figure 1: Three areas of interest on Open Source/Free Software

First of all, the Open Source/Free Software movements emerged from its own context. Though the term "Open Source" was coined on the 3rd February, 1998 (Open Source Initiative, 2000), the historical context of the movement includes the history of Unix operating system (Salus, 1995; Hauben & Hauben, 1997), the Internet (Licklider & Taylor, 1968; Hauben & Hauben, 1997), and the hacker culture (Levy, 1984; Turkle, 1984; Raymond, 1999a). The Free Software Foundation and the GNU project were also very significant (FSF). The contemporary context of Open Source/Free Software includes business interest in Open Source (Raymond, 1999b; Sun, 2000; Apple, 2000) such as how Linux was employed as a weapon against Microsoft (Bezroukov, 2000).

Open Source/Free Software communities were mostly made up of volunteers from technical background (Bentson, 2000) and thus technology is another indispensable aspects. The technological architecture (such as microkernel vs monolithic debate (DiBona, Ockman & Stone, 1999)) and features (such as technical supremacy of Linux over Microsoft (The Unix vs NT Organization, 2001)) of software were always

important focuses in the communities. As mentioned above, software engineering is also one of the several approaches which are used to investigate Open Source/Free Software.

Open Source/Free Software communities also consist of a socio-economical aspect and relevant topics includes virtual communities (Rheingold, 1993; Romm, Pliskin & Clarke, 1997; Wellman & Gulia, 1999), current state of hacker culture (Raymond, 1998; Raymond, 2000), information economy (Ghosh, 1998a; Clarke, 1999; Kollock, 1999) and the socio-political effects of Open Source/Free Software (Newman, 1999; Yee, 1999; Forge, 2000).

The model presented in this paper is developed from theories on virtual communities and Computer-Supported Co-operative Work (CSCW). The definition of CSCW is "concerned with the ways in which people work together and with the ways in which computer systems can be designed to support the collaborative aspects of work." (Rosenberg & Hutchison, 1994, p. 1) and it is related in the technical as well as socio-psychological aspects of a system. Since software is developed in a collaborative fashion by communication through computer systems in an Open Source/Free Software community, theories in CSCW is also relevant to the examination of Open Source/Free Software (Yamauchi, et al., 2000). Therefore, the model is mainly based on both technical and socio-economical aspects of Open Source/Free Software.

Open Source/Free Software Community, a definition

Before introducing a model to represent an Open Source/Free Software community, a definition of an Open Source/Free Software community is presented. According to Romm, Pliskin and Clarke (1997) on their comment of the three phase model on virtual communities, although virtual communities are not as 'robust' as face-to-face communities, there are four criteria to judge whether a certain online group can be regarded as a community, namely "shared goal and ideals; some degree of stability; growth; and loyalty and commitment by their members" (Romm, Pliskin and Clarke, 1997, p. 262). Therefore, an Open Source/Free Software community can be regards as a group of people who are committed to develop and deploy certain Free Software/ Open Source software and the group also has to satisfy the four criteria stated above.

Within the Open Source/Free Software movements, there are different sub-cultures. Raymond (1998) stated that there are different ideologies within members who support the idea of Open Source. Two most prominent factions are Open Source Initiative vs

Free Software Foundation. The difference between the two communities was nicely summarised by (Kelty, 2000, p. 312) as "Whereas FSF would sell freedom if they could, opensource.org sells a better mousetrap, or perhaps 'bug-trap' is the better metaphor." While the Free Software Foundation was hardline in taking closed-source software as morally wrong, Open Source Initiative is marketing the Open Source software development process as the definite method for software projects. It is not uncommon to find discussions on the differences and resolutions of the two communities can be found in popular Open Source/Free Software online forums such as Advogato (Advogato, 2000; Advogato, 2001). Therefore, according to one the four criteria stated on a virtual community, "shared goal and ideals" (Romm, Pliskin and Clarke, 1997, p. 262), it is more reasonable to say there are a number of communities within the Open Source/Free Software movements with different ideals rather than thinking that everyone in the movement adhere to exactly the same values.

A Framework on Computer-Supported Co-operative Work (CSCW) and Analysis of an Open Source/Free Software Community

In order to categorise and analyse what happens in an Open Source/Free Software Community, a framework on Computer-Supported Co-operative Work (CSCW) was considered. The framework is shown in figure 2 (Dix, 1994, p. 17). In the diagram, the circles with a 'P' denoted persons involved and the circle with an 'A' denotes the artefact(s) involved in CSCW. The persons involved can directly control the artefact and feedback is received from such maneuver. It is also possible to obtain information about how another person is controlling the artefact through the feedback process. This event is called feedthrough and it is denote by a line connecting the two persons via the artefact. In a CSCW system, the persons involved usually are provided a communication media to exchange ideas. The line 'direct communication' denotes this kind of communication. The dotted line deixis represented the content in the direct communication that referred to the artefact. Moreover, the persons involved may also communicate on concepts of a higher level such as the goal of the co-operation. The line 'understanding' denotes this kind of communication.

+

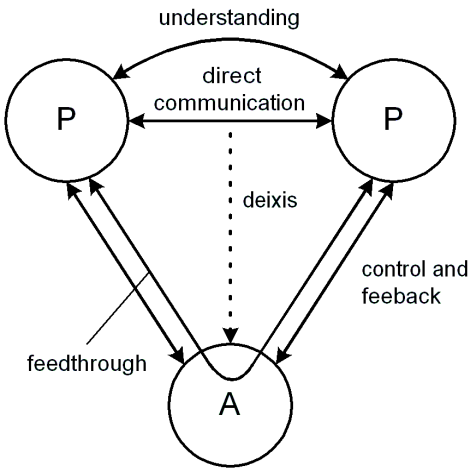


Figure 2: A Framework on CSCW

From this framework, important aspects of an Open Source/Free Software community can be identified. First of all, an Open Source/Free Software community is based on a communication media. As most of the important artefacts in an Open Source/Free Software community are information in digital format, these artefacts can also be contained in the communication media. The next important aspect is the artefacts, which is the contributions from the community members. On top of the artefacts, the communication on how to manage the artefacts is also very important. The understanding of co-operation in CSCW is analogous to the culture of an Open Source/Free Software community, which embodied understanding of high level concepts such as the goal and the nature of the community.

Four-Layer Model on an Open Source/Free Software Community

Based on the four important aspects in an Open Source/Free Software community identified, a model of an Open Source/Free Software community is built and shown in figure 3. The model is presented in a four-layer (4C) model.

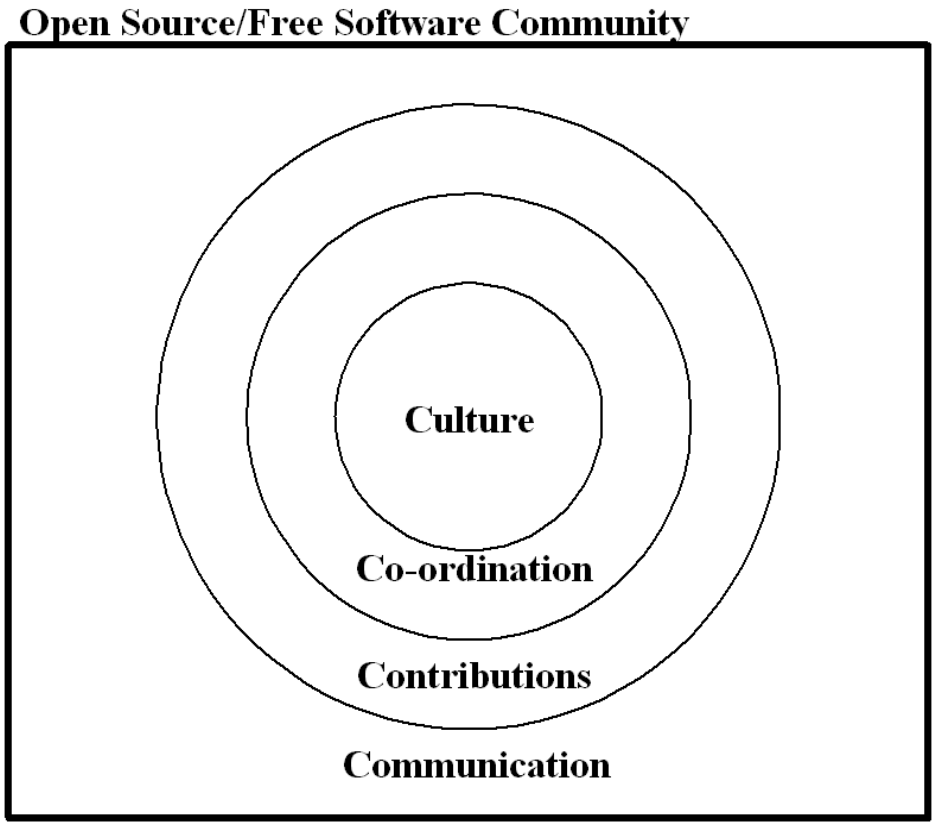


Figure 3: 4C Model of an Open Source/Free Software Community

The four layers represented in the model in figure 3 are communication, contributions, co-ordination and culture respectively. The communication media is the basic infrastructure for any interaction. Contributions referred to the different pieces of assistance given by individual developers via the communication media. Co-ordination is the process of organising fragments of contributions into usable products and the culture of the community in turn governs the rules in co-ordination.

An important enabling factor for Open Source/Free Software communities to exist is a media for communication. In the history of the practice of making source code freely available, the media could be just a roll of paper tape with a program on it in the case of the spacewar program (Levy, 1984). The media could also be a multi-user computer system that allowed every users to access absolute any source code and data in the system in the case of ITS, the Incompatible Time-sharing System (Levy, 1984). Nowadays, the Internet is the most frequently used communication media for Open Source/Free Software communities. Many (Bezroukov, 2000; Moon & Sproull, 2000; Raymond, 2000) recognized that the Internet as an important factor for the Linux

project to start. Kollock (1999) suggested that the Internet lower the cost of collaboration. On the other hand, Ghosh (1998a) used a cooking-pot as a metaphor to describe collaboration on the Internet. In the case of a physical cooking-pot, when everyone put in some ingredients to boil a tasty broth, one can only take a small portion of the broth, more or less the same as what one has put in. In the case of the Internet, the digital cooking-pot, which is an efficient cloning machine, everyone who contributes can also get complete copies what others have contributed.

An Open Source/Free Software project is built upon contributions from individual developers. These contributions included source code, suggested features (wish list), comments on project, bug reports and patches and also documentations. Source code is the basis of any program and thus any project. When a project starts, the existence of a runnable program with source code attracts more developers to participate (Fogel, 1999; Raymond, 2000). After using the program, developers or users may have suggestions on new features to add to the program. Comments may also be made on the direction of the project as well as the details of the source code. Zawinski (1999) pointed out that contributing quality comments could even worth more than source code. Bug reports and patches with source code are also welcomed to improve the stability of the program. Finally, a program cannot be used and a project cannot be maintained without documentations, and thus contributions to documentation are also important. With a communication media, all these contributions can be collected.

Co-ordination is required to package all these different contributions collected via the communication media into a piece of stable software. One of the most important management issues at the beginning of the project is promotion (Fogel, 1999; Raymond, 2000). It is important to build a community of developers and users for the project to proceed. Licensing, which legislates what kind of freedom is placed on the distribution of the source code, is another significant management issue at the start. Then a mechanism to judge which piece of contribution to be accepted or rejected has to be established. If the benevolent dictator (Raymond, 1998; Fogel, 1999) system is adopted, a maintainer has to be appointed to make final judgements on decisions of the project. A maintainer of a project can be transferred or taken over (forking) later on (Raymond, 1998; Fogel, 1999). If an autocratic (Raymond, 1998; Fogel, 1999) system is adopted, a membership system has to be setup and it may also involve a voting system for decision making. When the time comes to produce a stable version from an evolving project, a release procedure may be introduced (Fogel, 1999). This procedure

will 'freeze' normal development on the project and concentrate group effort on making the program stable. The 'freeze' measure will be lifted after the release and normal development procedures will resume.

The culture of an Open Source/Free Software community shapes the rules in co-ordination of Open Source/Free Software projects. Culture is defined as "the collective programming of the mind which distinguishes the members of one group or category of people from another." (Hofstede, 1997, p. 5) and the community of Free Software and Open Source movements can be argued to have enough affinity to be called a culture. First of all, most of the members in the community are technical people (Bentson, 2000), that value technical correctness (Pavlicek, 2000) and virtuosity (hack) other than formal authorities (Levy, 1984). Secondly, Linus Torvalds, the original author of Linux, released the source code of the system on the USENET because the culture encourages sharing (Ghosh, 1998b). Thirdly, Raymond (1998) also observed cultural rules in Open Source/Free Software communities in the transfer of maintainership and giving credits. Fourthly, formed mostly by volunteers, the culture endorses loose charter over complicated legalisations when the community tries to put management rules in writing, as volunteers tend to cooperate and reach consensus rather than exploiting the loopholes in the system (Fogel, 1999). The above is a general view of the culture and each Open Source/Free Software community also has its own variation.

Model on individual participation in an Open Source/Free Software Community

After introducing a model to an Open Source/Free Software community, one can consider the relationship of individual participants to the community. The model built to explain this relationship is shown in figure 4. The model includes the mentioned 4C model, the motivations and barriers when a developer decides to join an Open Source/Free Software community, the positive and negative results after interaction with an Open Source/Free Software community.

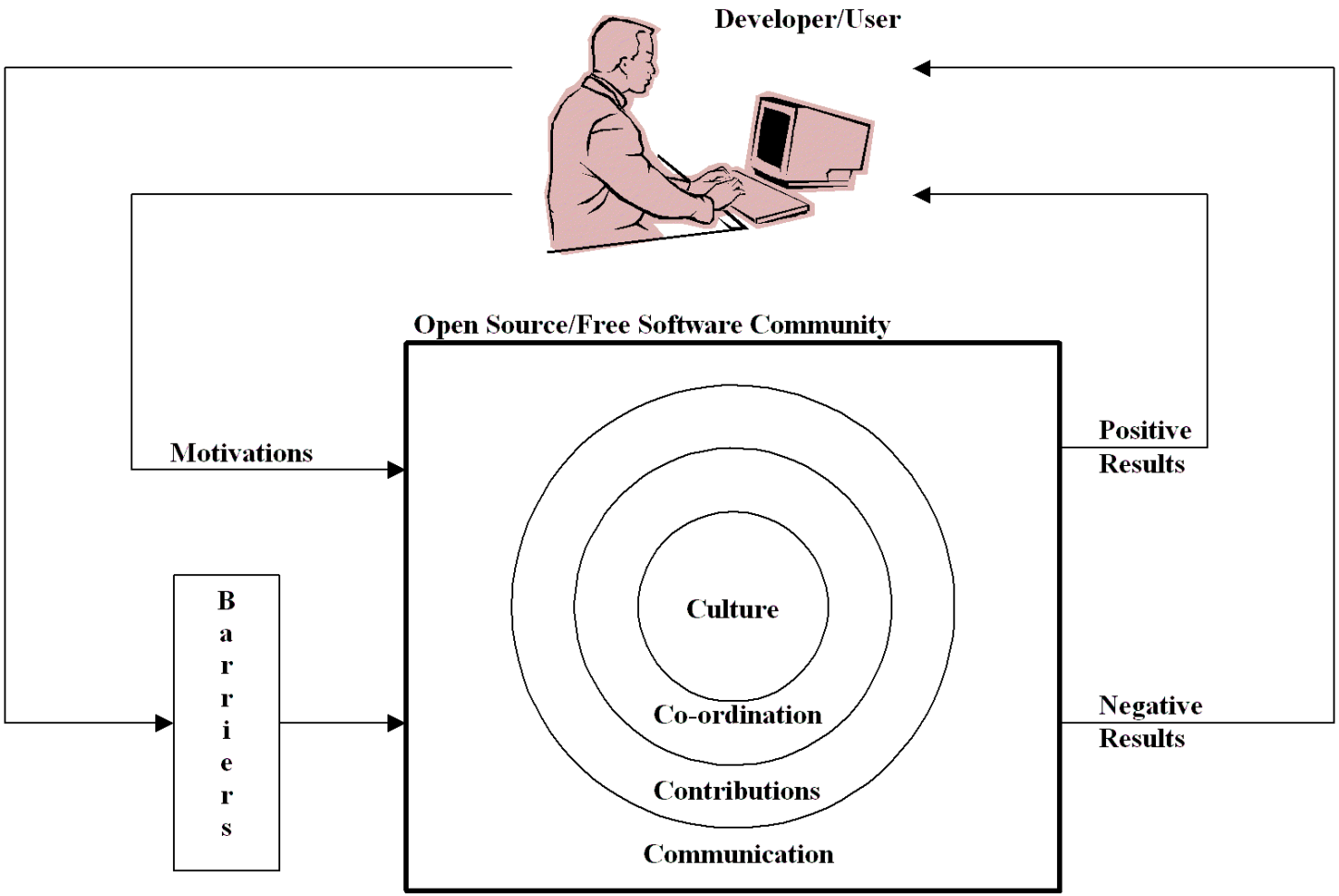


Figure 4: A Model on individual participation in an Open Source/Free Software Community

There are a number of motivations for a developer to join an Open Source/Free Software community. A developer may see that joining an Open Source/Free Software community as a good solution to solve his or her need on a piece of software (Fogel, 1999; Kuwabara, 2000; Raymond, 2000; Evers, 2001). Moreover, social factors such as reciprocal behaviour (Wellman & Gulia, 1997; Kollock, 1999), reputation (Ghosh, 1998a; Raymond, 1998; Fogel, 1999; Kollock, 1999; Kuwabara, 2000) and attraction to community (Foster, 1998; Kollock, 1999) may play a part. Lastly, altruism or idealism (Kollock, 1999) may also motivate developers to contribute.

Although there are a number of motivations for developers to join an Open Source/Free Software community, barriers also exist to deter them, as in any virtual communities (Romm, Pliskin & Clarke, 1997). Technically, Open Source/Free Software communities only accept developers who attain a high degree of competence (Raymond, 2000). The complexity of source code also created a barrier for contribution (Zawinski, 1999). On the other hand, software with poor design and inadequate documentation may deter contribution (mettw, 2000). The original developer also may not be willing to share his or her own code. Cultural barriers also exist. Firstly, language can be a barrier because person from certain backgrounds in some part of the world may find it hard to join an Open Source/Free Software community with English as the common language of communication (Fogel, 1999). Cultural mysteries also exist and they have to be solved before a member could be accepted by certain Open Source/Free Software communities (Raymond, 1998). The last but obvious reason is that a developer cannot or cease to involve in an Open Source/Free Software community because the developer can afford the time no more (Bezroukov, 1999a).

There are several positive outcomes as a result of joining an Open Source/Free Software community. A developer may have one own itch scratched (Raymond, 2000) and enjoyed programming in collaboration (Raymond, 1998; Fogel, 1999). He or she may learn more skills (Fogel, 1999) and build up one's reputation (Ghosh, 1998a; Raymond, 1998; Fogel, 1999; Kollock, 1999; Kuwabara, 2000) in the community as well.

Negative results from participation in an Open Source/Free Software community include lack of interest on one's project (Fogel, 1999; Raymond, 2000), rejection from

others (Maclachlan, 1999; Pennington, n.d.), hurts in management issues (Raymond, 1998; Hacker, 1999) and burn-out (Bezroukov, 1999a; Bezroukov, 1999b).

Discussion on the Construction of the Model

Kelty (2000; 2001) pointed out that "the Cathedral and Bazaar" described the process of how to run an Open Source/Free Software project as a replica of Linux. This focus unfortunately reduces the phenomenon of Open Source/Free Software into a technical process. This is, however, not to say that Raymond did not know about culture. On the contrary, he was the complier of "The New Hacker's Dictionary" (Raymond, 2001). Moreover, in the "Homesteading the Noosphere" (Raymond, 1998), the next essay after "The Cathedral and Bazaar", he mentioned various aspects of the different sub-cultures within Open Source. Unfortunately, probably in the process of marketing Free Software and by de-politicisation and renaming it to "Open Source" (Kelty, 2000), the complexity of the phenomenon was reduced. To conclude, the metaphor of the Cathedral and Bazaar is useful as an introductory, first estimate to the phenomenon of Open Source/Free Software but more is needed to explain the phenomenon. The model presented in this paper is one of the many attempts to contribute towards a more comprehensive and complex explanation, which covers both technical and social aspects.

The model presented in paper is an attempt to identify the important concepts of an Open Source/Free Software community and aspects when an individual participates in an Open Source/Free Software community, namely, communication, contributions, co-ordination, culture, motivation, barriers, positive and negative results. The model is designed not to be over-specific as Open Source/Free Software is still an evolving phenomenon (Fogel, 1999). However, the advantage and disadvantage with the model is its flexibility. Arguably, the model is flexible enough even to include other non-Open Source/Free Software community. For example, the model can be used to examine communities that chosen to use a closed-source license in a commercial environment. Moreover, by substituting contributions, co-ordination and culture by information, channel of communication and pedagogy, the model can be changed to analyse a virtual learning community. By looking at the model as the descendent of the three phase model on virtual communities (Romm, Pliskin and Clarke, 1997) and the framework on CSCW (Dix, 1994), it is not surprising that this model on Open Source/Free Software community could be expanded to explain many different systems as its parent models are general models on information systems. One obvious limitation is that there need to be collective agreement on the philosophy of how information should be managed

within the system, which is called culture in the model, in order for the model to produce a useful analysis. The advantage of this flexibility is that an Open Source/Free Software community can be compared with other information systems by a similar framework under this model. The disadvantage is that the model may disappoint those who want to pin down what Open Source/Free Software really is. However, it seems that from the above analysis that Open Source/Free Software actually includes a collection of different practices rather than a few distinct methods. Therefore, it will be useful to identify the different parameters and practices in future research.

According to the three phase model on virtual communities (Romm, Pliskin & Clarke, 1997), a virtual community may cause changes to its immediate environment and also transform the society. There are evidences that an Open Source/Free Software community also causes some of those changes. For example, one of the changes to the immediate environment is linguistic, which means changes in the use of language. In the case of the Open Source/Free Software communities, the Jargon File (Raymond, 2001), which is a dictionary with a collection of 2321 entries on hacker vocabulary, is a good piece evidence in this aspect. However, some of the impacts of the Open Source/Free Software communities, such as its impact to the software industry and its contribution to the debate of information freedom, are yet to be examined. The possibility of building an extended model therefore can be explored.

Implication to Study of Open Source/Free Software and Dependability

Dependability is defined as "system properties that allows us to rely on a system functioning as required. Dependability encompasses, among other attributes, reliability, safety, security, and availability." (Littewood & Strigini, 2000, p.1). According to Littlewood and Strigini (2000), Open Source/Free Software projects can serve as an archive for dependability research. By analysing the Open Source/Free Software by the model presented, several relevant research directions can be found.

As mentioned above, the model presented is suitable for the study of dependability because it covers both technical and social aspects of an Open Source/Free Software community. This coincides with the nature of the study of dependability that a multi-disciplinary approach is required (Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems, 2001).

From the analysis of the model presented, there is no "the Open Source community" but different sub-cultures and practices in different Open Source/Free Software communities. The implication for using Open Source/Free Software as an archive for research is that it is useful to first identify the different possible parameters in running an Open Source/Free Software project and view the archive as a collection of different approaches to software project management. Then researches can be done on discovering the relationships of these different parameters with dependability. One example is that the BSD community stressed on action (for example, submission of code) over discussion (Yamauchi, et al., 2000) while 17% of projects on a prominent Open Source/Free Software hosting site, SourceForge, are in planning stage (Kienzle, 2001). The topic of diversity can also be investigated from the archive of forking or rivalry projects such as Emacs/XEmacs, OpenBSD/FreeBSD/NetBSD and KDE vs GNOME. One may able to investigate relationship between the dependability of software and the situation that different teams are writing software that is similar.

Conclusion

The model presented in this paper covers both technical and social aspects of Open Source/Free Software with flexibility to include the different practices. This model includes a four-layer (4C) model of the Open Source/Free Software community, the motivations and barriers when a developer decides to join an Open Source/Free Software community, the positive and negative results which occur after interaction with an Open Source/Free Software community. Open Source/Free Software is found to be an archive of various software practices possibility useful in dependability research. Rivalry projects also exist which may benefit research in diversity.

List of References

Advogato 2000, 'Fallible Hacker Figureheads', <http://www.advogato.org/article/123.html> (Accessed 1 Feb. 02).

Advogato 2001, 'On Holy Wars and a Plea for Peace #2', <http://www.advogato.org/article/396.html> (Accessed 1 Feb. 02).

Apple 2000, 'Apple - Public Source', <http://www.publicsource.apple.com/> (Accessed 7 Jun. 00).

Bentson, R. 2000, 'The Proper Image for Linux', <http://www2.linuxjournal.com/lj-issues/issue57/2931.html> (Accessed 29 Dec. 00).

Bezroukov, N. 1999a, 'Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism)', *First Monday*, vol. 4, no. 10, Oct, 1999, <http://firstmonday.org/issues/issue4_10/bezroukov/index.html> (Accessed 2 Jun. 99).

Bezroukov, N. 1999b, 'A Second Look at the Cathedral and Bazaar', *First Monday*, vol. 4, no. 12, Dec, 1999, <http://firstmonday.org/issues/issue4_12/bezroukov/index.html> (Accessed 2 Jun. 99).

Bezroukov, N. 2000,' 4.1. Linus and Linux; Linus Torvalds' Short Unauthorized Biography', <http://www.softpanorama.org/People/Torvalds/Linus_Torvalds_biography.shtml>, (Accessed 3 Nov. 00).

Clarke, R. 1999, 'The Willingness of Net-Consumers to Pay: A Lack-of-Progress Report', *Proc. 12th International Bled EC Conf.*, Slovenia, June, <http://www.anu.edu.au/people/Roger.Clarke/EC/WillPay.html> (Accessed 22 Apr. 00).

Dempsey, B. et al. 1999, 'A quantitative profile of a community of open source Linux developers', <http://metalab.unc.edu/osrt/develpro.html> (Accessed 5 Oct. 00).

DiBona, C. Ockman S. & Stone M. (Eds) 1999, ’Appendix A: The Tanenbaum-Torvalds Debate’, In Chris DiBona, Sam Ockman and Mark Stone (Eds), *Open Sources: Voices from the Open Source Revolution*, CA, Sebastopol: O’Reilly & Associates, <http://www.oreilly.com/catalog/opensources/book/appa.html> (Accessed 11 Jul. 00).

Dix, A. 1994, ’Computer Supported Cooperative Work: A Framework’, In Rosenberg, D. & Hutchison, C. (Eds.), *Design Issues in CSCW*, London: Springer-Verlag, pp. 9-26.

Edwards, K. 2001, ’Towards a Theory for Understanding the Open Source Software Phenomenon’, <http://www.its.dtu.dk/ansat/ke/towards.pdf> (Accessed 8 Jun. 01).

Eunice, J. 1998a, ’Beyond the Cathedral, Beyond the Bazaar’, <http://www.illuminata.com/public/content/cathedral/cathedral5.htm> (Accessed 19 Oct. 01).

Eunice, J. 1998b, ’Beyond the Cathedral, Beyond the Bazaar’, <http://www.illuminata.com/public/content/cathedral/intro.htm> (Accessed 19 Oct. 01).

Evers, J. 2001, ’Users order StarOffice, protest Microsoft licensing scheme’, IDG News Service, 16 Jul., <http://www.nwfusion.com/news/2001/0716staroffice.html> (Accessed 13 Aug. 01).

Fogel, K. 1999, *Open Source Development with CVS*, Arizona:Coriolis.

Forge, S. 2000, ’Open Source: The Economics of Giving Away Stuff, and Software as a Political Statement’, Info, vol. 2, no. 1, February, Camord Publishing.

Foster, E. 1998, ’1997 Product of the Year: Best Technical Support Award: Linux User Community’, InfoWorld, <http://www.infoworld.com/cgi-bin/displayTC.pl?/97poy.supp.htm>, (Accessed 27 Nov. 00).

Ghosh, R. A & Prakash, V. V. 2000, 'The Orbiten Free Software Survey', First Monday, volume 5, number 7, Jul, 2000, <http://firstmonday.org/issues/issue5_7/ghosh/index.html> (Accessed 29 Aug. 00).

Ghosh, R. A. 1998a, 'Cooking pot markets: an economic model for the trade in free goods and services on the Internet', *First Monday*, vol. 3, no. 3, Mar, 1998, <http://www.firstmonday.dk/issues/issue3_3/ghosh/index.html > (Accessed on 2 Jun. 00).

Ghosh, R. A. 1998b, 'FM Interview with Linux Torvalds: What motivates free software developers', *First Monday*, vol. 3, no. 3, Mar, 1998, <http://www.firstmonday.org/issues/issue3_3/torvalds/index.html>

Hacker, J. Q. 1999, 'Feature: Conflicting Open Source Developers' <http://slashdot.org/features/99/07/12/1639202.shtml> (Accessed 26 Nov. 00).

Hamerly, J. & Paquin, T. 1999, 'Freeing the Source: The Story of Mozilla', In Chris DiBona, Sam Ockman and Mark Stone (Eds), *Open Sources: Voices from the Open Source Revolution*, CA, Sebastopol: O'Reilly & Associates, <http://www.oreilly.com/catalog/opensources/book/netrev.html> (Accessed 11 Jul. 00).

Hauben, M. & Hauben, R. 1997 *The Netizens and the Wonderful World of the Net: An Anthology*, <http://studentweb.tulane.edu/~rwoods/netbook/contents.html>, (Accessed 22 May 00).

Hofstede, G. H. 1997, *Cultures and Organizations*, Berkshire, England, McGraw-Hill.

Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems 2001, 'Aims and Objectives', <http://www.dirc.org.uk/aims/index.html> (Accessed 01 Feb. 02).

Johnson, K. 1999, 'Open-Source Software Development', <http://www.cpsc.ucalgary.ca/~johnsonk/SENG/SENG691/open.htm> (Accessed 22 Dec. 00)

Kelty, C. M. 2000, *Scale and Convention: Programmed Languages in a Regulated America*, Ph.D Dissertation, Mass. Institute of Technology.

Kelty, C. M. 2001, 'Hau to do things with words', *Knowledge and Society*, vol. 13, JAI Press, <http://www.kelty.org/or/papers/hauto.kelty.pdf> (Accessed 29 Jan 02).

Kienzle, R. 2001, 'Sourceforge Preliminary Project Analysis', <http://www.osstrategy.com/sfreport/> (Accessed 23 Jan. 02).

Kollock, P. 1999, 'The economies of online cooperation: gifts and public goods in cyberspace', In Smith, M. A. & Kollock, P., *Communities in Cyberspace*, Routledge, London, pp.220-242.

Kuwabara, K. 2000,'Linux: A Bazaar at the Edge of Chaos', *First Monday*, volume 5, number 3, Mar, 2000, <http://firstmonday.org/issues/issue5_3/kuwabara/index.html>, (Accessed 31 Jul. 00).

Levy, S. 1984, *Hackers: Heroes of The Computer Revolution*, Garden City, N.Y. : Anchor Press/Doubleday, 1984.

Licklider, J.C.R. & Taylor, R. 1968 'The Computer as a Communication Device' In *In Memoriam: J.C.R. Licklider 1915-1990*, Aug. 7, 1990, p. 40, reprinted by permission from Digital Research Center; originally published as 'The Computer as a Communication Device,' in *Science and Technology*, April, 1968, pg. 40.

Littlewood, B. & Strigini, L. 2000, 'Software reliability and dependability: a roadmap', In A. Finkelstein, *22nd Int. Conf. on Software Engineering*, ACM Press, Limerick, pp. 177-188, June 2000, <http://www.dirc.org.uk/publications/papers/11.pdf> (Accessed 21 Jan. 02).

Maclachlan, M. 1999, 'Panelists Describe Open Source Dictatorships', <http://www.techweb.com/news/story/TWB19990812S0003> (Accessed 26 Nov. 00).

mettw 2000, 'Contribution balance', In response to lalo, 'Ask the Advogatos: why do Free Software projects fail?', <http://www.advogato.org/article/128.html> (Accessed 19 Oct. 00).

Moody, G. 2001, *Rebel Code: The Inside Story of Linux and the Open Source Revolution*, Perseus, Cambridge, Massachusetts.

Moon, J. Y. & Sproull, L. 2000, 'Essence of Distributed Work: The Case of the Linux Kernel', First Monday, volume 5, number 11, Nov, 2000, <http://firstmonday.org/issues/issue5_11/moon/index.html> (Accessed 15 Nov. 00).

Newman, N. 1999 'The Origins and Future of Open Source Software: A NetAction White Paper ', http://www.netaction.org/opensrc/future/oss-whole.html (Accessed 28 Jul. 00).

Open Source Initiative 2000, 'History of the Open Source Initiative', <http://www.opensource.org/history.html> (Accessed 18 Jun. 00).

Pavlicek, R. C. 2000, *Embracing Insanity: Open Source Software Development*, SAMS, Indianapolis, Indiana.

Pennington n.d., 'Working on Free Software', <http://www106.pair.com/rhp/hacking.html>, (25 Nov. 00).

Raymond, E. S. 1998, 'Homesteading the Noosphere', <http://www.tuxedo.org/~esr/writings/homesteading/> (Accessed 30 May 00).

Raymond, E. S. 1999a, 'A Brief History of Hackerdom', In Chris DiBona, Sam Ockman and Mark Stone (Eds), *Open Sources: Voices from the Open Source Revolution*, O'Reilly & Associates, Sebastopol, CA, 1999, <http://www.oreilly.com/catalog/opensources/book/raymond.html> (Accessed 11 Jul. 00).

Raymond, E. S. 1999b, 'The Magic Cauldron', <http://www.tuxedo.org/~esr/writings/magic-cauldron/> (Accessed 30 May 00).

Raymond, E. S. 2000, 'The Cathedral and the Bazaar', <http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html> (Accessed 30 May 00).

Raymond, E. S. (Ed.) 2001, 'Jargon File Resources', Version 4.3.1, 29 Jun., <http://www.tuxedo.org/~esr/jargon/jargon.html> (Accessed 28 Aug. 2001).

Rheingold, H. 1993, *The Virtual Community*, Reading, Massachusetts: Addison-Wesley.

Romm, C., Pliskin, N. & Clarke, R. 1997, 'Virtual Communities and Society: Toward an Integrative Three Phase Model', *International Journal of Information Management*, Great Britian, vol. 17, no. 4, pp. 261-270.

Rosenberg, D. & Hutchison, C. 1994, 'Introduction', In Rosenberg, D. & Hutchison, C. (Eds.), *Design Issues in CSCW*, London: Springer-Verlag, pp. 1-8.

Royce, W. W. 1970, Managing the Development of Large Software Systems: Concepts and Techniques, *WESCON Technical Papers*, Vol. 14.

Salus, P. H. 1995, *A Quarter Century of Unix*, Reading, Massachusetts: Addison-Wesley.

Sun 2000, 'Sun Community Source Licensing', <http://www.sun.com/software/communitysource/> (Accessed 7 Jun. 00).

The Unix vs NT Organisation 2001, 'The Unix vs NT Organisation', <http://www.unix-vs-nt.org/> (Accessed 14 Sep. 01).

Turkle, S. 1984, 'Hackers: Loving the Machine for Itself', In Turkle, S., *The Second Self: Computers and the Human Spirit*, NY: Simon & Schuster, Chapter 6, pp.196-238.

Vixie, P. 1999, 'Software Engineering', In Chris DiBona, Sam Ockman and Mark Stone (Eds), *Open Sources: Voices from the Open Source Revolution*, O'Reilly & Associates, Sebastopol, CA, 1999, <http://www.oreilly.com/catalog/opensources/book/vixie.html> (Accessed 11 Jul. 00).

Wellman, B. & Gulia, M 1999, 'Net Surfers Don't Ride Alone: Virtual Communities as Communities', In Wellman, B. (Ed.), *Networks in the global village: life in contemporary communities*, Westview Press, Boulder, CO, pp. 331-366.

Yamauchi, Y. et al. 2000, 'Collaboration with Lean Media: How Open-Source Software Succeeds', ACM Conference on Computer Supported Cooperative Work (CSCW2000), Philadelphia, PA <http://www.bol.ucla.edu/~yutaka/papers/yamauchi_cscw2000.pdf > (Accessed 18 Jun. 01).

Yee, D. 1999, 'Development, Ethical Trading, and Free Software', <http://danny.oz.au/freedom/ip/aidfs.html> (Accessed 11 Sep. 00).

Zawinski, J. 1999, 'resignation and postmortem.', <http://www.jwz.org/gruntle/nomo.html> (Accessed 22 Jun. 2000).

On the Pareto distribution of Sourceforge projects

Francis Hunt[1] and Paul Johnson[2]

[1] Centre for Technology Management, Cambridge University Engineering Department, Mill Lane, Cambridge CB2 1RX

[2] Marconi Labs, Gates Building, J J Thomson Avenue, Cambridge CB3 0FD

Abstract

Open source software has risen to prominence within the last decade, largely due to the success of well known projects such as the GNU/Linux operating system and the Apache web server, amongst others. Their significant commercial impact, with GNU/Linux reportedly running on 25% of server machines and Apache on 60% of web servers, has prompted many companies who use and who develop software to reassess their traditional modes of functioning. A number of companies such as IBM, HP and Sun have invested significantly in developing open source software. Much early written work on open source software development aimed at raising awareness and advocating its uptake. More recently the interest has been in quantifying and qualifying the advantages, disadvantages and other features of open source software. This paper aims to contribute in this second area.

Most work on open source implicitly treats all projects as equivalent, for want of ways of classifying them. Benefits of 'typical' projects are claimed, with little attention to what constitutes a 'typical' project. In this paper we look at data available on SourceForge, a web site hosting upward of 30,000 open source projects and characterise the distribution of projects. Considering the number of downloads per week of the software, we show that for the most part the data follows a Pareto type distribution i.e. there are a small number of exceptionally popular projects, most projects being much less popular, and the number of projects with more than a given number of downloads tails off exponentially. We offer explanations for this distribution and for the places where the actual distribution deviates from the model and propose ways that these explanations could be tested. In particular there seem to be fewer than expected projects with a small number of weekly downloads. Likely explanations for this would seem to be either that projects with a small number of downloads per week do not tend to use SourceForge, or that this small number of downloads indicates a low level of interest in the project and such projects are inherently unstable (either they die or become more popular).

Two practical applications of this work are: it is useful for people or companies starting an Open Source project to have an idea of what a 'typical' project might entail; secondly, it enables analysis of best practice and benefits to be tied to some sort of classification of projects and allows questions such as how benefits scale with project size to be examined in detail.

Introduction

Open source software has risen to prominence within the last decade, largely due to the success of well known projects such as the GNU/Linux operating system and Apache web server, amongst others. Their significant commercial impact, with GNU/Linux reportedly running on 25% of server machines and Apache on 60% of web servers, has prompted many companies who use and who develop software to reassess their traditional modes of functioning. A number of companies such as IBM, HP and Sun have invested significantly in developing open source software. Much early written work on open source software development was aimed at raising awareness and advocating its uptake, the most famous example being Eric Raymond's "The Cathedral and the Bazaar" essay (1999). More recently the interest has been in quantifying and qualifying the advantages, disadvantages and other features of open source software. This paper aims to contribute in this second area.

This paper analyses and draws conclusions from statistics on the Sourceforge website, a site hosting over 30,000 open source software projects. This site was set up in November 1999, providing freely available infrastructure for running open source software projects. It is the premier site for hosting open source projects, and conclusions about open source development drawn from the Sourceforge site are likely to be relevant to a broad range of open source projects. In this paper we analyse the distribution of projects according to the frequency that their software is downloaded.

Quantitative studies of open source software development are relatively rare. Notable exceptions include studies of the Apache project (Mockus, Fielding et al. 2000), the Gnome project (Koch and Schneider 2000) and of Linux e.g.(Wheeler 2001). Rene

Kienzle (2001) has also investigated the statistics on Sourceforge, in particular the number of developers associated with different projects. His results do not overlap those of this paper and can be considered complementary.

The rest of this paper is structured as follows: first we discuss the data used in the analysis; then we examine the distribution of projects according to the frequency of project downloads; finally we draw conclusions and suggest avenues of further research.

Data

Sourceforge hosts over 30,000 open source development projects. It provides free of charge infrastructure for running such projects, including version control, space for a project website, bug tracking and mailing lists. Significantly (for this paper) it also collects and displays statistics on the various projects, such as the number of times a piece of software has been downloaded on each of the last 30 days, the number of times its web pages have been viewed and the number of times software has been checked in to the version control system. These are all measures of project activity; other such measures which are not displayed on the statistics page include activity on the project mailing lists, bugs reported and new official releases of code. A number of measures are combined into an overall measure of project activity[1], and the most active projects are listed.

Having originally obtained permission to study their website in February 2001, we collected the statistics from all the projects listed on the most active project list on 22nd October and again on the 22nd November, providing two contiguous sets of 30 days[2]. The statistics collected for each day and project were the number of downloads, the number of webpage views and the number of CVS commits.

A natural first question is how reliable are these data. The statistics collection and processing routines on Sourceforge are known to contain bugs (there are open bug reports #462957 from the 19th September on download statistics; #455161 from 24th August on CVS statistics; and #451204 from 15th August on the pageview statistics). However we believe that the download statistics are in general reliable, but care is needed in making generalisations from outliers. In terms of the reliablity of the data capture operation from the Sourceforge site, a visual check was made between the data as displayed on Sourceforge and the data as recorded in our database on five of the projects and we are moderately confident that there were no systematic errors in transcribing the data.

The second question is how representative are these data. This question splits into: how representative the data are of active projects on Sourceforge; and how representative the data are of open source projects in general. The data is representative of active projects on Sourceforge since it contains all active projects. Sourceforge is also the premier site for hosting open source projects, so it is plausible that the data are representative of open source software development as a whole. Some very well known projects do not use Sourceforge e.g. Linux, Apache, Mozilla. There is

---

[1] The actual metric used is in the source code is (Scholl 2001):
`log(3*forum_msgs)+log(4*project_taks)+log(3*bugs)+ log(10*patches) +log(5*supports) +log(cvs_commits) + log(5*developers) +log(5*filereleases) +log(0.3*downloads)) * servey_rating_agregate`.

[2] This data is available for download from http://www-mmd.eng.cam.ac.uk/people/fhh10/fhh10.htm. Although the two sets of 30 days are contiguous, data for 27th October is missing for unknown reasons.

also a cluster of GNU projects that are hosted elsewhere and a recently formed competitor to Sourceforge called Savannah[3]. There are also a number of open source projects that are hosted by companies. Nonetheless, it seems safe to assume that the majority of active open source projects in the world are hosted on Sourceforge and hence studying the Sourceforge data tells us something about open source development.

Analysis

We investigate 3 issues in this paper:

- time series of total sourceforge downloads
- cross sectional distribution of projects at a moment in time
- differential behaviour of segments of the cross section

Time series of total sourceforge downloads

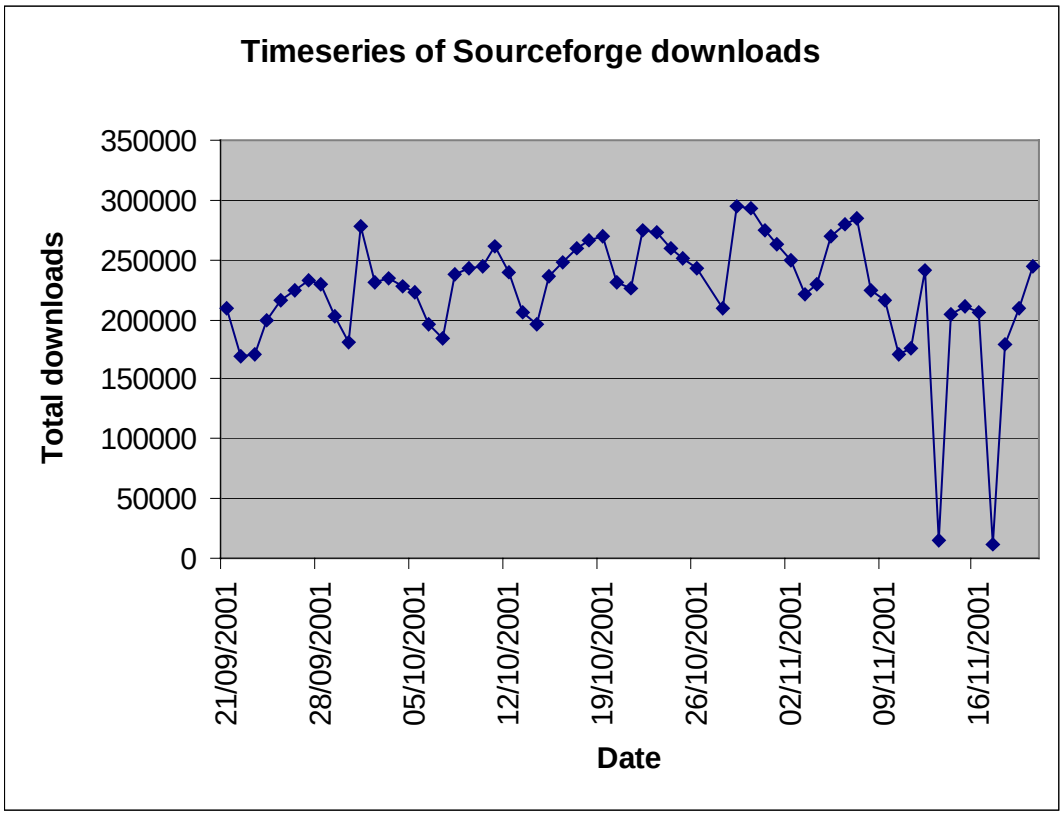


Figure 1: Total downloads from Sourceforge site

Summing the downloads on all the active projects and plotting the time series of the total downloads in figure 1 immediately highlights a number of interesting points. Firstly the initial six weeks indicate a general increase in download activity, though it is possible that this increase is in fact an a return to normality after terrorist attacks in

[3] Savannah was set up by people who thought it ethically unacceptable that VA, the company which own Sourceforge, should produce an enhanced closed source version of the Sourceforge software.

the US on September 11. Secondly, something peculiar happens in mid November, in particular on Tuesday 13th and Saturday 17th November. We do not use this data in our subsequent analysis, because of this unexplained turbulence. Thirdly there is a noticeable weekly cycle. The first day plotted is Friday 21st September and the troughs of the weekly cycle occur at weekends. This is perhaps slightly surprising and suggestive of commercial use of software on Sourceforge, or at least use of commercial resources to download the software.

Cross sectional distribution

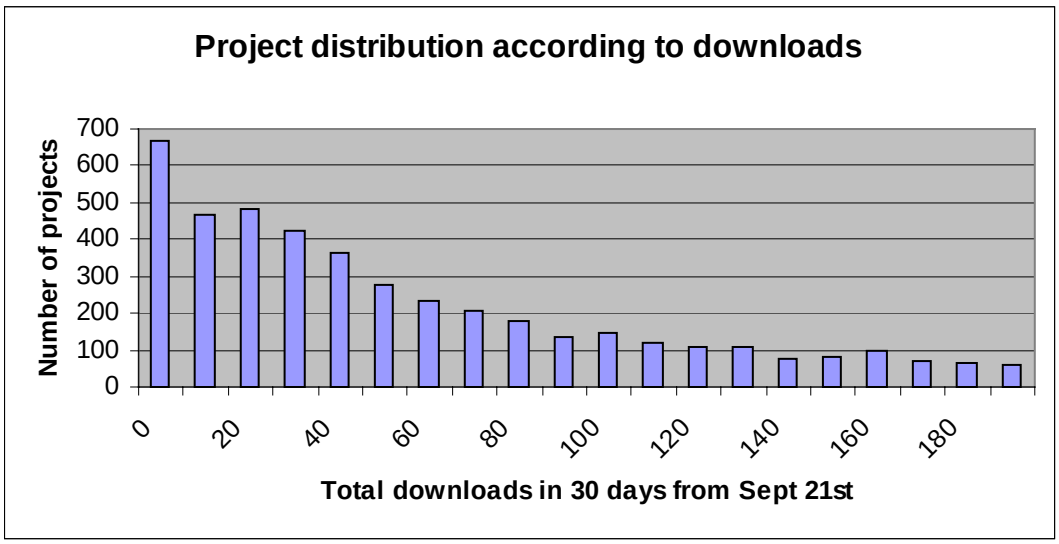


Figure 2: Sourceforge project distribution

If we examine the distribution of projects according to the total number of downloads they received in the 30 days from September 21st, plotting in figure 2 the projects as a histogram with bins width 10, we obtain a very heavily skewed distribution with a tail that extends out to more than 600,000 downloads. The median number of monthly downloads is 70 i.e. half the active projects have between 0 and 70 downloads whilst the other half have between 70 and 600,000 downloads. Such heavily skewed distributions, though not as common as the omnipresent normal distribution, do occur in a wide range of phenomena such as the distribution of incomes (Pareto 1896), earthquake severities (Richter 1958), forest fire sizes, word usage frequencies (Zipf 1949) and web site popularities (Adamic and Huberman 2000). Of particular interest in all the examples just mentioned is that the distribution tails off exponentially e.g. in the case of incomes, this means the ratio of the number of people who earn more than you to the number of people who earn 10 times more than you is the same regardless of your income. Distributions with this property are variously called Pareto, Zipf or power law distributions. Plotting the logarithm of the frequency of an event against the logarithm of its 'size' yields a straight line graph for the tail of such a distribution.

In figure 3 we plot the number of projects with a given number of downloads. We see that the distribution does have the characteristic Pareto tail. There are a number of proposed explanations for such distributions e.g (Bak 1997), but most relevant here is likely to be the winner-takes-all nature of project popularity: as a project grows in popularity it becomes more attractive since there is more likely to be good documentation, knowledgeable people to provide support, further development work, software tools that work with it, …etc.

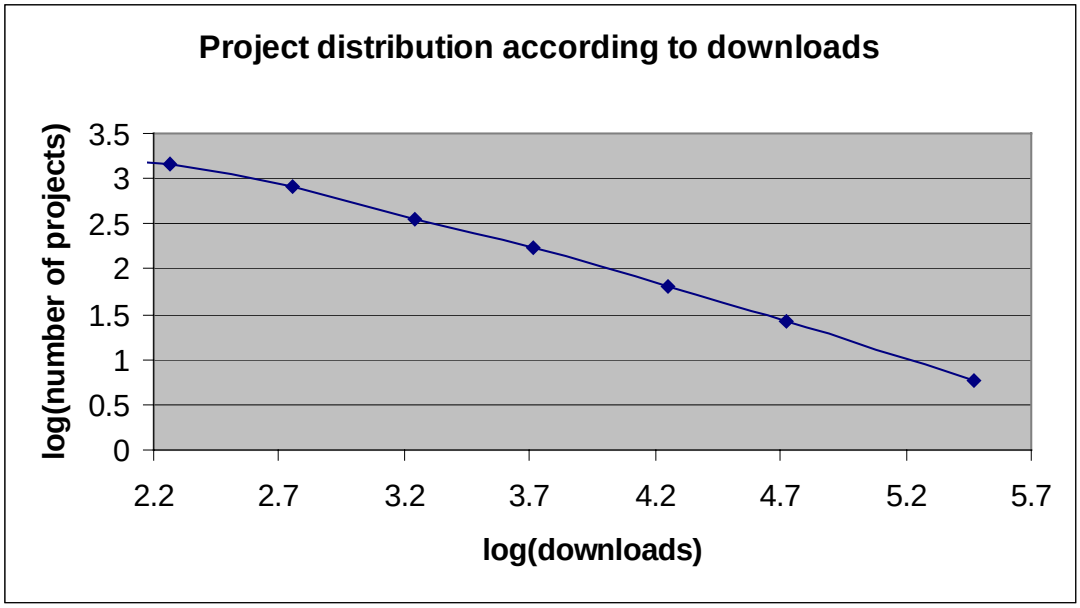


Figure 3: Log-log plot of download frequencies exhibiting a power law over three orders of magnitude

Looking more closely at the number of project downloads in figure 4 using reduced bin sizes we see that there are fewer than expected projects with a low number of downloads. Having checked our data collection, this does not seem to be due to an error in the data collection, or to be an artefact of the sampling strategy. Thus projects with low numbers of downloads are relatively uncommon on Sourceforge. Two competing explanations for this are that either projects with low numbers of

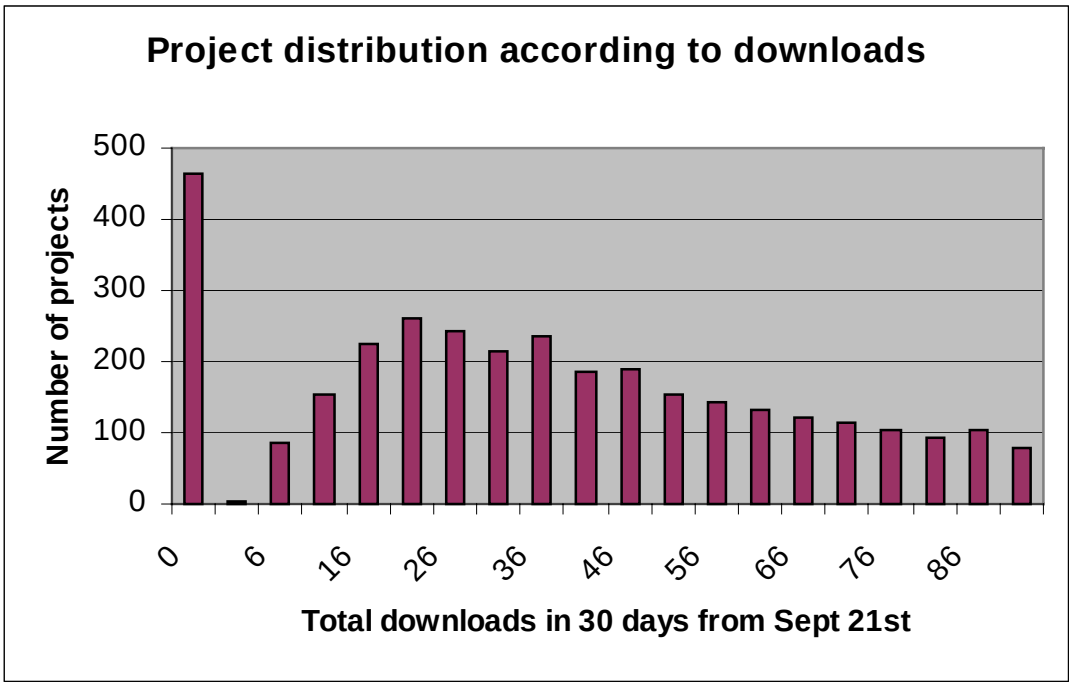


Figure 4: Distribution of projects with few total downloads

downloads do not use Sourceforge, or that such projects are inherently unstable i.e either they rapidly grow or they rapidly die off. We investigate whether this is the case in the next section.

Differential behaviours of segments of the cross-section

In this section we examine firstly the relative growth in downloads for projects with various numbers of downloads. Are projects with low numbers of downloads more likely to grow or to shrink than other projects? To measure the relative growth of each project, we divided the slope of the best fit line through the downloads of 30 days from September 21st, by the mean number of downloads in these 30 days. Averaging these relative growth figures over bins of ~200 projects and plotting this relative growth against the log of the mean number of downloads, we see clearly in figure 5 that low download projects grow faster than high download projects. (This does not seem to be a quantization effect i.e. due to number of downloads being an integer, since such an effect should not bias the direction of growth.)

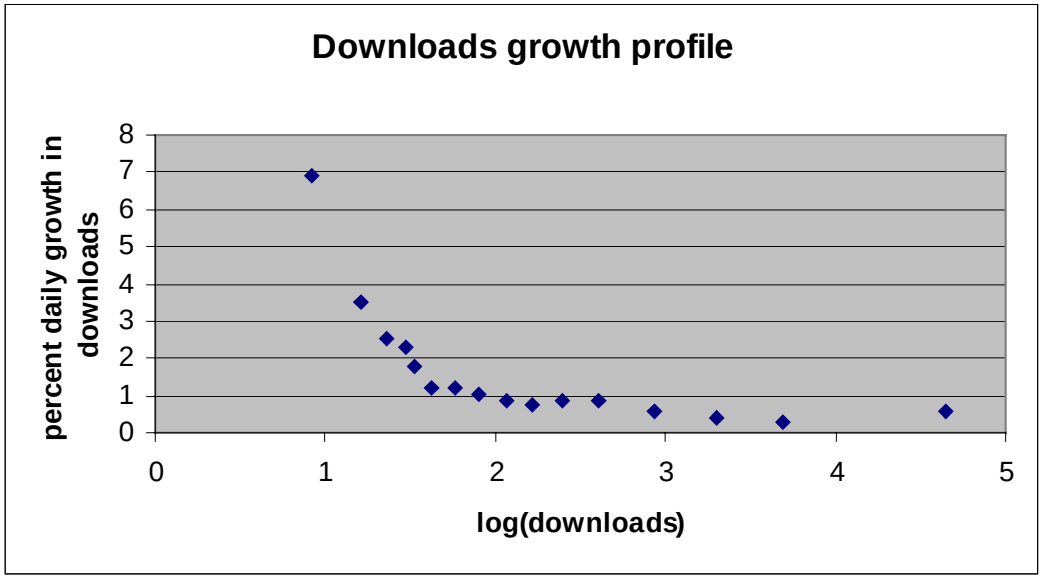


Figure 5: Growth profile of projects with differing numbers of downloads

It is worth noting that although the relative growth of projects was stronger for projects with few downloads, the overall trend in total downloads in the period observed is upwards. It may be that the correct conclusion is that overall trend in downloads over all projects is amplified in the projects with few downloads. To test this, we would need data for a period when the overall number of downloads from Sourceforge was declining[4].

The second issue we consider in this section is whether projects with low numbers of downloads are more unstable. Figure 5 has shown that projects with low numbers of downloads on average grow more strongly than those with high numbers. Is this true of all such projects, or does this average disguise a wide variety of outcomes? Intuitively, it is to be expected that a project with a low number of downloads is more easily influenced by external events. In figure 6 we plot the standard deviation of the

[4] If such a period does not exist, then the hypothesis could still be tested by looking at periods of differing overall growth in downloads from Sourceforge.

number of downloads over the 30 days from September 21, normalised by the mean number of downloads. As expected this show that projects with low numbers of downloads are subject to greater variations in downloads.

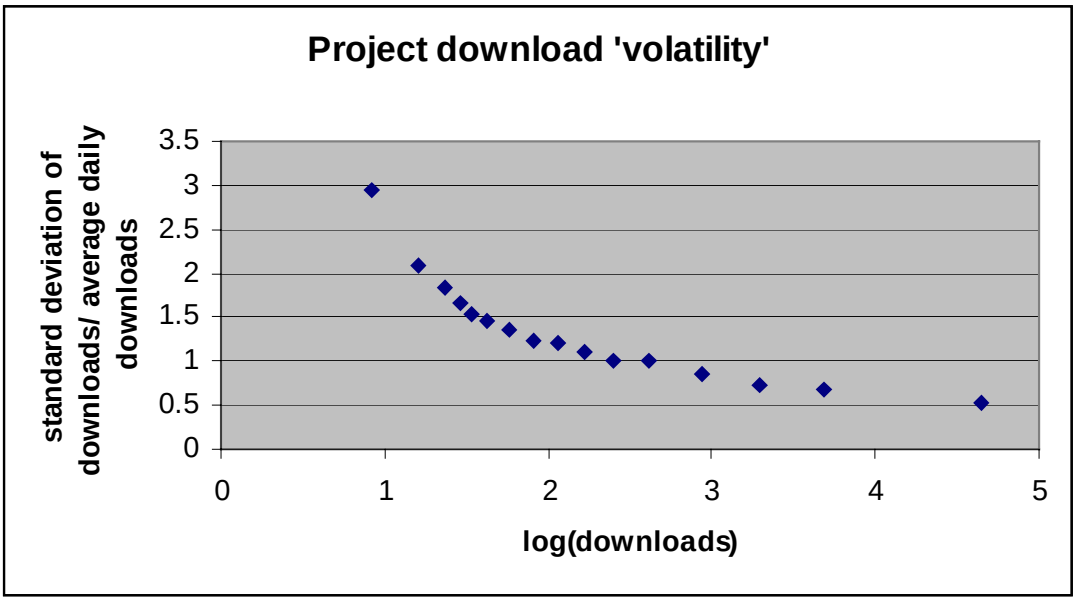


Figure 6: Profile of project download 'volatility'

Conclusions and further work

In this paper we have characterised the distribution of projects according to the number of downloads they receive. The open source projects most frequently studied (such as Linux, Apache, Mozilla) lie at the extreme end of a highly skewed distribution. Although studying these may be a fruitful approach in identifying 'best practice', it is potentially misleading since the problems and solutions found in the running of these projects may be inappropriate for most projects. Studying instead the median projects may prove useful.

In setting up a site to host open source projects, it is reasonable to assume that the resource requirements of the various projects will also be distributed according to a Pareto distribution. This is helpful in planning support for these projects.

The questions considered and answered in this paper bear further study, particularly against different samples of Sourceforge data. Given the richness of the data set there are also many unasked questions – readers are invited to download the dataset from the CTM website and experiment.

References

Adamic, L. A. and Huberman, B. A. (2000). "The nature of markets in the World Wide Web." Quarterly Journal of Electronic Commerce 1(1): 5-12.

Bak, P. (1997). How nature works : the science of self-organized criticality. Oxford, Oxford University Press.

Kienzle, R. (2001). "Sourceforge preliminary project analysis." available online at http://www.osstrategy.com/sfreport

Koch, S. and Schneider, G. (2000). "Results from software engineering research into open source development projects using public data." available online at http://opensource.mit.edu/papers/koch-ossoftwareengineering.pdf

Mockus, A., Fielding, T., et al. (2000). "A case study of open source development: the Apache server". 22nd International Conference on Software Engineering, Limerick, Ireland.

Pareto, V. (1896). Course of Political Economy. Lausanne.

Raymond, E. S. (1999). "The cathedral and the bazaar" in The cathedral and the bazaar : musings on Linux and open source by an accidental revolutionary. Sebastopol, CA ; Beijing, O'Reilly.

Richter, C. F. (1958). Elementary seismology. San Francisco ; London, W. H. Freeman.

Scholl, K.-U. (2001). "RE: How are statistics calculated?" posting to discussion forum available online at http://sourceforge.net/forum/message.php?msg_id=222302

Wheeler, D. A. (2001). "More than a gigabuck: estimating GNU/Linux's size." available online at http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

Zipf, G. K. (1949). Human behaviour and the principle of least effort : an introduction to human ecology. Cambridge, Mass., Addison-Wesley Press.

Goal-Diversity in the Design of Dependable Computer-Based Systems

A. T. Lawrie and C. B. Jones
Department of Computing Science
University of Newcastle University NE1 7RU
February 2002

Abstract

This paper sets out an argument for experimenting with some aspects of the "Open Source" development style in the creation of "Computer-Based Systems". The particular objective is to find ways of increasing the "Dependability" of systems. The most interesting facet of Open Source development in this connection is the use of multiple developers to introduce diversity into the design process. In addition to relying on the fact that no two developers are identical, the suggestion here is that asking each developer to emphasize different goals might result in solutions whose comparison and combination could increase dependability.

Keywords

COMPUTER-BASED SYSTEM, DEPENDABILITY, DESIGN PROCESS, DIVERSITY, FUNCTIONAL GOALS, GOAL-DIVERSITY, HUMAN-DIVERSITY, HUMAN-REDUNDANCY, NON-FUNCTIONAL GOALS, OPEN-SOURCE SOFTWARE PROCESS, REDUNDANCY.

1. Pre-Prologue

In the spirit of the proposed approach to development, the two authors (ATL, CBJ) have discussed many ways of presenting the ideas in this paper. In the end, two rather different styles are given below: in the Section titled "Prologue" (CBJ), an outline argument is presented; Sections 3 – 6 (ATL) expand on this argument and provide definitions of terms etc.

2. Prologue

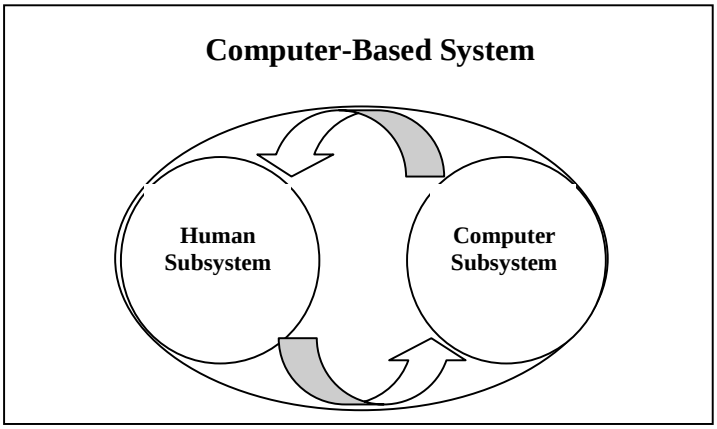
- The systems of interest are referred to as "Computer-Based Systems" to reflect the key role of people in the overall system.
- The aim is the creation of systems that are dependable.
- Dependability is often achieved by careful use of redundancy.
- Systems are created by (groups of) humans.

- The interest then is to use redundancy in the design and development process.
- Such redundancy is present in projects described as “Open Source” developments.
- Humans inevitably introduce an element of diversity into any task.
- Setting different secondary goals for designs can enhance diversity.
- Careful comparison and combination of diverse solutions could result in dependable systems.
- The whole discussion can be related to standard dependability notions such as the distinction between *faults, errors* and *failures*.

3. The Design Process of Computer-Based Systems

The phrase “Computer-Based System” (CBS) is used in a socio-technical sense that extends the chosen view of systems to include both humans and computers. This is illustrated in Fig 3.1.

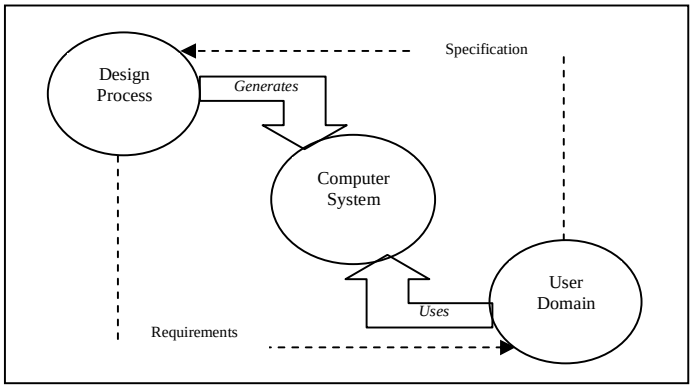
Figure 3.1 **Two-Subsystem CBS Definition**



This definition has ramifications since it interprets the human subsystem mainly in terms of the user-domain and emphasises the need to balance and consider the influences of both technology and social issues in their development [Mason & Willcocks, 1994]. Although this is critically important, it ignores the other key human role in the subsystem that concerns the design and development of the computer system.

To clarify this point, it is possible to illustrate how the view of a CBS could be extended to include three subsystems:[1] 1) the design-process, 2) the computer system, and 3) the user-domain. This view is illustrated in Fig 3.2.

Figure 3.2. **Three Subsystem CBS Definition**



When viewed in this way, a number of important considerations can be discussed regarding the design and use of CBS.

1. The two human subsystems can be considered as natural-systems and the computer-system can be considered as an artificial system
    - Natural systems can set and change their goals whereas artificial systems are designed by humans to fulfil human needs and therefore have their goals designed into their function[2].
2. Here then, we can appreciate the important dominating relationship between the design process and the computer-system. The design process generates the computer-system (see [Jones, 2002]).
3. The goals of the design-process will therefore have an overriding influence upon the eventual computer-system produced. Any errors or shortcomings that arise in the design process can manifest themselves as defects or deficiencies in the computer-system (see Section 4)

[1] It should be noted that the terms for design process and end-user domain include both their primary secondary environments (see: [Cooke & Slacke, 1991]). Additionally, the term computer-system includes both hardware and software components.

[2] Whether natural or artificial, most systems are goal-directed i.e. their behaviour is purposive towards achieving and maintaining some desired state [Heylighen, 2001].

4. The design specification represents a formal interpretation of the user-domains requirements, in terms of goals of the computer-system: misunderstandings or omissions will result in a computer-system that does not fully satisfy this purpose.
5. Over time, the user-domain may change its goals. Unless the design-process can change the computer system's "designed-in" goals accordingly, the computer-system will no longer fully satisfy the user-domains goals.[3]

3.1 Section Summary

The key point from this Section is that the design process is a key (socio-technical) subsystem in a CBS. Its players are responsible for the interpretation, creation, and evolution of the user-domains goals and expectations – via computer system development. Unfortunately, the existing CBS definition, at best, makes its inclusion implicit.

4. Dependability of Computer-Based Systems

The stance taken by the dependability community is to accept that, in any non-trivial software system, it is almost certain residual design faults will remain in the CBS [Randall, 2000]. Therefore, dependability is concerned with how such systems can be designed and developed to provide an acceptable continuity of service in the event of such faults giving rise to errors that may affect the expected delivery of service (see Section 4.1).

In order to help achieve this goal, a large body of theoretical knowledge and technical application has been combined into a conceptual framework. At the highest level, this framework identifies three principal factors influencing dependability.[4]

4.1. Impairments to Achieving Dependability

The impairments of dependability are concerned with the nature of problems in complex systems. These are faults, errors, and failures:

---

[3] Software evolution and maintenance is problematic and extremely costly. It may consume 80% of a software system's total life-cycle costs. Legacy systems represent computer-systems that can no longer fully satisfy the user-domains goals [Sommerville, 2001].

[4] Unless otherwise cited, Sections 4.1 through to 4.3 is with direct reference to [Laprie, 1992].

- **Faults:** are the hypothesized cause(s) of a system error. A fault becomes active when it produces an error
- **Errors:** are any part(s) of the system state that is liable to lead to a subsequent system failure. During system execution, the presence of active faults can only be determined by the detection of errors.
- **Failures:** occur whenever the delivered service no longer complies with the specification - this being an agreed description of the system's expected function and/or service.

4.2. Means to Achieving Dependability

There exists a collection of methods and techniques to promote the ability to deliver a service on which reliance can be placed, and to establish confidence in the system's ability to help accomplish this:

- **Fault prevention:** how to prevent fault occurrence or introduction into the CBS system.
- **Fault removal:** how to reduce the presence (number, or seriousness) of faults;
- **Fault tolerance**: how to provide a service complying with the specification in spite of faults;
- **Fault forecasting:** how to estimate the present number, the future incidence, and the consequences of faults.

4.3. Attributes to Achieving Dependability

System properties can be identified that help reveal certain desirable attributes of dependability. However, depending upon the users and application domain, such properties may be more (or less) emphasized. The main attributes are [Laprie, 1995]. :

- **Availability**: readiness for usage;
- **Confidentiality:** non-occurrence of unauthorized disclosure of information
- **Integrity:** non-occurrence of improper alterations of information
- **Maintainability:** the ability to undergo repairs and evolution

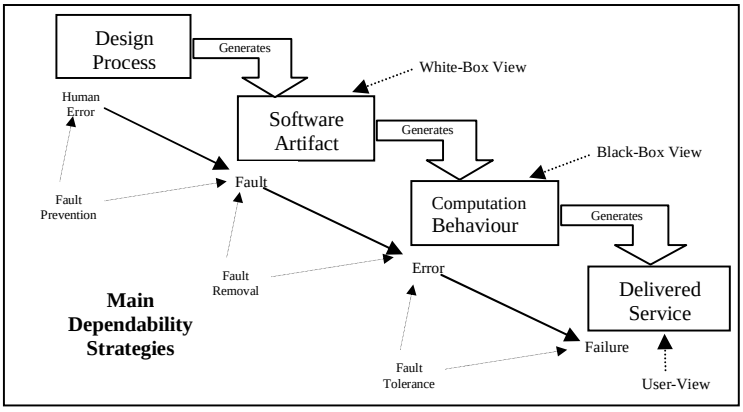
- **Reliability:** continuity of service;
- **Safety:** non-occurrence of catastrophic consequences on the environment

Furthermore, some of these may be compound attributes generated from other ones. For example, **Security** is seen as being the combination of attributes **Integrity, Availability**, and **Confidentiality.**

4.4. A CBS View of Achieving Dependability

By mapping these *means* and *impairments* to an extended version of the three subsystem CBS definition (cf. Fig 4.1), the dependability factors documented above can be illustrated to reveal the generic strategies available to achieving greater dependability of CBS during their design and development.

Figure 4.1 **Main CBS Dependability Strategies**



Firstly, as already highlighted in Section 3, Fig. 4.1 shows the dominating influence of the design process. Any human errors or oversights in the design process can quickly result in generating design faults in the software artefact[5], which then, during execution,

[5] The software artefact represents a "white-box" compositional view of the system…whereas the computational behaviour represents a "black-box" dynamic execution view of the system (cf. [Jones, 2002]). Of particular interest is that, since the user can only judge the provision of service from a black-box view of the CBS, perceptions of service failure may vary from one user to another. Equally, while faults and errors (in an absolute technical sense i.e. a fault manifests into an error which then affects the computational behaviour) may occur at the white-box and black box levels, it is only considered (i.e. judged) a failure if it becomes undesirably perceptible to the user's view of required service delivery.

become activated into errors and result in service delivery failures later during operational use. Secondly, the main dependability strategies employed are also shown, in terms of *fault prevention* in the design process, *fault removal* in the software artefact, and *fault tolerance* to intervene and limit errors causing service failure during operational usage.

Fault prevention, although an important *means* to achieving dependability, is seen as a *'general'* system and software engineering responsibility. While *fault-tolerance* is seen as a specialist area concerned directly with dependability. *Fault removal* may be employed by both fields – either at the process level (i.e. testing) or system level (i.e. fault-masking) (see: [Laprie, 1992], [Randall, 2000]).

Fault forecasting is not shown in Fig. 4.1 as it is not easily illustrated and is concerned with techniques (i.e. fault-injection) to ensure a representational *"coverage"* of the systems intended operational usage. Nevertheless, this *means* of achieving increased dependability of CBS is vital in terms of generating and verifying fault assumptions to determine the right *fault-tolerant* mechanisms to apply [Randall, 2000].

4.5 The Complexity Involved in Achieving Dependable CBS

Dependability is an inclusive concept but all of its aspects can have very subtle interactions and interpretations in any specific context. It can also be seen that the dependability attributes relate not to "what" functionality is delivered, but "how" that functionality is delivered. They therefore relate to the desirable non-functional qualities that promote CBS dependability (see: [Lamsweerde, 2001], [Jackson, 2001]). For example:

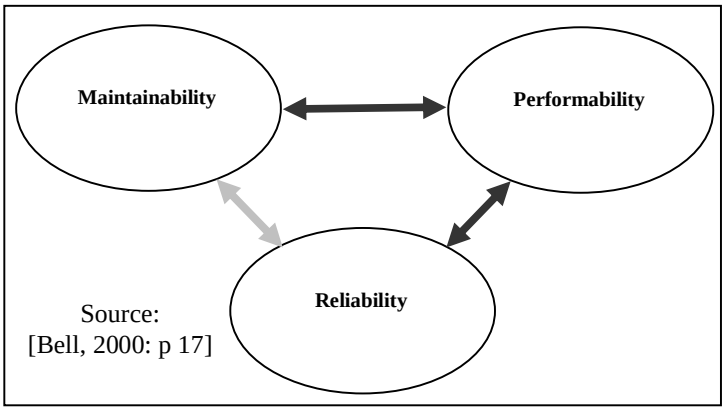
- **Reliability** of CBS is where the system behaviour *provides* (and only provides) that functionality needed for service delivery
- **Availability** of CBS is where *authorised access* to the required functionality can be provided whenever service delivery is needed.

Therefore, a failure in the CBS can occur without a failure in the service (but not vice versa – hence the 'generates' relationship in diagram 4.1)

- **Safety** of CBS is where the required functionality does not result in service delivery that can result in *damage* to the user or wider environment
- **Security** of CBS is where the required functionality does not result in service delivery that can be mitigated by *unauthorized* accidental or malicious *access* by others.
- **Maintainability** of CBS is where required service delivery *changes*, over time - by the user-domain, can be satisfied by equivalent functionality changes in the computer system.
- **Performability** of CBS is where the required functionality can be provided at a *time* needed to provide the delivered service. [6]

However, not only may these non-functional qualities vary from one specific application to another, but also, as [Bell, 2000] notes, in any design or development context the relationship between these non-functional attributes can be either complementary or conflicting (these relationships may however vary also between specific applications). A simplified and generic view is shown in Fig. 4.2:

Figure 4.2: **Complementary and Conflicting Goals in CBS Design**



Here, we can see that while reliability and maintainability may often complement each other during design and development, trying to achieve increased performance, also, may well conflict with, and militate against fulfilment of, the other two design goals[7].

[6] The "*close proximity*" of non-functional considerations relating to timeliness of functionality provision and the actual functionality provided in designing and developing real-time applications is highly visible and usually made explicit.

[7] The pursuit of increased structural flexibility offered by low data coupling, information hiding, and high cohesion to achieve maintainability goals will often also complement the achievement of reliability goals through increased faults and errors control and containment offered by program scope localisation. However, the pursuit of performance goals using such structures as inline-functions (c.f. Prata, 1995) and fast complex algorithms can often reduce program comprehension and produce unanticipated and

4.6 Redundancy and Diversity

One key general weapon to achieve Dependability is *Redundancy*. In many engineering applications, one can "over engineer" by, for example, introducing more strength than is required in materials or leaving more than minimum time. In situations where random failure or decay is the enemies of dependability, the use of multiple instances of a component (as in "Triple Modular Redundancy") can increase dependability. But design errors are not random and the execution of three copies of a flawed algorithm will do nothing to remove their inherent undependability. Redundancy can be utilised in software systems but it must be weeded to a way of achieving diverse solutions.

4.7 Section Summary

The key point from this Section is that, in complex CBS, residual design faults from the design process are almost inevitable. Awareness of this has given rise to a range of dependability strategies to prevent, remove, tolerate, and forecast faults, errors, and failures that may occur. To aid this goal, dependability employs a range of mechanisms that leverage redundancy and diversity in fault tolerant strategies. However despite the dominating role of the design process, concepts of redundancy and diversity are little used to facilitate the other dependability strategies or overcome the inherent difficulties of promoting the desirable attributes that embody dependable CBS.

5. Human Redundancy and Human Diversity

5.1 Human Redundancy

It has been argued in Section 4 that redundancy is fundamental to improving the dependability of engineered systems. In particular, it is a core feature of the many fault-tolerant strategies that have been applied to improve CBS dependability. What is less well understood is how forms of human-redundancy can be applied to the other dependability strategies of fault-prevention, fault-removal and correction, fault-coverage – involving the design process. Furthermore, it is legitimate to consider how human redundancy may also help promote the integration of important non-functional attributes that reinforce the dependability of CBS.

---

undesirable side-effects – that can result in faults, errors, and failures later. Therefore undermining potentially both maintainability and/or reliability goals.

5.2. Human Redundancy and Open Source

The emergence of open-source software development is a recent example of how human-redundancy can be employed in a highly decentralized way [Raymond, 1999]. Factors common (see [Gacek et al., 2001] for a discussion of different Open Source attributes) too many Open Source projects include development through geographically remote collaboration across the Internet. The main form of communication and coordination is usually via email, website domains, and central source-code repositories. Such products as the Linux operating system, the Kde desktop, and the Apache web-server have become highly successful with both industrial and domestic users and advocates.

The voluntary and indirectly subsidized nature of open-source development [Meyer, 2000] allows the potential for a level of duplication of development effort that could be rarely (if ever) matched and supported in traditional software development projects. Successful open-source projects can be supported by hundreds, and sometimes thousands, of contributing developers. Furthermore, [Yamouchi et al., 2000] indicates that a combination of voluntary effort and self-appointed work allocation supports the potential for duplication of effort and overlapping development. Such human-redundancy characteristics are often found in high reliability organisations where dedicated duplication and overlapping responsibilities are employed for crosschecking and verification [Viller et al., 1997]. This is, however, in stark contrast to traditional software engineering practice where economic constraints view human resources are economically limited and any duplication could result in increased costs and reduced commercial competitiveness. Traditional software development has increasingly been managed along concurrent engineering lines where project schedules are expedited by ensuring that the work is divided up and allocated on an individual task basis. [Weinberg, 1971] argued that this inhibits the ability to judge the quality of programs, as no source exists for generating comparative criteria.

5.3. Human Diversity

As noted in Section 4.6, software redundancy also requires diversity. Therefore, implicit in the discussion in Sections 5.1 and 5.2, is the fact that human redundancy is valuable because humans have a degree of differentiation. For example, humans have different intellectual abilities, experiences, knowledge, and personalities [Westerman et al., 1997]. If this were not the case, human redundancy – in terms of task duplication and responsibility overlapping, would be of no value since the redundant human resources would always reach the same view and make the same mistakes. Despite this, the levels of diversity required in developing software systems are very high and even with attempts to 'force' diversity within the task environment [Popov et al., 1999] humans are still prone to common-mode-failure[8]. However, this paper is not concerned with such issues[9], but instead, focuses upon how human resources could be reorganized within the design process to employ human redundancy and diversity to increase the potential of the other existing means of dependability such as fault prevention, fault-removal, fault-coverage, and design for dependability. Three areas are explored in Sections 5.3.1 to 5.3.3, which also draws upon the open-source development process where appropriate.

5.3.1 Increased Fault-Removal and Correction

One of the most visible benefits of the open-source development process is its increased bug finding ability through massive peer reviews of submitted source-code. This was characterised by [Raymond, 1999] in his seminal paper on evaluating the open-source software development approach as:

> *"Given enough eyeballs, all bugs are shallow." (p. 41)*

This informal approach to using human diversity for bug finding was, however, exemplified much earlier by [Weinberg, 1971] in his "egoless programming" philosophy.

---

[8] This is where two (or more) developers make the same human error during system development resulting in multiple version failure under the same functionality demands (cf. [Knight & Leveson, 1986]).

[9] Although we do not wish to imply that design and cognitive diversity for minimizing common-mode-failure is unimportant.

To return to the main strategies of achieving dependability in CBS (see Fig. 4.1), this example demonstrates how the design process can improve the dependability of CBS through leveraging existing cognitive diversity [Westerman et al., 1997] for increased fault detection, removal, and correction before deployment.

5.3.2 Increased Fault-Coverage

A much less visible characteristic of open-source development is the belief that, because of the voluntary nature of open-source, developers naturally "gravitate" to software development work they are naturally interested and/or already knowledgeable in performing [Lang, 2000]. This, combined with the reality that open-source developers are also users of the software they develop [Gacek et al., 2001] reveals that open-source developers have an intrinsically enhanced understanding and knowledge of the user-domain in which the software will be deployed. Consequently, this improves the particular developer's ability to anticipate potential usage exceptions, and likely faults, errors, and failures that the system may be susceptible too once in operational use [Randall, 2000].

On a less positive note, the open-source development approach appears only to develop software where there are well-established existing systems to clone/improve upon or where the development knowledge required is well exposed [Meyer, 2000]. This seems to limit major interest to the development of such systems as commercial-of-the-shelf applications (e.g. desktops, word-processors, office-applications, RDMS etc) or systems and systems development type software (e.g. compilers, operating systems, programming IDEs etc). Therefore, while it is possible that the open-source software development approach may be capable of developing such systems more dependably, it highlights that the open-source software process may be unworkable for development of software systems for specific application domains like business applications, process control software, or medical information system [Gacek et al., 2001]. It appears that, in contrast to traditional development approaches, open-source development is not subject to the additional requirements engineering challenges with which traditional software engineering is often faced.

However, influenced by the open-source phenomenon, [Anderson, 1999] carried out an experiment in massive parallel requirements engineering to derive a security specification for a (notional) national lottery system. He reported positively on how human diversity can be used to increase the reliability of computer-based systems through enhanced specification completeness and consistency checking. His findings are consistent with social constructivist views that different confirming observations that support each other increase the reliability of what is perceived [Heylighen, 2001].

While the open-source process indicates that greater fault-coverage can only be achieved though human redundancy and diversity in known or well-established user-domains, [Anderson, 1999]'s example possibly indicates otherwise. It suggests that such an approach could be utilized as a fault-prevention approach to decrease the risks of design faults by ignoring or omitting important user-domain details in deriving CBS specifications.

5.3.3 Increased Problem-Solving and Solution Finding

A much more ambiguous possibility, in open-source development, is the increased potential for problem solving and solution finding through massive forms of human redundancy and diversity. In his review of formal and informal design philosophies [McPhee, 1997] noted that the benefits of the informal route was the increased learning and specific knowledge acquisition that an explorative and iterative approach to design offered. Such views are reinforced by the experiences of [Raymond, 1999] when he designed the open source product "Fetchmail". During the project he became convinced that having many developers scrutinize source-code designs can result in someone helping you reframe the problem and simplify the design solution. He noted that:

> *"…It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too…at a higher level of design, it can be very valuable to have the thinking of many co-developers random-*

walking through the design-space near your product…exploration essentially by diffusion…This works very well. " (p. 47-52).

Software design and development has long been recognized as one of the most complex tasks imaginable [Brooks, 1995], [Glass, 1998]. This is because a designer is often faced with a multiplicity of design goals to accomplish [Weinberg, 1971]. In such situations, studies have shown that, as a result of this complexity, developers will focus on the most prioritized goals through treating other important system attributes as "free-variables" to be traded-off in their achievement [Weinberg and Schulman, 1974]. However, it is suggested here that the open-source software process is 'geared' towards openly resolving such technical conflicts. A justification for such a view is that open-source software development is driven by, and relies upon, good developers having ongoing "self-interest" to continue supporting the product [Sanders, 1998]. To achieve this, the open-source approach must be focused upon searching and finding appropriate design solutions to resolve such conflicts in order to accommodate the interests of the majority[10]. A good example of this is discussed in [Pettit et al., 2001]. They argue that the reason why the Linux kernel was designed to support dynamically linked modules was through a very large group of people pursuing their own individual interests of wanting to add or remove large sections of functionality in a convenient manner to suite their own personal and technical needs. In short, they believed the stimulus for such a solution was because:

"Linux was made by a very large group of people with a very diverse set of objectives." (p. 40).

Although this is a much less certain attribute of open-source software development, there are indications that a combination of massive human-redundancy and diversity of individual development goals helps to overcome the inherent complexity of software development by supporting increased problem reframing and solution finding. This is

[10] The Apache "shared-leadership" project is a good example of how design is the result of accommodating and resolving design solutions through majority consensus [Fielding, 1999]. Furthermore, it is suggested that the inability to resolve technical design conflicts can often result in "code-forking" of one project into two different design directions (cf. [Moody, 2001]).

achieved not only through increased design ideas generated, but also through greater visibility[11] of the many design conflicts and trade-offs regarding important system attributes that developers make during software development. To return to the theme of Section 4, concerning the important desirable dependability attributes that promote dependability of CBS, it is suggested that leveraging human redundancy and diversity in a similar way may improve the integration of such qualities in CBS design.

5.4. Section Summary

The key point of this section was to explore how human forms of redundancy and diversity may be employed within the design process to promote the other dependability strategies of fault prevention, removal, coverage, and design for dependability. It has been identified that human forms of redundancy are relevant for CBS design as they are inherently differentiated and therefore bring a level of diversity required for software development. As examples, we have identified human redundancy and diversity issues raised by the recent phenomenon of Open Source Software development. The Open Source approach begins to suggest that human redundancy and diversity can be utilised favourably for all three dependability strategies. Furthermore, there are indications that human redundancy and diversity increase problem solving and solution finding through the leveraging of others design views/ideas and conflict resolution of self-orientated development goals.

6. Goal-Diversity

6.1 Engineering Human Diversity

As discussed in Section 3, the design process is a natural system that sets and changes its own goals. Furthermore, it has a dominant affect on the computer systems it creates. Since a CBS is an artificial system that has its goals designed-in, the goals pursued by the developers will have an overriding influence on the eventual system produced. Any omissions, conflicts, or mismatches are likely to result in defects and deficiencies in the system it eventually generates. Therefore, one way to improve control and help reduce

[11] See also [Lamsweerde, 2001]: the review of "goal-orientated requirements" and the potential for goals to making requirements and design conflicts more identifiable.

such problems is to influence the goals of the developers through careful goal setting[12] of required design goals to be achieved and maintained throughout the development of CBS.

Taking the lead from existing approaches of "forced" diversity [Popov et al., 1999], cognitive-engineering [Westerman et al., 1997], along with goal-setting in management theory [Latham & Locke, 1979], diversity can be engineered through goal-diversity to achieve purposive design behaviour that deliberately predisposing individual developers to pursue and maintain different goals that promote the needed functionality of the system along with the desirable non-functional attributes that promote dependability of CBS.

The following Section from 6.2 to 6.4 discuss the specific benefits that may be derived from such an approach – along with initial ideas of how it could be implemented.

6.2. Goal-Promotion

Non-functional attributes of the system are more likely to be fulfilled when they possess a close proximity to the required functionality. For instance, system performance, in terms of its timeliness of functionality, is so closely linked to required functionality in 'real-time' systems that this non-functional attribute will be explicitly promoted as an important development goal to be achieved during the design process. However, maintainability of the CBS is often so remote from present required functionality that it is more likely to remain and implicit user-domain expectation of the system in the future. When this happens achievement of the non-functional goal is ultimately left to the responsibility of the particular developer(s) involved – with regards to their discretion, skill, and professionalism. As we have seen, the complexity and multiplicity of the software design task will often result in such implicit goals being traded-off for more explicitly demanded ones. Consequently, the integrity of the CBS, in terms of its maintainability attribute, may well be compromised and will not become an issue until

[12] The value of goal setting has been long recognized for its ability to create purposive goal-directed behaviour and attitudes in both individuals and groups see: [Latham & Locke, 1979] i.e. management by objectives, [Demarco & Lister, 1987] i.e. goal alignment in teams, [Covey, 1992] i.e. personal improvement.

years later when the user-domain requires the system functionality to be changed, corrected, or enhanced. Therefore, if one wants a CBS to possess the desirable non-functional attributes that support dependability, such attributes must be explicitly promoted as desired non-functional goals of the system.[13]

The idea of goal promotion is to achieve both increased diversity and coverage of important non-functional attributes of CBS by utilising human redundancy to implement the same functional specification as a primary goal, during development, while deliberately predisposing individual developers of the contributing group to take responsibility and ownership of an important non-functional attribute as their own secondary goal during implementation. This not only reduces the complexity of the task, in terms of removing the multiplicity of goals an individual developer must consider, but also ensures sufficient coverage of important desirable attributes of the CBS.

6.3. Iterative Design Phases

The notion of goal-promotion from Section 6.2 is an important prerequisite for this phase to work correctly - as it relies upon continued goal-ownership of secondary non-functional goals throughout the iterative stages of design review and goal relationship identification (see: Sections 6.3.1 – 6.3.2). This ensures that individual developers continue to defend their own secondary goals and critically evaluate and question the other developer's secondary goal design decisions throughout the design and development of the system. As already discussed in Section 5.3.3, it is proposed here that a number of iterative design and refinement phases (see Sections 6.3.1 – 6.3.2) will promote increased learning and problem domain understanding. This is necessary with such an approach as it was already noted in Section 4.5 how the non-functional attributes of dependability have subtle interactions and emphasis given any specific application. Therefore, because of this novelty, it will be necessary for the developers to acquire a deeper understanding before a specific set of evaluation criteria will emerge that will allow the developers to be able to judge the design rationales to assess more accurately

[13] It appears from Bill Gates recent email (see: [Boutin, 2002]) that Microsoft are placing an increased value and priority on such non-functional dependability attributes of Availability and **Security** in the future in order to achieve and realise their design goal(s) of ***"Trustworthy Computing"*** for .NET.

whether a particular non-functional goal has been achieved or not (or to what degree it has been achieved or not).

It is also suggested here that a deeper understanding of the problem domain will also support dependability strategies of increased fault-prevention and fault-coverage through the increased sensitivity of developers to other developers design decisions and fault, error, failure assumptions. This will allow them the increased ability to anticipate potential defects and deficiencies of the CBS in advance of its deployment.

6.3.1 Peer-Review

The benefits of peer review inspections of program design have already been discussed in the context of human redundancy and diversity and the open-source approach. The "many eyes" affect not only supports fault detection and removal strategies of dependability but, (as noted in Section 6.3 above) in the context of this approach, it also increases the visibility of the overall development through stimulating greater insight through reviewing diverse approaches to the design problem by other contributing developers.

As an initial attempt at envisaging how it should be undertaken, it is suggested that the beginning of the design phase should be undertaken by developers working in complete isolation from each other until they have a deeper overall knowledge of the design problem.[14] After an initial attempt at the design each developer then reviews all of the other developers' diverse attempts at the design problem. Along with promoting fault detection and removal strategies the purpose of the review will be to identify important goal relationships that exist between the diverse designs. This is explored further in Section 6.3.2 below.

[14] The justification for this isolation is to preclude interruptions and negative group affects that may squash ideas and innovative approaches to the problem. Once all of the developers have a thorough understanding of the problem it is hypothesised that such influences will have little effect.

6.3.2 Goal Relationship Identification

One of the benefits of a goal-orientated approach is that it quickly allows identification of relationships that exist between goals [Lamsweerde, 2001]. As we seen from Section 4.5, non-functional goals can be related as being either complementary or conflicting. However, it is also suggested here, that in some cases, two goals – while not being complementary, may also not necessarily be conflicting either. In such circumstances, the two goals represent a different interpretation and emphasis during their implementation, and could be made compatible when these concrete specifics are reworked or re-factored.

As discussed in Section 6.3.1, during the review stages an important consideration for developers will be to recognise the important relationships that may exist between diverse versions. Where the individual developer believes that another developers non-functional goal is complementary or compatible to theirs they should redesign their program to implement it. However, where the developer believes that another developer's non-functional goal is conflicting with their own they should document their reasons why they think it cannot be integrated into their program. After redesign, by each developer, to integrate complementary and compatible non-functional goals of the other developers the design review process begins again - only this time with the redesigned and reintegrated programs. The benefit of such an approach is that it allows a synthesis of non-functional goals to be iteratively integrated and reviewed. More importantly, however, it also begins to unearth particular conflicting aspects of the design problem - where concentration of the contributing developers should be focused. However, such technical conflict shouldn't be viewed negatively, as the discussion of open-source suggests in Section 5.3.3, it can stimulate the search and identification of higher-level design solutions.

6.4. Conflict Resolution

It is inevitable, however, that even with iterative stages of review and redesign to find solutions to outstanding non-functional goal conflicts, some goal-conflicts will remain intractable and unresolved. In such situations the prioritization of one goal over another will need to take place. Goal-orientated approaches are highly valuable in such situations as they facilitate the evaluation of goals with regards to their priority status [Lamsweerde,

2001]. Furthermore, since the emphasis upon certain desirable attributes of dependability is reliant upon the specific user and application domain, it will be necessary to get the input from the user-domain to explicitly decide which dependability attributes are more important than others. Here, also, goal-orientated approaches are useful as they permit an interface for discussion between the technical aspects of the design process and the business/user/management aspects of the user-domain[15] [Lamsweerde, 2001].

The important point to this is that the user-domain has a direct and explicit input into the prioritisation of what dependability attributes are considered more (or less) important. This makes non-functional prioritisation an explicit user-domain consideration – rather than an implicit, and often unfulfilled expectation, which is often determined discretely by a single developer during system development. Therefore, such an approach has the additional advantage of making the dimensions and attributes of CBS dependability (and potentially undependability) known to the user-domain.

6.5 Section Summary

The key point of this section is that we can engineer diversity by predisposing developers to pursue certain desirable goals. This is vitally important since the goals of the process have an overriding influence on the eventual dependability of the CBS. Goal promotion not only helps reduce the complexity of the task faced by developers, but also ensures sufficient diversity and coverage of the important attributes that support dependability of CBS. However, because different attributes are important in different user and application domains' it is important to apply an iterative approach that allows progressive problem understanding, synthesis, integration, and evaluation criteria of important non-functional attributes to emerge. Finally, it is vitally important that irreconcilable goal conflicts are explicitly prioritised within the user-domain in which the CBS will operate. This allows the limits of the CBS dependability dimensions to be known.

[15] Although outside the scope of this paper, in real-life application, this goal-oriented approach to design-diversity, would probably require an equivalent goal-orientated approach to requirements engineering [cf. Lamsweerde, 2001].

7. Future Research

The ideas presented by the authors are explorative, in nature, and a number of experimental and empirical possibilities have been discussed. It should first be noted that, due to the inherent range of programming abilities and non-linear nature of complexity scaling in software engineering, experiments in software design are inherently difficult to control and generalise from. However, the interdisciplinary research project (i.e. DIRC see acknowledgements in Section 9) in which this research will be carried out has resources and facilities that will be useful for conducting such research. Secondly, a number of possible approaches include:

- A pilot experiment into the benefits of increasing human redundancy in fault detection. It is conjectured that a law of diminishing returns will be experienced.

- A pilot experiment into "goal-diversity" using students initially and later attempts to confirm any results through observations within an industrial setting. It is thought that the initial experiment would utilise two groups attempting the same design problem and giving them different (non-functional) goals to pursue (i.e. performance vs. maintainability). Eventual solutions would then be exchanged, discussed, and then merged. Depending upon the outcome of such an experiment, it may be repeated later using three such groups and objectives. With regards to a later industrial investigation, it has been suggested that a study should observe how they manage non-functional goals within the design/development process, in terms of which goals are achieved, maintained, and or compromised.

8. Conclusions

This paper has argued that redundancy and diversity are important ways to achieve dependability in CBS. However, despite the fact that the design process is mostly responsible for the residual design faults that are to be expected (and ultimately tolerated) in any non-trivial CBS. There has not been the same exploitation of redundancy and/or diversity principles, at the process level, that has been practiced and pursued by fault-tolerant strategies at the computer-system level. Nevertheless, the nature of the Open-

Source Software development process begins to suggest that massive forms of human redundancy and diversity can help improve the dependability of CBS through leveraging them for other dependability strategies such as fault prevention, removal and coverage. There are also indications from the Open Source approach that human redundancy and diversity can promote increased problem solving and solution finding via multiple views/ideas and the resolution of self-orientated goal-conflict. In recognition of these Open Source influences, and the important influence of design process goals, the notion of goal-diversity was discussed. It has been suggested that such an approach may improve the dependability of CBS through a process of deliberately predisposing developers to pursue diverse design goals. It is suggested that this may lead to greater levels of design diversity, coverage, synthesis, solution finding, and integration of desired non-functional attributes that promote the dependability and trustworthiness of CBS.

9. Acknowledgements

The research of both authors is supported by the EPSRC grant to the Interdisciplinary Research Collaboration on Dependability of Computer-Based Systems; that of ATL is also funded by an EPSRC PhD studentship. Both authors are grateful to many colleagues in "DIRC" for fruitful discussions but would like to single out Denis Besnard and Carles Sala-Oliveras for particularly intensive interchanges on the subjects reported here.

10. References

Anderson, R., (1999) *"How to Cheat at the Lottery (or, Massively Parallel Requirements Engineering),"* in Proc. Computer Security Applications Conference, Phoenix, AZ,

Bell, D. (2000). *Software engineering: A programming approac*h. 3rd edition. Addison-Wesley, U.K.

Boutin, P., (2002) *Bill Gates Email on Trustworthy Computing*. Online at URL: http://paulboutin.weblogger.com/stories/storyReader$155

Brooks, F. P. (1995). *The mythical man month: Essays on software engineerin*g. Anniversary Edition. Addison-Wesley, New York, NY.

Cooke, S. & Slack, N. (1991). *Making management decision*s. Prentice-Hall, UK.

Covey, S. R., (1992) *The Seven Habits of Highly Effective People: Powerful Lessons in Personal Change*. Simon & Schuster Ltd. London. UK.

Demarco, T., Lister, T., (1987) *Peopleware: Productive Projects and Teams*. Dorset House Publishing. New York. USA.

Fielding, R. T., (1999) *Shared Leadership in the Apache Project.* Communications of the ACM. Vol. 42, No. 4. April 1999. pp-42-43.

Gacek, C., Lawrie, T., Arief, B. (2001) *The Many Meanings of Open Source*. Department of Computing Science, University of Newcastle-upon-Tyne, Technical Report CS-TR-737

Glass, R.L., (1998) *Software Runaways; Lessons Learned from Massive Software Project Failures*. Prentice Hall New Jersey, USA.

Heylighen, F.H., (2001) *Cybernetics and Second-Order Cybernetics* :in: R.A. Meyers (ed.) *Encyclopedia of Physical Science & Technology* (3rd ed), 2001. Academic Press, New York. USA.

Jackson, M., (2001) *Problem Frames: Analyzing and structuring software development problems.* Addison-Wesley Publishers. Harlow. UK.

Jones, C.B., (2002) *Providing a formal basis for dependability notions.* Department of Computer Science. University of Newcastle. UK. (to be published)

Knight, J.C., Leveson, N.G. (1986) *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming.* IEEE Transactions on Software Engineering. Vol. 12. No. 1. January., pp 96-109.

Lamsweerde, A. V., (2001) *Goal-Orientated Requirements Engineering: A Guided Tour*. Proceedings of Requirements Engineering 2001, 5th IEEE International Symposium on Requirements Engineering, Toronto, August 2001. pp 249-263.

Lang. R. (2000) *Open Source Software: Issues and Implications*. News @ SEI. Vol. 3. No. 1. Winter 2000., pp. 6-7. Online at URL: http://www.sei.edu

Laprie, J. C., (1995) *"Dependable Computing: Concepts, Limits, Challenges,"* in *25th* IEEE International Symposium on Fault-Tolerant Computing - Special Issue, pp. 42-54, Pasadena, California, USA, IEEE.

Laprie, J.C., (1992) (Ed.). *Dependability: Basic concepts and terminology — in English, French, German, Italian and Japanese,* Dependable Computing and Fault Tolerance. Vienna, Austria, Springer-Verlag

Latham, G.P., Locke, E.A., (1979) *Goal-Setting – A Motivational Technique That Works.* Organizational Dynamics, Vol. 8. No 2. pp 68-80.

Mason, D., Willcocks, L., (1994) *Systems Analysis, Systems Design.* Alfred Waller Publishing, Oxfordshire, UK.

McPhee, K., (1997) *Design Theory and Software Design*, Technical Report TR 96-26 (October 1996 : - Revised May, 1997) Department of Computer Science, University of Alberta, Edmonton, Alberta Canada.

Meyer, B., (2000) *The Ethics of Free Software*. Software Development Magazine. Online at URL: http://www.sdmagazine.com/articles/2000/0003/0003d/0003d.htm

Moody, G., (2001) *Rebel Code: Linux and The Open Source Revolution*. Allen Lane Penguin Press. Hammondsworth Middlesex. UK.

Pettit, K., Chen, S., Coffing, C., Ho, T., Brockmeier, J., Harris, A., (2000) *Suse Linux: Install, Configure, and Customize*. Prima Publishing, California. USA.

Popov, P., Strigini, L., Romanovsky, A., (1999) *Choosing Effective Methods for Design Diversity – how to progress from intuition to science*. Lecture Notes in Computer Science, Vol. 1698.

Prata, S., (1995) *C++ Primer Plus* (2nd Edition). Waite Group Press. California. USA

Randell, B. (2000). *Turing Memorial Lecture: Facing up to faults*. The Computer Journal, Vol. 43. No. 2. pp 95-106.

Raymond, E.S. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Rielly & Associates, Inc. USA

Sanders, J., (1998) *Linux, Open Source, and Software's Future*. IEEE Software. September/October 1998. pp 88-91.

Sommerville, I., (2001) *Software Engineering* (6th Edition). Addison-Wesley Publishers. Essex. UK.

Viller, S., Bowers, J., Rodden, T., (1997) *Human Factors in Requirements Engineering: A Survey of Human Sciences Literature Relevant to the Improvement of Dependable Systems Development Processes*. Technical Report CSEG/8/1997. Computing Department. Lancaster University. UK.

Weinberg, G. M. (1971). *The psychology of computer programmin*g. Van Nostrand Reinhold, London.

Weinberg, G.M., Schulman, E.L., (1974) *Goals and Performance in Computer*

Programming. Human Factors. Vol. 16. No. 1.pp 70-77.

Westerman, S. J., Shryane, N. M., Crawshaw, C. M. & Hockey, G. R. J. (1997). *Engineering cognitive diversity*. in F. Redmill & T. Anderson (Eds). *Safer System*s. Proceedings of the 5th Safety-critical Systems Symposium, Brighton, UK (pp. 111-120).

Yamouchi, Y., Yokozawa, M., Shinohara, T., Ishida. T. (2000). *Collaboration with Lean Media: How Open-Source Software Succeeds*. Conference Paper Presented at Computer Supported Cooperative Work Conference 2nd to 6th December 2000. Philadelphia., pp 329-338.

An Overview of the Software Engineering Process and Tools in the Mozilla Project

Christian Robottom Reis
<kiko@async.com.br>
Async Open Source
R. Santa Cruz 462 Sala 6
São Carlos, SP
Brazil 13560-780

Renata Pontin de Mattos Fortes
<renata@icmc.sc.usp.br>
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
P.O. Box 668, São Carlos, SP
Brazil 13560-970

February 8, 2002

Abstract

The Mozilla Project is an Open Source Software project which is dedicated to development of the Mozilla Web browser and application framework. Possessing one of the largest and most complex communities of developers among Open Source projects, it presents interesting requirements for a software process and the tools to support it. Over the past four years, process and tools have been refined to a point where they are both stable and effective in serving the project's needs.

This paper describes the software engineering aspect of a large Open Source project. It also covers the software engineering tools used in the Mozilla Project, since the Mozilla process and tools are intimately related. These tools include Bugzilla, a Web application designed for bug tracking, bug triage, code review and correction; Tinderbox, an automated build and regression testing system; Bonsai, a tool which performs queries to the CVS code repository; and LXR, a hypertext-based source code browser.

Keywords: open source software, free software, software engineering, software process, software engineering tools, bug tracking, nightly builds, code versioning.

1 Introduction

The Mozilla Project[42] is an Open Source Software [28] (OSS) project which is dedicated to developing the Mozilla Web browser suite. Since its creation in 1998, the project has attracted thousands of participants, and has arguably one of the largest communities working on an OSS project today[45, 15]. Mozilla is very important to the Web development and free software communities, mainly because it fills a perceived gap for an Open Source, standards-compliant Web browser (figure 1).

Although its main product is the browser, the Mozilla Project has a number of related subprojects. The browser is developed using a set of open technologies which compose the Mozilla application framework, a platform-independent suite of languages and libraries. These technologies include:

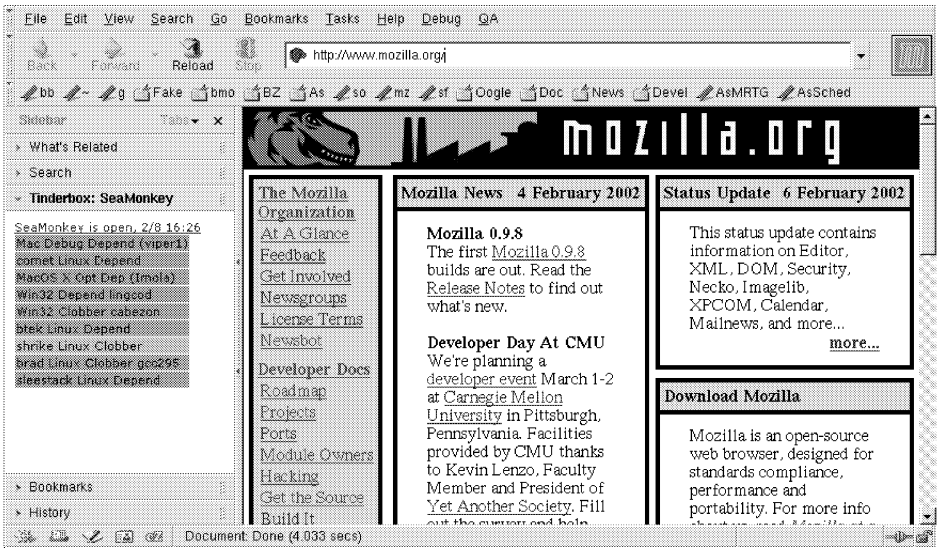


Figure 1: A screen capture of the Mozilla Web browser

- XUL, The XML User Interface Language, a cross-platform user interface description language
- XBL, the eXtensible Binding Language, a language used to modify the behavior of elements in documents
- Javascript, an ECMA-standardized language for scripting Web applications
- Gecko, Mozilla's cross-platform, embeddable layout engine

The framework is used as the basis for a number of commercial products[3, 2], including CiTEC Doczilla, Tuxia Nanozilla, IBM Warpzilla and the OEone Operating Environment (figure 2).

Apart from the projects directly related to the Mozilla browser and application platform, the Mozilla Project has also developed a number of software tools which support the community of developers which works on the main browser and framework projects. This paper will primarily discuss these software tools and the development process in which they are used.

1.1 About the Mozilla Organization and community

The Mozilla Organization[43] (mozilla.org) is a group which exists to support the development of the browser suite. mozilla.org is responsible for managing, planning and providing server resources to support the development of Mozilla. To a large extent, it is also responsible for developing and enforcing the Mozilla process, and acts as the final arbitrator between disputes over changes to the codebase.

The organization is composed of selected people from the community which act as managers and technical lead for the different Mozilla Projects. Each member of the organization is responsible for a Mozilla-related task, including Web site maintenance, documentation, architecture design and release engineering. There are currently 14 people listed as mozilla.org staff from a number of different organizations, including Netscape, Redhat and OEone[44].

mozilla.org is charged with leadership for the Mozilla Project, but it is important to realize that the actual work is performed by a large number of people who are not necessarily part of the organization itself, which are identified simply as the Mozilla community. The community consists of volunteers,

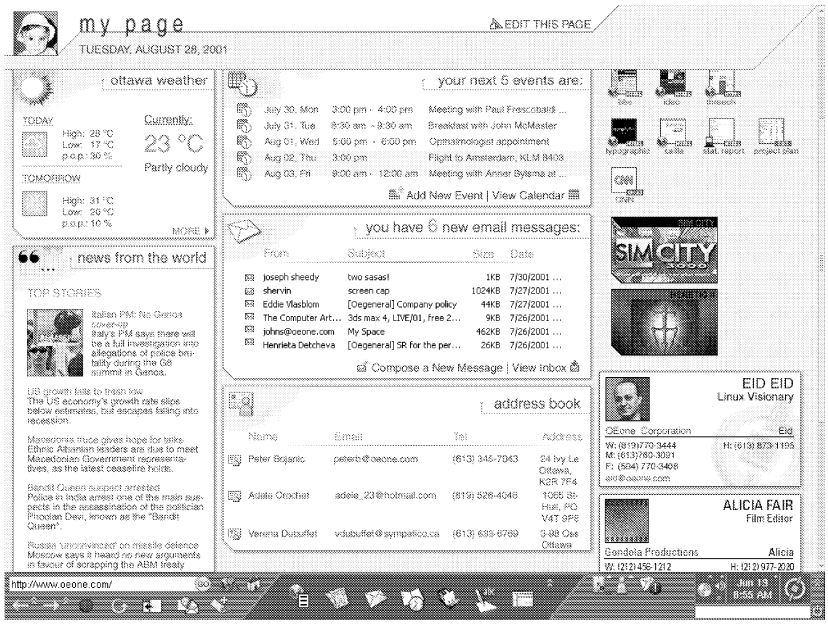


Figure 2: A screen capture of the OEone Personal Portal, a product derived from Mozilla source code.

paid contributors and mozilla.org staff.

Each of the Mozilla Projects are maintained and supported by different groups. Since participation is freely accepted and welcomed, it is hard to even identify the exact membership of a certain group beyond the fact that certain people work frequently on the same project. However, common contributors may be granted certain privileges over time (such as write access to the source control repositories) which can help identify the core members.

mozilla.org may also define a special role for developers and community members that are particularly skilled or committed to the project. These roles include super reviewer, module owner, Bugzilla component owner and driver. These roles are described in section 3.

1.2 Unique aspects of the Mozilla Project

Although the number of active OSS projects today is quite large[19], the Mozilla Project is an interesting target for OSS research for a number of reasons.

- The Mozilla codebase is one of the largest and fastest moving among OSS projects. Its size is comparable to the Linux kernel and larger than the whole XFree86 graphical environment: David Wheeler's 2001 study showed the browser to have just over two million lines of code[49], and the project has evolved much since then. If one makes a comparison with the age of the Linux kernel, now more than ten years old, and the X Window System (upon which XFree86 is based) which has been developed for over 15 years, the short time taken to develop such a massive codebase is quite startling.

- The original Netscape Navigator 5 codebase was donated by Netscape to mozilla.org [21], so there was a significant amount of pre-existing code at the time the Open Source project was officially started. This is not uncommon among OSS projects in general, but it presents significant changes to the "natural" design and requirements processes in a conventional, from-scratch project. However, it is important to note here that large parts of the original code were rewritten, as discussed further in section 2.

- The number of developers is high[45], many of them being directly paid by Netscape, OEone, Sun, IBM and other companies that fund development of the browser suite and framework. The number of volunteer contributors is also remarkable: Bugzilla, the bug-tracking software used for Mozilla registers over 16,000 active users, over 14,000 unique bug reporters, and over 1,000 "trusted" people who can perform tasks such as confirming and editing bugs[15]. Table 1 counts the active contributors that made changes to the code repository over the past 8 months[1].

	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb (incomplete)
Developers submitting code	143	160	152	157	158	150	159	104
New developers submitting code	2	11	7	5	16	5	12	1
Total code submissions	1577	1892	1997	2355	2348	1594	2083	466

Table 1: Statistics for source code submission from July 2001 to February 2002[45].

- Much of the development infrastructure used in the Mozilla Project was developed to deal with its massive organizational requirements. Software engineering tools to support distributed development were a requirement and developed since day one of the project's existence. These tools are considered by some to rival the significance of the browser suite as a Mozilla product[1].

- The Mozilla Project aims to create a polished, easy-to-use application for end-users of widely varying computer skills, whereas the majority of OSS projects concentrate on applications where the developer is also a domain expert[30, 29]. This presents requirements that most OSS projects never face, and developers must often achieve consensus on controversial issues with which they may not be familiar. Easy installation and usability, for instance, are two examples of potentially unfamiliar territory to developers.

These aspects contribute to a colorful and interesting project scenario, which is constantly evolving in public view. The project generates an astonishing amount of software engineering data, and because the Mozilla software tools were designed for recording and allowing convenient access to this data, it becomes a straightforward proposition to study the project and its evolution.

2 A brief background on the Mozilla process

Work started on Mozilla in March 1998, using the original codebase which was donated by Netscape Communications. Because of this, and because Netscape already intended to use Mozilla as a basis for a commercial browser, many of the requirements had been determined by the original code and documentation developed by Netscape[36]. The other main requirement established by the steering group was compliance to HTML standards, which is a very noteworthy intention, since the W3C had yet to see an organization devote itself to strictly implementing its standards.

Since it was essentially the same codebase, the original design matched the Netscape Navigator 5 design[2]. After less than a year of development, however, the technical lead of mozilla.org came to the somewhat controversial[31, 47] conclusion that the original codebase would prove impossible to evolve to suit the requirements of a standards-compliant Web browser[16]. Some code modules were completely rewritten – such as the layout engine, which needed to be thoroughly changed to support the new technologies which had been standardized by the W3C. The rewritten code has resulted in a next-generation

[1]Note that only people who directly submitted code to the repository are counted. Contributors without write access to the repository need to ask other developers, and therefore are not included.

[2]Actually, certain components were removed because of licensing and legal issues[21].

browser, but the cost in time has been high[22]: after four years, version 1.0 is scheduled for release in April 2002[17].

The rewriting activity also changed the project's effective maturity. It shifted from being a project devoted to browser maintenance, with gradual enhancement and refactoring, to an "initial implementation", where new features are specified and designed from scratch. This change, associated with a desire for high modularity, produced the many new technologies which form the actual browser platform: Gecko, XUL and XBL, for instance, were designed during this phase. From our analysis, it also had a negative impact on the stability of the code and its API, since this new technology had yet to be broken in and tested. Release 1.0 is expected to finally consolidate these technologies into a stable platform for development.

Because there has always been an implicit agreement that process would be essential if Mozilla was to be successfully developed[37], a great deal of thought was given to developing and documenting the Mozilla software process, which in part evolved from Netscape practices. This evolution has been guided by a need to suit the open community which has been attracted to the browser. Coordinating source code changes from a large number of concurrent, distributed developers, has generated a number of demands in both process and support tools:

- effective version control
- a well-defined protocol for integrating source code changes
- a high degree of accountability for people who integrate this code
- high modularity
- custom development tools
- good communication channels

The following section will describe how these demands have been integrated into the Mozilla software process.

3 Aspects of the Mozilla software process

The Mozilla Project has a very broad software process, which encompasses many different aspects of development. We will cover in this section the aspects that we consider most relevant, though it is not a complete reference. Further and more in-depth description can be found on the Mozilla Web site[42].

3.1 Modularity and module ownership

The Mozilla browser is developed using the Mozilla application framework[3]. One of the characteristics of the design of this framework is that it is highly componentized[4] due to the use of a cross-platform component library called XPCOM[41]. This design is by nature modular, and parts such as the Javascript engine, the runtime libraries, and the framework itself can be (and have been[3]) reused independently of the browser to develop other products. The high modularity also permits developers to concentrate on areas of the code without needing to understand the entire architecture; it allows for a gradual learning curve, which is important for project newcomers. Modularity, finally, is what makes possible development in spite of the natural concurrency presented by a distributed developer community.

[3]The browser itself is a front end written mainly in Javascript, XUL and XBL, and interpreted at runtime. The framework constitutes the back end code and is written in XPCOM and C++.

[4]Note that code components are not the same as Bugzilla components.

Most code modules in Mozilla have one or more associated components in the Bugzilla bug tracking tool. Each Bugzilla component has an owner, which is the default assignee for new issues reported, and a Quality Assurance contact, which oversees the progress of these issues and can intervene as needed to help the developer (Quality Assurance is further discussed in section 3.6).

If an issue interests another developer, he is allowed to assign the task to himself and propose fixes. He will usually coordinate with the former owner to avoid duplicated or mismatched work. In this way, responsibility is delegated from the component owners to other interested developers. At other times, if the engineer is overburdened or unable to make the change, a "help wanted" marker is placed upon the bug report to make it clear that volunteers are needed.

The component owner is responsible for the module, and is considered an authority with regard to selecting and implementing changes. If he considers a contributed change relevant and well-implemented, he will allow it; otherwise, he can provide advice on how it should be improved, or ultimately ignore or veto the change. Bugzilla comments are usually the means of communication used to this end. This mechanism occasionally causes dispute, as virtually anyone has freedom to contest a decision made.

The areas of most dispute are, not surprisingly, issues which do not relate to the code architecture (the "back end") itself, but the browser user interface and functionality. The need for a process which specifically covers interface changes has shown itself to be urgently needed[11, 9] as the following comment from a developer in Bugzilla bug 75338, "Clean up context menus for Navigator/Messenger content area", indicates:

> Additional Comment #79 From Ian 'Hixie' Hickson 2002-01-18 08:26
>
> mpt: I agree with djk. Moving toolbars needs more discoverability. Either a menu item (like in Gnome), or grippies, or a tooltip... just _something_ to inform power users that moving toolbars is possible.
>
> marlon: Rumours are that you have also written a spec. Could you please publish it so that it can be compared to mpt's?
>
> marlon and mpt: It is a pity you did not work on these specs together (either in pixeljockeys, n.p.m.ui, or, at a pinch, by e-mail). Mozilla's QA people work as a team, Mozilla's engineers work as a team, Mozilla's evangelists work as a team. Yet repeatedly we end up with two conflicting views for UI specs with the parties not cooperating.
>
> staff@mozilla.org: Do we have a process in place yet for resolving the conflicts that are bound to happen as soon as someone tries to implement one of these specs? We've been waiting for YEARS now for some sort of UI process to be determined. It would be very unfortunate if this continued lack of process resulted in the different parties doing their own UI instead of contributing and working together on one UI.

3.2 "Bug-driven" development

Every code change in the Mozilla codebase is made as part of a "fix" for a uniquely numbered "bug"[5]. Though it commonly has a pejorative connotation, in the Mozilla Project the term **bug** is used to refer to any filed request for modification in the software, be it an actual defect, an enhancement, or a change

[5]This is actually an exaggeration; minor compilation fixes, and changes which are not part of the default browser build, can be checked in with no associated bug number.

in functionality. All change requests and their associated implementations have a unique number which identifies them. As of February 2002, there have been over 123,000 bugs reported in the Bugzilla defect-tracking tool, with over 80,000 bug reports for the Browser product alone.

Each bug has a number of changeable properties attached to it, including the following:

1. **Owner**: The person currently in charge of the bug.
2. **Summary**: A one-line description of the bug.
3. **Attachments**: Various files attached to the bug. Attachments can be test cases, screen captures and modifications to the codebase (commonly called "patches").
4. **Comments**: A number of comments describing and discussing the problem, possible fixes, and the code changes proposed.
5. **Status**: Describes the current state of the bug; one of UNCONFIRMED, NEW, ASSIGNED, RESOLVED, REOPENED, VERIFIED, CLOSED.
6. **Severity** and **Priority**: fields that describe the impact of the bug, and the order in which a bug should be (or is being) fixed.

Each bug is created with a state of UNCONFIRMED or NEW (depending on the experience the reporter is credited with). The task of actually confirming bugs by reproducing them rests on the volunteers who perform bug triage, which is one of the quality assurance activities in Mozilla.

Component owners are notified of bugs filed in their responsibility, and schedule them according to their priority and severity. This is hardly a strict process and, as discussed in the previous section, bugs are often reassigned from component owners to others when volunteer help is offered, the owner is overburdened, or others who know the code better are available. For people working on Mozilla under hire for a company, there are usually managers who schedule fixes among their developers according to their own needs.

Bug numbers are used, in this manner, as a code between developers, and they are exchanged freely and continuously through email, chat and Web sites. An excerpt from an online chat among Bradley Baetz and Joaquin Blas, two Mozilla developers, follows:

> <bbaetz>: kin fixed bugs 83650 and 97207!
> <kin> bbaetz: I knew it would excite some people :-)
> <bbaetz> kin: it was really annoying in bugzilla
> <kin> bbaetz: yeah I've been annoyed with it for quite some time. The problem was trying to get a reliable test case.
> <bbaetz> kin: I didn't file it - I thought it just came under the usual plain text annoying stuff

There are many high-profile bugs, which are voted on by the community, and these bugs are a focal point in end-user attention and expectation. The bug number remains the most valuable token system created by the project; the tool behind it, Bugzilla, is discussed in section 4.2.

3.3 Requirements

Often a controversial aspect of OSS projects[23, 26], the requirements process in Mozilla is also somewhat peculiar. High-level requirements are laid down by mozilla.org management, but since these are

few and very abstract, such as "complete Web Standards compliance", most of the decisions on functionality inclusion and change are discussed piecemeal by the community and module owners through bug and newsgroup comments. Though some areas, such as the user interface[6] and the module APIs, are reasonably detailed and documented[40, 38], many areas evolve on a case-to-case basis. Mike Shaver, a mozilla.org staff member since the project's creation, states this as follows:

> Were the original high-level requirements for Mozilla a combination of Netscape Navigator 5 features, "perceived needs" and standards compliance?
>
> Shaver: Largely, yes. There hasn't been a great deal of detailed specification for Mozilla, and I think that's one of the areas that we're going to need to nail down for 1.0, and the brave Peter Bojanic has signed up to help us get through that process. It's going to be hard, and a lot of people (including myself!) are going to be sad to see desired features or fixes miss the cut, but without a hard line we'll never get on to the next part of this great experiment.

A requirements change starts as most other proposed code changes. Discussion on the relevance of the change is started through one of the following routes:

- A message thread is started on a public newsgroup, regarding a change in functionality. Other people will usually comment on relevance and discuss advantages and disadvantages of the approach. The essence of the debate is often quite technical and will usually culminate in a bug being filed on the change, or the idea being abandoned.
- When a person has a specific idea for a change (such as bug 117162, "Mozilla could show that current URL is present in the bookmarks"[7] or bug 64066, "Could image blocker also block IFRAMEs?"[10]) a bug will often be directly filed with no newsgroup discussion. Discussion often occurs in the bug itself, and the vote system, community members, module owner, and ultimately mozilla.org staff will help determine if it is a relevant change or not.

Once a change has been decided upon, there is no guarantee it will be implemented soon (or ever). It is up to the community and mozilla.org management to see that developers follow up on the task and propose source code changes that will implement the functionality requested. Even when a change is actually implemented, there is a rare chance it will not be accepted into the codebase; Bugzilla lists a (small) number of bugs marked WONTFIX with patches. One example is bug 63458: "<blink> tag supported in standards mode": while the bug reporter did provide a patch to fix the issue – removing blink tag support when operating the browser in standards-compliance mode – it was decided that blink support was not harmful and should be allowed.

It is hard to say that the requirements process is generally inadequate: the community has active participation in the adoption of proposed features, and anyone is free to implement a desired change and submit it for approval. The fact that module owners and mozilla.org staff are the final authorities for determining the relevance of a change does present a barrier to adoption of controversial features, but this is hardly unprecedented, and seems a consequence of the level of control that the Mozilla process requires. Boris Zbarsky, a community member and developer with active participation in the project, describes the general requirements barrier as "You can get a feature in if you're willing to write code for it", and reckoned the module owners were "fair" when judging a proposed change.

[6] Surprisingly, heated discussion regularly occurs regarding one of the parts of Mozilla with detailed specifications: the user interface.

3.4 Design

The actual process of designing the Mozilla software architecture is difficult to abstract because of two important issues: first, the design inherits in part from original Netscape experience, so it was not completely invented in public view; second, because design discussions are inherently difficult to capture[20] and usually have sparse record.

According to Mike Shaver and Dan Mosedale, an engineer for Netscape, the original Mozilla design was a direct evolution of the Netscape Navigator 5 architecture. A part of it is still intact from that period: the Javascript engine and NSPR, an abstraction layer for the C library and threading, for example, survived the change to the new layout engine largely unchanged. The new layout engine was developed by a group of Netscape engineers inspired by the work done for "Xena", the original, all-Java, Netscape Navigator 6 plan. Their intent was to design a framework which would allow Mozilla to be written using similar technologies to those used in Web development — mark-up, style sheets, and scripting — and the fact that the browser today can be modified and enhanced without a compiler owes largely to this vision[7].

The design and specifications of both the original architecture[36] and the new layout architecture[46, 35] are documented, and there are a number of efforts to extend the design and code documentation through automated processing and analysis[13, 5].

As reflected by the largely ad-hoc requirements process, most design issues are tackled as they come: bugs are filed for changing the design or API of a certain component or class, and the ordinary discussion ensues over bug comments. The constant potential for changes in the design requires the module owner and super reviewers to have good knowledge of the current design. Both are ultimately responsible for judging and allowing the design to evolve in a certain direction. It is clear from our experience that the senior developers involved have a very good notion of the architecture and its evolution.

One issue this constant change produces is the burdensome task of keeping documentation up to date, and the documents on mozilla.org have aged considerably over these years. Shaver puts it this way:

> With few exceptions, the Mozilla design documentation is painfully out of date. I think there are worse things in the world than having people read source to learn – though I'm sure some will deride me as a cowboy-coder for not having proper respect for formal documentation – but the Mozilla code isn't always self-documenting either. Code that's gone through the super review process tends to be better, but [parts of the code were developed before super review was in place].

Because of the "continuous design adjustment", there is a constant need to perform refactoring of the legacy code. Refactoring allows developers to simplify APIs, remove unused code, and generally improve the modularity and readability of the code. This is hardly uncommon: OSS projects have been known to refactor continuously to avoid architecture breakdown. An example of this is bug 70929: "Refactor nsIGlobalHistory"[6], which involves creating a public interface to manipulate browser history[8], promoting better reuse of that part of the code (in this case, specifically for people who are embedding the Gecko layout engine in other applications).

Though the design process does have shortcomings, the availability of source code along with the review techniques discussed in the next section go a long way to allowing browser development to remain healthy.

[7] It is possible to develop and make patches which alter Mozilla without having a compiler tool set if the changes needed affect only the interpreted front-end

[8] Browser "history" is a conceptually a list of visited URLs through which Back and Forward buttons navigate

3.5 Distributed development and formal reviews

One of the premises the Mozilla Project was based upon was that face-to-face communication should not be required for development, which is strictly the rule for most, if not all, OSS projects[50, 27]. Thus all code would have to be designed, implemented, tested and integrated without relying on personal contact to solve problems. This poses many difficult situations and requires planning and support tools[24], but as the project shows, this goal is actually feasible.

All developers work using revision control (see section 4.1, CVS) on a common, centralized, codebase, which allows changes to be developed concurrently and independently "checked in". There is a single image of the code, and at any time any developer can easily retrieve the "tip", which is the latest version of the Mozilla source. This ensures that all developers see each other's changes, and that regression tests can be done on the very latest integrated code. The Bonsai and Tinderbox tools provide a way to query in real time the status of the repository, and the most recent changes. They are discussed in section 4.3.

The fact that developers rarely meet personally has some consequences which are often overlooked. First, participants are forced to communicate clearly and through written English, using email, online chat, newsgroups and bug comments. Second, lack of documentation is a significant barrier to entry to any novice participant – personal explanations of general overviews are difficult to obtain – though the fact that source code is available offsets this somewhat. Third, real time communication provides an important mechanism to educate developers and clarify the code. Because many of the lead developers are often online and available, questions and informal design reviews can be quickly performed (at the cost of some dispersion and interruption). Fourth, the lack of tools to aid communication and visibility into the process can seriously hamper the project's progress, so their availability, quality and usability are of dire importance.

Perhaps one of the most striking features of the process is the strict code review and approval system that code changes go through before integration. While other projects do include review as one of their base activities (Eric Raymond's "eyeballs" hypothesis[30]) the Mozilla Project is one of the first to systematically implement tool-supported formal review. Code reviews were instituted early in Mozilla as a way to avoid having incorrect code integrated into the repository (and eventually break the compilation), which was a major cause of delays and inability to work on behalf of the developer team. Super reviews, which involve review from a senior engineer very familiar with the code[18], were added later to provide support for more critical design evaluations, and developers acknowledge that code quality went up significantly once it started.

The review process works as follows: a developer working on a change for a bug produces a patch, which is a generated text file which describes the line-by-line differences made between the developer's local version and the latest version in the code repository. This patch is then attached to a bug in the Bugzilla system, and the developer requests review. A reviewer, which can be the module owner or anyone else familiar with the code, will then read the code critically and either grant review or ask for changes.

The actual review consists of bug comments describing changes that should be made to improve the code quality, questions about an unclear section, or recommendations on various other aspects of the patch (performance impact, dependencies, related problems). If no issues remain, the reviewer will mark his seal of approval by adding "r=(reviewer name)" to a bug comment. For some modules, super review is also needed: a senior developer, very intimate with the code, must mark approval for integration to be allowed (the mark used is sr=(name)). An excerpt from a review discussion on bug 119768: "Remove button in Smart Browsing should be context disabled"[8] is included.

In this session, a contributor includes a patch (indicated by the word "attachment") , and requests review. The super reviewer requests a change which the contributor accepts and integrates into a second patch:

> Additional Comment #2 From Samir Gehani 2002-01-16 15:52
>
> Created an attachment (id=65331)
> Fix to disable remove button when no domains present in disabled list.
>
> Additional Comment #4 From Samir Gehani 2002-02-07 12:15
>
> morse, please r.
> hewitt, please sr.
>
> Additional Comment #5 From Stephen P. Morse 2002-02-07 12:38
>
> (From update of attachment 65331)
> r=morse
>
> Additional Comment #6 From Joe Hewitt 2002-02-07 14:36
>
> Can you change this:
> + removeButton.disabled = "true";
> + else
> + removeButton.removeAttribute("disabled");
>
> To this:
> + removeButton.disabled = true;
> + else
> + removeButton.disabled = false;
>
> Additional Comment #7 From Samir Gehani 2002-02-07 14:44
>
> Created an attachment (id=68409)
> Patch rev 2 with reviewer changes.

Raymond pointed out that code review in OSS was an important tool in preventing bugs and improving code quality. Mozilla has taken this idea and implemented a formalized system which solves problems without becoming an overly burdensome bureaucracy. It also suggests that in Mozilla, above all, people are ultimately concerned with the quality of code being submitted.

3.6 Quality assurance and bug triage

Though many OSS projects do in fact include in their process Quality Assurance (QA) procedures[51], the Mozilla Project was also a pioneer in the use of the term itself to describe an explicit process. QA in Mozilla is performed by different classes of people, ranging from engineers hired by Netscape and other commercial contributors, to informal, ad-hoc volunteers willing to test and triage problems.

The QA team in mozilla has the mission of "finding and constructively reporting relevant bugs in mozilla.org OSS projects."[39]. From the mission statement, it becomes clear that QA work is intimately involved with testing the code and using the Mozilla development tools. Reality confirms this,

and many of the activities QA performs are reflected in the way these tools operate. The main activities QA is responsible for are testing and bug triage. There are many testing activities in Mozilla: ad-hoc volunteer testing, smoketests and even contributed functional tests, which are run by Netscape client QA.

Smoketests are different enough in Mozilla to warrant a separate explanation. Asa Dotzler, a mozilla.org staff member in charge of QA, explains that Mozilla smoketest plans are more thorough than what is generally considered a "smoketest", but not as complete as full functional tests.

Smoketesting is intimately linked to the "nightly build" process. Every day, the latest version of the Mozilla browser is compiled on the various supported platforms, and the binaries built are placed on a public FTP server. A group of testers then downloads these binaries and proceeds to execute a documented series of test cases which are required to work in order to declare that browser version as stable. If the tests fail, the people responsible for the offending changes are contacted and required to either fix or roll back their changes. This allows development to continue on a code base that is guaranteed to not include major regressions. Smoketesting is performed every day for the three most important Mozilla platforms: Windows, Mac OS and Linux, and is considered to be a fundamental tool in allowing concurrent development to succeed and not cause the code to "fall apart" due to regressions.

Bug triage involves working on the thousands of bugs filed in Bugzilla, reproducing and assigning problems, invalidating bugs when the described "bug" is actually intended behavior, or when there is a lack of reproducible steps, and working to better share the burden of fixes among the developers available. QA volunteers, along with the module owners, also target bugs to different versions, helping define what will be implemented for the next stable release.

By inviting volunteers to perform QA tasks, the Mozilla Project has effectively harnessed the attention of a very large and dedicated population of testers, who report bugs and follow up on reproducibility and resolution issues. Users are encouraged to file bugs because of the open nature of Bugzilla, and because they feel developers are actually paying attention to them.

Having a public forum for reporting problems and actually getting developer feedback on them is one of the most important advantages we have identified in the whole QA process. It not only prevents problems from being ignored or forgotten, but also makes viable a historical analysis of the data being stored. The fact the user base is a combined beta test community and QA assistance group is a fundamental feature in the Mozilla quality process.

4 The Mozilla development tools

This section describes a number of tools which were developed or customized by the Mozilla organization to support their software development process. One of the most important characteristics noted in the following sections is the integration between these tools, and how it allows management insight into the actual development of the browser.

Most of the Mozilla software tools are accessed through a Web browser. The easy access provided by the Web is suited to the needs of the Mozilla community, and underscores the importance of supporting a distributed software development process. All tools are available under an Open Source license[33], so they can be freely used in any (private or OSS) project.

4.1 CVS

CVS[4] is a freely available version control system. It is by far the most commonly used version control system in OSS projects, and it models very closely the code integration practices in them[48]. CVS

provides most features associated with version control. An overview of interesting points in the tool follows.

- **Central Repository**: CVS operates in a client-server fashion: a central server stores the code and version information, and clients request the code and receive copies of it.
- **Versions**: the basic functionality of a version control system, CVS allows previous versions of the files to be retrieved.
- **Branches**: CVS supports multiple "scenarios" of development, which can carry on independently. This feature is important when integrating new features with high impact.
- **Concurrent Development**: CVS does not place locks upon code "checked out". Instead, each developer checks out a local copy of the code, changes it, and when ready, submits it back to the repository. Conflicts in changes are handled client-side, and not in the repository.
- **Network support**: CVS works well over wide area networks, which is a pre-requisite for OSS development.
- **User interface**: CVS is implemented as a command-line tool that performs the many tasks involved with version control: checking out code, updating a local copy, printing differences between versions, and committing changes. The easy integration with Unix text processing tools is a very positive aspect.

Though shortcomings of CVS have been discussed and propositions for alternative solutions have been made[25], it remains a very stable and reliable product. The community has invested a lot of time on integrating CVS with other tools, and our opinion is that CVS will remain in wide use in OSS for a very long time.

4.2 Bugzilla

Bugzilla[14] is the flagship tool in the Mozilla suite of software tools, and is also the most developed of them. It is a Web-based issue-tracking system, written primarily in Perl. As explained before, Bugzilla supports a symbolic communication between users based on its key concept, the bug number.

Bugzilla sports a great deal of functionality, and its features match the Mozilla process quite closely. Bugzilla provides the support for the many different activities that were described as being part of the Mozilla process, and acts as a central hub for communication and code review among community members.

- Developers use it daily to register patches and to request and perform review.
- QA uses it to gauge progress and to report and triage bugs.
- Managers use it to allocate developers and track progress on major issues.

Bugzilla is so essential to the Mozilla community that when outages occur, work for many people is completely blocked. Its notable characteristics include:

- **User Accounts**: each Bugzilla user must create an account, identified by his email address. The account creation and validation process is very simple, which reduces barrier to entry and encourages participation.
- **Bug attributes**: bugs have properties that match very closely Mozilla process requirements, and allow for fine control over responsibility, scheduling, dependencies and status. Figure 3 shows the various attributes of a bug.

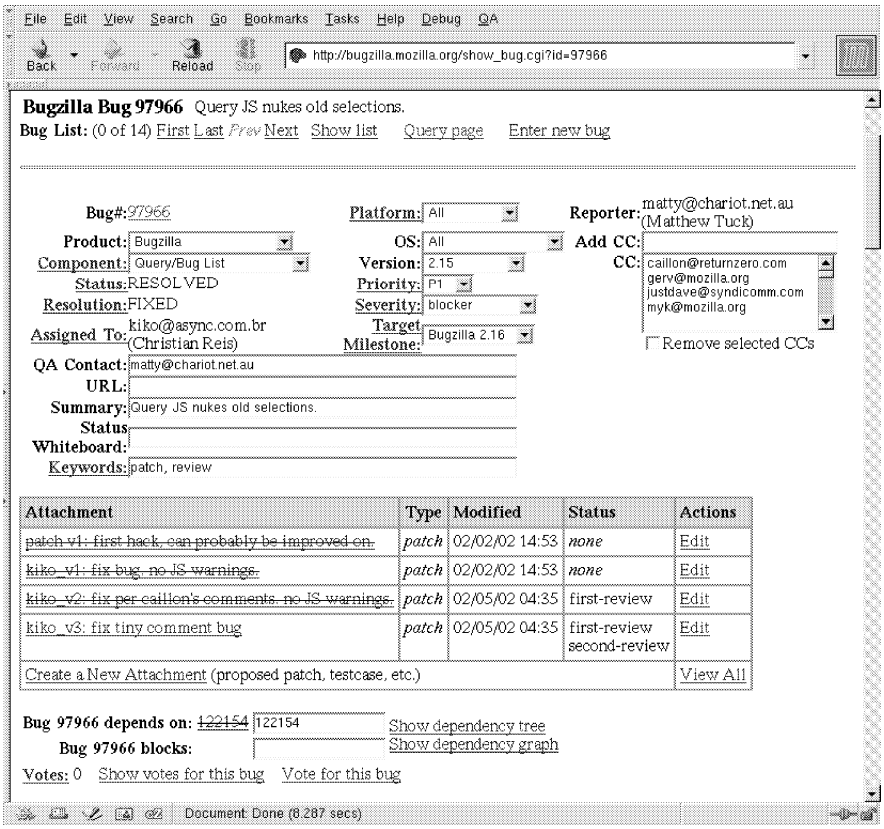


Figure 3: A screen capture of a Bugzilla bug form.

- **Comment log**: because each bug keeps a sequential list of user comments, Bugzilla works very well as a focused discussion forum. Features like automatic hyperlinking and commenting are very convenient.
- **Attachment tracker**: bugs can have "attachments", which are files uploaded by the user and linked to a specific bug. Most of the attachments are patches for bugs, but they can also be test cases, screen captures and specifications. The attachment tracker also provides support for reviewing code, and has a special interface for it.
- **Query interface**: the mozilla.org installation of Bugzilla registers hundreds of thousands of bugs. To support queries on this database, there is a search function that allows one to specify what attributes define the bug being searched for.
- **Email integration**: Bugzilla changes are mailed to all parties that are registered with the bug, allowing people to be notified of requests and the bug's progress.
- **Simple conceptual model**: Bugzilla is basically a bug register, and though it has a lot of associated functionality, the concepts are simple to grasp: products, components (which are subdivisions of a product), attachments and bugs.

Bugzilla has proven to be scalable and effective in easing coordination of development in very large products. It is used in a large number of organizations and companies outside of the Mozilla Project, including Netscape, NASA, Conectiva, Redhat and Gnome[2, 32, 12]. A traffic analysis of Mozilla's Bugzilla site for the week of January 31st to February 6th, 2002, which was the week of Mozilla release 0.9.8, shows a daily average of almost 5,500 unique visitors, which underscores Bugzilla's importance

to the community.

Development of Bugzilla is occurring rapidly, and many new features are planned or being implemented. The next release, version 2.16, concentrates on allowing Web pages to be easily modified to suit local needs and layout, and complying with the HTML 4.01 Transitional standard. Other development areas include further modularization, greater customizability of features, tighter integration with other tools, and user interface improvements.

4.3 Bonsai and Tinderbox

Bonsai and Tinderbox are tools which provide real time access to code changes and their impact on Mozilla the compile and test cluster. Bonsai[34] is a query interface to the CVS repository. Bonsai sports a number of features:

- allows one to query CVS for the latest check-ins done to the repository
- shows check-in comments with hyperlinks to the Bugzilla bug that was fixed
- provides an interface to view differences between versions of files in the CVS repository, allowing a developer to quickly track down and visualize a set of changes that have landed
- allows visual identification of which developers are responsible for which sections of a single code file
- provides a recently added graphing mechanism based on the cvsgraph tool, which depicts the different versions of files on the project repository's branches

Bonsai is an excellent tool for tracking the actual progress in terms of code check-ins. It is also important for statistical analysis of the repository activity: the numbers for developer participation listed in table 1 were obtained from the Bonsai check-in database.

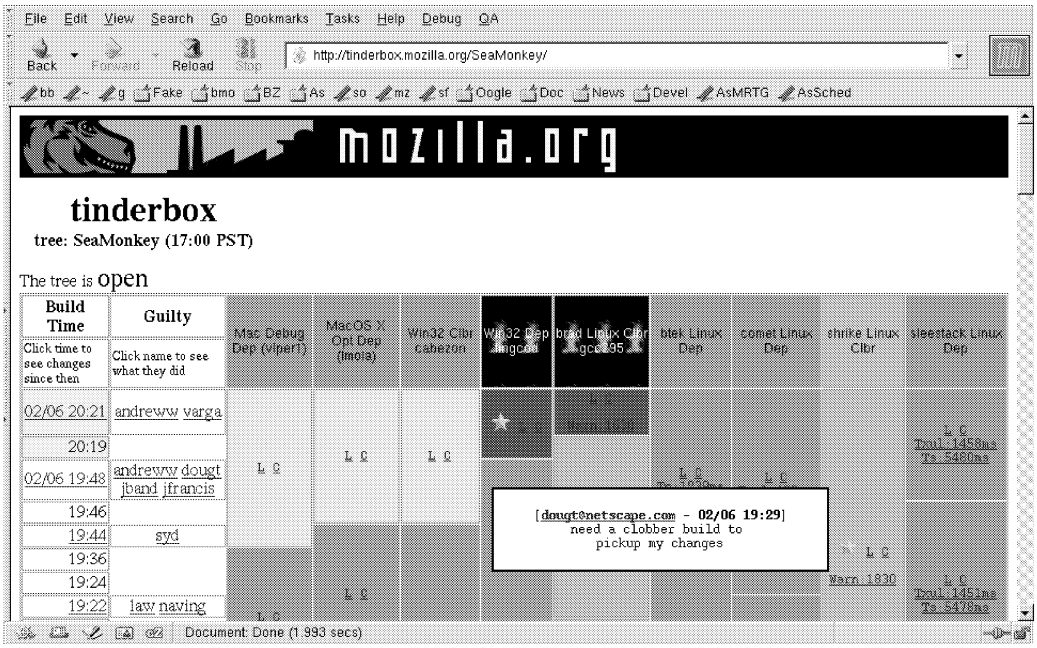


Figure 4: A screen capture of Tinderbox output.

Tinderbox is an automated system that tracks compilations and tests. It is a client-server tool: client machines of various architectures and operating systems form a cluster. These machines' task is to compile, test and report results back to the Tinderbox server. The main user-visible feature (figure 4) is a page displaying the compilation results associated with the individual machines in the cluster, and the code changes that were integrated to the repository at the same time. Each machine of the compile cluster is represented by a column, which shows the various builds that happened during a period of time. The most recent builds appear in the top-most column. The color of each section of the column indicates the compilation results:

- red indicates compilation failed
- orange indicates compilation succeeded, but the test-suite failed
- green indicates compilation and tests ran successfully
- yellow indicates compilation is still in progress

As time goes by, new builds are generated and enter the display. The left-most column indicates the names of people that checked in changes during that time period, and provides links to associated Bonsai queries. This way, it is trivial to find out what code change was committed by which developer, and what bug it fixed.

Tinderbox is a very interesting tool. It allows regressions to be quickly tracked down and fixed, since code check-ins that cause problems will cause the compilation machines to fail. It also allows code that is not cross-platform compatible to be identified and fixed, removing the need for each developer to guarantee it works on all architectures and operating systems. The test suites developed for Tinderbox, apart from performing some functional testing, also help track down regressions in performance, as exemplified by figure 5, which is generated as part of the test suite.

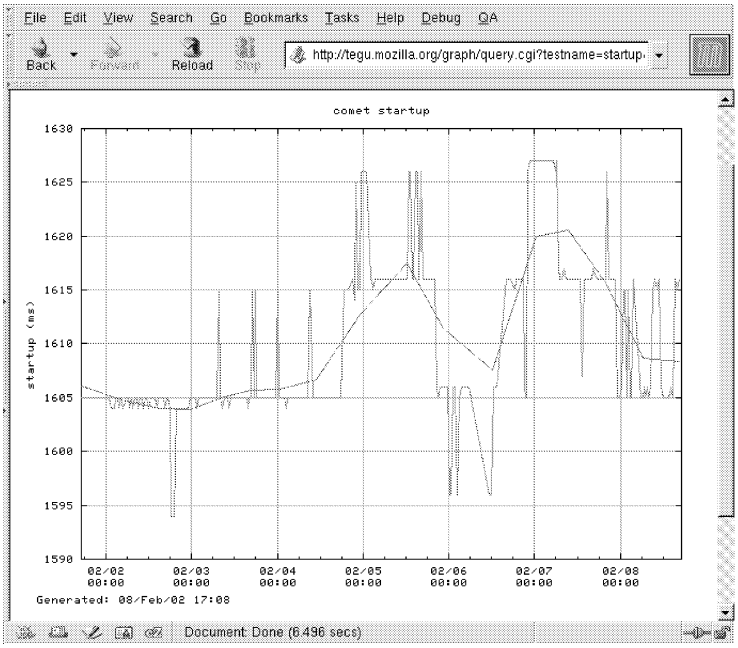


Figure 5: A graph of the time it takes to start the Mozilla browser, varying over days.

4.4 LXR

LXR (figure 6) is a hypertext tool that indexes source code and generates HTML pages. It was originally developed as a tool for studying the Linux kernel, but was adapted to Mozilla by the community. It displays code files as pages, with links to each line and for each identifier: functions, classes and variables are all hyperlinked. It is possible to access, for instance, the declaration and implementation of a function that is being called in a certain file.

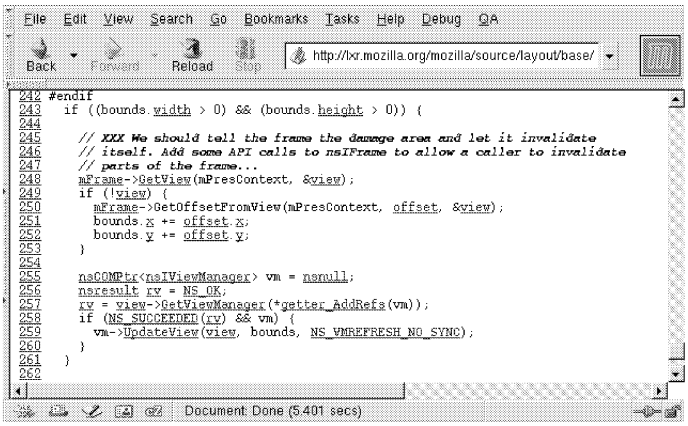


Figure 6: A screen capture of LXR displaying a page of the Mozilla code.

LXR also performs free text searching for file names and contents, and querying for named identifiers. The community uses it basically as a reference and learning tool: LXR links are often posted on bug comments and in IRC conversations, and provide an easy way to point developers to a specific part of the very large code repository.

4.5 Communication tools

The Mozilla community uses Bugzilla to a great extent as a communication tool, but for certain tasks mailing lists, network newsgroups and IRC are preferred. Mailing lists and newsgroups are generally used for discussion that is less focused than bug comments: new features, design review, measurements and statistics, and less technical inquiries.

IRC is a real time communication system which organizes participants in "channels". It presents an interesting scenario for a distributed development environment: although it is real time, it allows for selective communication and thus can be used as a reasonable substitute for a telephone when technical matters are to be discussed. Mozilla's IRC channels are divided by intended audience. For example, there is a channel for technical discussions relating to the browser (#mozilla) but also a channel for volunteers performing smoke tests (#smoketest). A number of "bots", which are programs that present themselves as IRC users, but which respond to specific commands, are hosted and provide convenient reference material when needed. As an example, in the following session, a bot named ssdbot looks for messages containing the word bug followed by a number, and answers with bug information and a Bugzilla link:

> <timeless> does anyone know what bug 124441 could be duped against? (it's a browser bug iirc)
> <ssdbot> timeless: Bug http://bugzilla.mozilla.org/show_bug.cgi?id=124441 nor, –, —, asa@mozilla.org, UNCO, CPU/memory consumption when loading large pages
> <Jake> It sounds familar... IIRC it had buglist in the summary....
> <Jake> timeless: It's bug 77460
> <ssdbot> Jake: Bug http://bugzilla.mozilla.org/show_bug.cgi?id=77460 cri, P2,

mozilla0.9.9, rjc@netscape.com, RESO FIXED, buglists hang mozilla for long periods of time using all available CPU

5 Conclusions

This paper has covered a very broad subject, and has only begun to describe the complexity and richness which the Mozilla Project displays. It is intended as an invitation to other interested researchers in the software engineering community to study the project. Research in the Mozilla Project is both suited to newcomers to the field of software engineering, who often have no way to actually watch and understand how a software process occurs "in the wild", and experienced software engineers, who may have contributions to make, and opportunities to analyze how an OSS project really evolves.

It is important not to overlook the opportunity that Mozilla presents to analyze an ongoing software process outside of a corporate environment. Software engineering field research is often limited by the willingness of a software development group, and the availability of a suitable project to study; the Mozilla Project offers a chance to simultaneously analyze and participate in the development of an important software product.

5.1 Future work

We have listed here a number of areas that might prove interesting research goals.

1. Measurements of developer productivity through analysis of the CVS and Bugzilla data available, with a view to what affects this productivity, and how it compares with non-OSS projects.
2. In-depth studies of how real time communication can be used to integrate geographically distributed development teams.
3. Subjective analysis of the social interactions between community members with relation to changes proposed, with a view to suggesting a good way to arbitrating disputes over product features.
4. Surveys of user and developer participation in the project to gather general satisfaction and perceived problems with relation to the Mozilla software process.
5. Statistical analysis of the quality data collected by the Mozilla software tools, specially if it were made possible to analyze the data on product MTBF that is collected by mozilla.org This, in conjunction with Bugzilla and CVS data, could provide an insight into what changes have larger impact on product stability.

We will continue our study of the project and hope to publish more useful research material.

5.2 Credits

Much credit should go to the Mozilla community for providing helpful information, reviews and suggestions. All of the images, quotes and IRC logs used in this paper are authentic and represent actual interaction between developers, and they must be thanked for allowing us to use their content in this paper.

Special thanks to Mike Shaver, Dan Mosedale, Christopher Aillon, Matthew Thomas, Ian Hickson, Asa Dotzler and Dawn Endico for providing statistics, review and insights into the Mozilla process.

References

[1] Estimating Linux's Size. 2001. URL: http://www.webreview.com/browsers/1999/08_20_99_2.shtml. Visited February 2002.

[2] "Alex" Kin Lee. Who else is working on Mozilla-based projects? 2000. URL: http://www.gerbilbox.com/newzilla/mozilla/general05.php. Visited February 2002.

[3] Mitchell Baker. State of the Mozilla Project: Out and About. 2001. URL: http://www.mozilla.org/docs/ora-oss2001/state-of-mozilla/. Visited February 2002.

[4] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[5] Brian W. Bramlett. Mozilla/SeaMonkey Code Documentation and Cross-Reference . 2002. URL: http://unstable.elemental.com/mozilla/. Visited February 2002.

[6] Mozilla.org Bugzilla. Bug 70929: Refactor nsIGlobalHistory. 2001. URL: http://bugzilla.mozilla.org/show_bug.cgi?id=70929. Visited February 2002.

[7] Mozilla.org Bugzilla. Bug 117162: Mozilla could show that current URL is present in the bookmarks. 2002. URL: http://bugzilla.mozilla.org/show_bug.cgi?id=117162. Visited February 2002.

[8] Mozilla.org Bugzilla. Bug 119768: Remove button in Smart Browsing should be context disabled. 2002. URL: http://bugzilla.mozilla.org/show_bug.cgi?id=119768. Visited February 2002.

[9] Mozilla.org Bugzilla. Bug 49543: Separate Toolbar from Address Bar [urlbar]. 2002. URL: http://bugzilla.mozilla.org/show_bug.cgi?id=49543. Visited February 2002.

[10] Mozilla.org Bugzilla. Bug 64066: Could image blocker also block IFRAMEs? 2002. URL: http://bugzilla.mozilla.org/show_bug.cgi?id=64066. Visited February 2002.

[11] Mozilla.org Bugzilla. Bug 75538: Clean up context menus for Navigator/Messenger content area. 2002. URL: http://mozilla.org/roadmap/roadmap-26-Oct-1998.html. Visited February 2002.

[12] Conectiva S.A. Conectiva Bugzilla Database. 2002. URL: http://bugzilla.conectiva.com/. Visited February 2002.

[13] Datrix (Bell Labs Canada). Source Code Analysis of Netscape's Communicator 5.0 developer release. 1998. URL: http://www.iro.umontreal.ca/labs/gelo/datrix/Mozilla-analysis/contents.%html. Visited February 2002.

[14] David Miller and Jacob Steenhagen. Bugzilla Bug Tracking System. 2002. URL: http://www.bugzilla.org/. Visited February 2002.

[15] Asa Dotzler. Mozilla Community Quality Assurance and Testing. 2002. URL: http://mozilla.org/events/dev-day2001/community-testing/. Visited February 2002.

[16] Brendan Eich. Development Roadmap [old]. 1998. URL: http://mozilla.org/roadmap/roadmap-26-Oct-1998.html. Visited February 2002.

[17] Brendan Eich. Mozilla Development Roadmap. 2001. URL: http://mozilla.org/roadmap. Visited February 2002.

[18] Brendan Eich and Mitchell Baker. Mozilla "super-review". 2000. URL: http://www.mozilla.org/hacking/reviewers.html. Visited February 2002.

[19] Freshmeat.net. Statistics and Top 20. 2002. URL: http://freshmeat.net/stats/. Visited February 2002.

[20] Thomas R. Gruber and Daniel M. Russell. *Design Knowledge and Design Rationale: A Framework for Representation, Capture, and Use*. Knowledge Systems Laboratory, Stanford University, 1990.

[21] Jim Hamerly, Tom Paquin, and Susan Walton. Freeing the Source: The Story of Mozilla. 1999. URL: http://www.oreilly.com/catalog/opensources/book/netrev.html. Visited February 2002.

[22] Frank Hecker. Mozilla at One. 1999. URL: http://mozilla.org/mozilla-at-one.html. Visited February 2002.

[23] Lisa GR Henderson. Requirements Elicitation in Open-Source Programs. 2000. URL: http://www.stsc.hill.af.mil/crosstalk/2000/jul/henderson.asp. Visited February 2002.

[24] James D. Herbsleb and Rebecca E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of ICSE*, pages 85–95, Los Angeles, May 1999. IEEECSP.

[25] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The Project Revision Control System. In *Proceedings of the International Symposium on System Configuration Management*, number 8, Brussels, 1998.

[26] Steven C. McConnell. Open source methodology: Ready for prime time? *IEEE Software*, 16(4):6–8, July/August 1999.

[27] Audris Mockus, Roy Fielding, and James Herbsleb. A case study of open source software development: The Apache Server. In *Proceedings of ICSE*, pages 263–272. IEEECSP, June 2000.

[28] Bruce Perens. The Open Source Definition. In *Open Sources*, pages 171–188. O'Reilly and Associates, Sebastopol, 1st edition, 1999.

[29] Eric S. Raymond. Homesteading the Noosphere. In *The Cathedral and The Bazaar*, pages 79–135. O'Reilly and Associates, Sebastopol, 1st edition, 1999.

[30] Eric S. Raymond. The Cathedral and The Bazaar. In *The Cathedral and The Bazaar*, pages 27–78. O'Reilly and Associates, Sebastopol, 1st edition, 1999.

[31] Joel Spolsky. Things You Should Never Do, Part I. 2002. URL: http://joelonsoftware.com/articles/fog0000000069.html. Visited February 2002.

[32] The GNOME Project. Gnome's Bugzilla Bug Database. 2002. URL: http://bugzilla.gnome.org/. Visited February 2002.

[33] The GNU Project. Various Licenses and Comments about Them. 2001. URL: http://www.gnu.org/philosophy/license-list.html. Visited April 2001.

[34] The Mozilla Organization. Bonsai. 2000. URL: http://www.mozilla.org/bonsai.html. Visited February 2002.

[35] The Mozilla Organization. The XPToolkit Architecture. 2000. URL: http://www.mozilla.org/xpfe/xptoolkit/index.html. Visited February 2002.

[36] The Mozilla Organization. Documentation Graveyard. 2001. URL: http://www.mozilla.org/classic/. Visited February 2002.

[37] The Mozilla Organization. Hacking Mozilla. 2001. URL: http://www.mozilla.org/hacking/. Visited February 2002.

[38] The Mozilla Organization. Mozilla Developer Documentation. 2001. URL: http://www.mozilla.org/docs/. Visited February 2002.

[39] The Mozilla Organization. Mozilla Quality Assurance. 2001. URL: http://www.mozilla.org/quality/. Visited February 2002.

[40] The Mozilla Organization. User Experience Specifications. 2001. URL: http://www.mozilla.org/projects/ui/communicator/. Visited February 2002.

[41] The Mozilla Organization. XPCOM. 2001. URL: http://www.mozilla.org/projects/xpcom/. Visited February 2002.

[42] The Mozilla Organization. mozilla.org. 2002. URL: http://www.mozilla.org/. Visited February 2002.

[43] The Mozilla Organization. mozilla.org at a glance. 2002. URL: http://www.mozilla.org/mozorg.html. Visited February 2002.

[44] The Mozilla Organization. mozilla.org Staff Members. 2002. URL: http://www.mozilla.org/about/stafflist.html. Visited February 2002.

[45] The Mozilla Organization. mozilla.org Statistics. 2002. URL: http://webtools.mozilla.org/miscstats/. Visited February 2002.

[46] The Mozilla Organization. NGLayout Project: Technical Documentation. 2002. URL: http://www.mozilla.org/newlayout/doc/. Visited February 2002.

[47] Matthew Thomas. Not that I know anything about Software Engineering. 2002. URL: http://mpt.phrasewise.com/2002/01/27. Visited February 2002.

[48] André van der Hoek. Configuration management and open source projects. In *Proceedings of ICSE Workshop: Software Engineering over the Internet*. IEEECSP, 2000.

[49] David A. Wheeler. Estimating Linux's Size. 2001. URL: http://www.dwheeler.com/sloc. Visited February 2002.

[50] Yutaka Yamauchi, Makoto Yokozawa, Takeshi Shinohara, and Toru Ishida. Collaboration with Lean Media: How Open Source Succeeds. In *Proceedings of CSCW*, pages 329–338. ACM Press, 2000.

[51] Luyin Zhao and Sebastian Elbaum. A survey on quality related activities in open source. *ACM SIGSOFT Software Engineering Notes*, 25(3):54–57, May 2000.

Architectural Requirements for an Open Source Component and Artefact Repository system within GENESIS

Cornelia Boldyreff, David Nutter* and Stephen Rank

February 11, 2002

David.Nutterdurham.ac.uk

Abstract

When software is being created by distributed teams of software engineers, it is necessary to manage the work-flow, processes, and artefacts which are involved in the engineering process. The GENESIS project aims to address some of the technical issues involved by providing a software system to support distributed development. One of the parts of the system will be known as OSCAR, a repository for managing distributed artefacts. Artefacts can be process models, software components, design documents, or any other kind of entity associated with the software engineering process. OSCAR will be designed as a light-weight distributed system, managing the storage and access to a distributed repository of artefacts.

This paper presents and discusses the requirements for OSCAR, and suggests a possible architecture for a software system which will meet those requirements. OSCAR will be a reliable and light-weight distributed system, managing both artefacts and meta-data corresponding to the artefacts. Users of OSCAR will be able to access the distributed repository through a local interface, using the searching and indexing capabilities of the system to locate and retrieve components. OSCAR must be able to store and retrieve both artefacts and meta-data efficiently. It must be possible for OSCAR to inter-operate with existing artefact management systems (such as CVS) and to collect metrics about the contents of and accesses to the repository.

The next stage in the GENESIS project is to complete the requirements for the whole of the system (in addition to the OSCAR sub-system) and then to design the software. The software will initially be developed in a traditional closed-source fashion until the first release is finished. After the first release, the GENESIS software will become open source, and will be developed accordingly.

1 Introduction

A software artefact is an item produced by any phase of a software development process which is not limited to actual software code but include requirements,

design, quality assurance and maintenance information. Consequently the management of software artefacts is different to that of software code and changes must be made to the *software repository* to reflect this.

In particular software artefacts inherit characteristics from:

- The software process that created them,
- Preceding versions of the artefact and obviously preceding versions of the software process,
- The dependencies each artefact has upon other artefacts and
- The *Actors* (whether human or machine) who are responsible for the artefact.

Traditional software repositories are generally passive data stores designed to support structured access to information by tools and external systems. OSCAR itself must support a more aware approach to artefact management through the notion of "active" software artefacts that are aware of their own evolution and present related information upon examination by system clients.

Since GENESIS is a process-aware software engineering environment, the artefact management system must support process control software. The GENESIS system requires that it is *non-intrusive* when deployed within an organisation. Consequently the support offered by GENESIS must not force participation from the users, instead the system must monitor user activities and alter its behaviour accordingly. To assure the non-intrusivity of the finished system and to satisfy the additional requirements imposed by the eventual release of the GENESIS system as Open Source Software (OSS) OSCAR must employ supported open standards. This perhaps seems a fait accompli; after all releasing GENESIS as Open Source will effectively produce a new open "standard". This approach ignores the advantages that can be gained by both adding "weight" to existing standards and re-using pre-written Open Source components. Open standards also offer related benefits [Raymond, 1999]:

- Existing development community,
- Ease of deployment,
- Ease of extension and
- Ease of accessibility.

This paper examines the requirements for the OSCAR subsystem within GENESIS. The paper is organised as follows:

Section 1.1 discusses work related to the GENESIS project and OSCAR in particular,

Section 2 describes the requirements of the domain that OSCAR operates in

Section 3 describes a sample use case for OSCAR,

Section 4 describes the high-level requirements of the OSCAR subsystem,

Section 5 describes a proposed architecture for OSCAR and suggests some design features of OSCAR,

Section 6 describes an ontology of artefacts for the OSCAR subsystem,

Section 7 outlines the development and exploitation strategy for OSCAR. Finally,

Section 8 summarises the previous section

1.1 Related Work

Partial solutions for software artefact management exist, where items produced in the software process are treated as more than mere version-controlled files. The products in this area are of three major types: commercial software, Open Source or Free software and research prototypes. In addition, these packages are primarily focused on other domains (such as collaborative working) than artefact management but possess additional artefact management capabilities.

Of the commercial packages, advanced Software Configuration Management (SCM) environments such as Starbase and Clearcase provide the closest functionality to true Software Artefact Management systems [Leblang, 1994]. Aside from an enriched file-system that provides a global name-space which artefacts inhabit, such systems also have process support in the style of traditional workflow systems.

Such systems are characterised by their heavyweight approach (process-centric) to software development and centralised architecture. For projects heavily distributed in both time and space and without access to high-availability services at all points, such an architecture is both difficult to implement, untrustworthy and potentially harmful to existing successful organisational processes [Sachs, 1995]. Careful identification of requirements and design [Selvin, 1999] will produce a system that facilitates existing organisational processes rather than prescribing new ones [Flores et al., 1988].

Open Source and research systems are generally more lightweight and specialised in their approach, focusing on one aspect of the domain such as version control, process control, collaborative working through hypermedia and so forth. Examples include the Oz and OzWeb (described in [Kaiser, 1998], [Gail E. Kaiser and Yang, 1997] and [Jiang et al., 1997]) environments. Haake [Haake, 1999] made a compelling case for this lightweight approach, arguing that flexibility of working practice if not process was more beneficial than enforcing a prescriptive focus through CSCW tools. Kukkonen has produced work in a similar vein. [Oinas-Kukkonen and Rossi, 1999]

2 Domain

The OSCAR tool will be situated in the domain of distributed software engineering. One of the key characteristics of this domain is that the users of the tool (programmers, configuration managers, requirements engineers, *etc.*) are distributed. That is, users are not necessarily all located at one physical site. Thus OSCAR must be able to manage the necessary distribution of artefacts

to users as required; it is not possible to assume that each user will have local access to the repository.

Entities in the domain are:

Actors In general, a user is a software engineer of some kind (requirements engineer, software designer, tester, *etc.*), a person with managerial responsibility for the software process or indeed a software agent acting on behalf of these other Actors.

Artefacts Artefacts are stored in the repository. An artifact can be anything which can be stored as a file on a computer system, such as process models, software design documents, source code, executables, testing suites, *etc.*. Additionally the artefact includes *meta-data* which characterises it for the purposes of search and retrieval.

Networks As the OSCAR tool must operate in support of distributed teams, it need to allow and support communication.

It is the task of the OSCAR tool to manage artefacts. Users will interact with the tool to:

- Store artefacts,
- Retrieve artefacts,
- Update artefacts and
- Search for artefacts which match the user's needs (for example, to find a software component which implements a particular task).

As an example, the meta-data for an artefact could be:

Name	Value
Artefact name	Text entry widget
Creation date	25th December 2001
Creator	A Smith
Creator	J Bloggs
Modification date	14th January 2002
Modification comments	Fixed possible buffer-overflow
Modification date	26th January 2002
Modification comments	Now matches look-and-feel model for project

3 Example Use-Case

In this section, a simple use case [Jacobson et al., 1999] for the OSCAR tool is outlined and examined. The use case involves a user of the system (a software developer) retrieving a component from the OSCAR repository. In this example, the developer (who is the actor) provides a template for the artefacts which he or she is looking for. This template is then given to the local client to the OSCAR system which retrieves matching artefacts from the (distributed) repository. The template can, for example, be a simple set of keywords, in which case the search is similar to that executed by web search engines.

3.1 The 'Retrieve Artefact from Template' Use Case

As an outline, the sequence of events for this use case are as follows:

1. The Developer identifies him/herself to the system.
2. The Developer constructs a Template for the kind of artefact that they are interested in.
3. The Developer supplies the Template to the local client of the OSCAR system.
4. The local client searches the repository for artefacts which match.
5. The local client presents the results, ranking them in order of relevance to the template, to the user.
6. The Developer selects a matching artefact from those presented to him or her.
7. The local client retrieves the artefact and delivers it to the user.

Given that OSCAR is a distributed repository system, step 4 will involve the following:

- The local client performs a search on the all the meta-data available about artefacts in the distributed repository. For example, if there is a single centralised store of all the meta-data, the local client will query this store. If the meta-data is distributed throughout the network, each OSCAR repository will be searched.
- The meta-data repository or repositories search for items of meta-data which match the template.
- The meta-data repository or repositories return the matching meta-data.

Step 7 can be broken down into the following steps:

- The Developer asks the local client to fetch a particular artefact.
- The local client determines which of the distributed OSCAR servers holds the artefact in question.
- The local client requests that the particular server returns the artefact.
- The server transmits the artefact to the local client.

There are two alternative models mentioned above (centralised or distributed meta-data services). Figure 1 shows the networking connections required for a local client with a central meta-data server. Using a central server introduces a single point of failure into the system, but it avoids the problem of distributing meta-data throughout the system and maintaining consistency. This is a trade-off which will have to be examined more carefully before a decision can be made. In figure 1, the arc labelled 'search' corresponds to the connection made by the client to the repository during the search on the meta-data, while the arc labelled 'retrieve' corresponds to the connection made by the local client to the artifact repository which contains the particular artefact which the developer has requested.

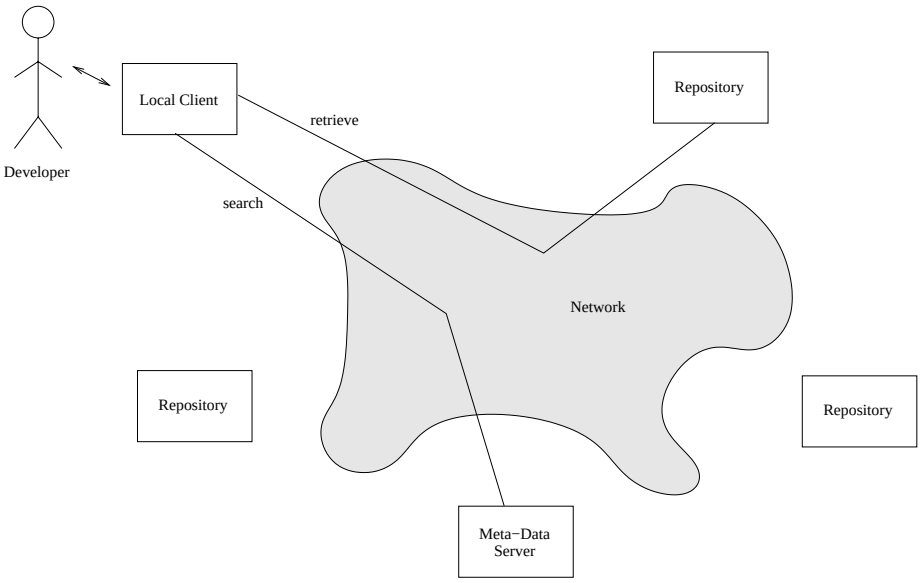


Figure 1: Searching for and Retrieving an Artefact

4 Requirements

4.1 Presentation Requirements

OSCAR must present the data it contains in a meaningful manner to clients which may be human, machine or a mixture of the two; for example a "guardian agent" approach where intelligent software mediates the human's interaction with the information. Consequently the following software requirements:

1. Intelligent data transformation. While OSCAR will have a common data format, clients and other external sources will have their own data that will need translating before it is meaningful, for both for input and output with OSCAR.

   Additionally the manner in which data is presented will change when the task the client is performing changes, and for human users with the experience level of the individual user.

2. Presentation of dependencies between artefacts. To facilitate tasks such as impact analysis the dependencies between artefacts of interest in OSCAR must be presented to clients as supplemental information during interaction. This will assist in alleviating the problems caused by changing superficially unrelated items.

3. Presentation of search results. Information generated by search jobs must be transformed into an appropriate form for the clients. While a printed report may be fine for a human they would be less useful to a tool looking for software components to compose.

4. Presence awareness. Like dependency presentation, the activities of other users should be indicated during interaction to facilitate collaboration.

Again, the means of indication must vary according to the characteristics of each client.

4.2 Indexing Requirements

OSCAR must index its contents to allow rich user queries. Queries are generally are intended to match appropriate artefacts to retrieve from OSCAR and also to score them at the client to present the best possible match.

5. Fuzzy/similarity based searching. Rather than just a simple boolean search like that found in many software products, the nature of OSCAR and the data contained within it requires a more sophisticated method of selecting search results. The type of search envisioned is conceptually similar to that outlined by Lai [Lai and Tait, 1998] that mimics the search techniques of humans looking for images.

6. Rich, extensible meta-data model to describe OSCAR's contents. Such a model is necessary to support the sophisticated searching and indexing OSCAR requires.

7. Reasonable response time. As the system is potentially highly distributed, searches must return a reasonable set of matches in a reasonable[1] time to ensure that a search does not take so long to complete that it impairs the performance of the rest of the system

8. Transparent distributed search. Whether an artefact (or its descriptive meta-data) is held locally or elsewhere in the system the search and subsequent retrieval of artefacts should be exactly the same.

4.3 Metrics Requirements

When users interact with OSCAR whether directly or from elsewhere in the GENESIS environment their actions naturally affect the environment. A record of these activities and their consequences provides an insight into the working practices of the virtual organisation built around OSCAR and GENESIS. Consequently a metrics system and extensible instrumentation package must be integrated with OSCAR to build this record. The requirements for this subsystem are as follows:

9. Transparent collection and calculation of metrics. To support the requirement for non-intrusive behaviour in GENESIS the metrics module should not require mandatory operation on the part of the user for correct operation.

10. Supplementing artefact meta-data with information derived from the metrics. Therefore the history and characteristics of the activities using an artefact should "surround" it when presented to the user and provide additional properties to search.

[1] "Reasonable" is defined by the client requesting the search.

11. Collation of certain metrics into additional documentation items. The information generated by some artefacts may not be useful when directly associated with the artefacts themselves. Instead a virtual document should be produced containing the information and associated with the artefact. This approach is intended to reduce information overload on human users.

12. Optionally extensible by users. If a user has a requirement for a metric not included in the standard instrumentation package, adding the metric should be simple.

4.4 Data Storage Requirements

The GENESIS system stores two kinds of data for each artefact, the *meta-data* and *artefact data*. Table 1 indicates the characteristics of each of these types:

	Meta-data	**Artefact Data**
Size	Small, relatively constant	Any size, variable
Structure	Highly structured	Partially structured
Operations	Mainly Query/Read	Read/Write balanced
Format	Plain text	Binary and plain text
Dependencies	None	Other artefact data
Links	Other meta-data	(optional) Data source

Table 1: Data Characteristics

The storage requirements for these two types of data differ significantly:

Meta-data
- Fast to search (sorted data)
- Lots of relational information
- Powerful query/edit system

Artefact Data
- Efficient storage (little wasted space or CPU time)
- Fast to access (no need to query a specialised program like a DBMS to retrieve data
- Error resistant (versioning,backoff etc)

This indicates the following software requirements:

13. A separate fast and structured store for meta-data, supporting a sophisticated query language.

14. A space efficient store for artefact data, containing a backup of any changes to meta-data.

15. A distribution model that allows optional distribution of both data stores in a *global name-space* while still retaining the error resistance and recovery properties of each.

4.5 Non-Functional and Inherited Requirements

The final set of requirements are those that are not directly related to OSCAR's functionality or are inherited from those of the wider GENESIS environment. Though they are not explicitly part of OSCAR they will still affect the design of the system:

16. The key requirement is that the GENESIS tool must be non-intrusive. GENESIS and consequently OSCAR must not gratuitously affect the existing organisational processes.

17. A high degree of dependability is necessary for OSCAR to be accepted and trusted with artefacts.

18. The requirement for distribution provides both dependability benefits (redundancy) and challenges (synchronisation and change control). The distribution models selected shall take both into account.

19. To facilitate the requirement for a non-invasive environment, GENESIS and OSCAR must be open and interoperable with existing systems.

5 Proposed Architecture

The sample use case for OSCAR provided earlier in the paper illustrates the four aspects of the artefact management system:

Presentation OSCAR shall be responsible for taking the stored content and delivering it to the user agents in the appropriate format. Some transformation or repackaging of data is expected here. In addition this layer shall implement the security (largely access control) and maintenance (expiring and compressing old data) features the repository requires.

Indexing The indexing system shall provide a method of searching and navigating the artefact repository's stored meta-data. The information provided by this layer forms the basis of the operations offered by the presentation layer which shall then act on the artefacts themselves.

Metrics The behaviour of the system must be captured for later study for purposes of quality assurance, supplementing the artefact meta-data, further user-needs analysis and research. The metrics engine will perform this task

Storage Large volumes of data must be stored efficiently by the system. The format of this data must be chosen to facilitate the following:

- Speed and ease of transformation at the presentation layer
- Speed and ease of indexing the data for retrieval

Raw size is not a problem as large artefacts may be distributed around the GENESIS repository network as necessary. Consequently a highly structured and extensible data format or set of data formats is necessary. Two types of data shall be stored *Meta-data* and *Artefact Data*. These

may be stored together or apart though if stored apart they must be linked in some way.

In figure 2 the two data stores are shown separately as most indexing operations shall be performed using meta-data only for speed.

The storage requirements of meta-data suggest that a DBMS might be appropriate. Relations between artefacts (linking and dependencies) may be encoded as part of the stored meta-data in the DBMS and in addition the DBMS's structured query language provides the fine-grained query and edit control necessary to effectively search the stored data. Finally the DBMS itself shall facilitate the query process by optimising the storage of the data for rapid querying.

Since many DBMS implementations support transaction services either internally or through a distributed transaction manager, this layer of the system is an ideal place to situate all synchronisation services for the rest of the GENESIS system.

Artefact data has rather different storage requirements; efficient storage and retrieval of specific data is of greater importance than search performance. Since the search capabilities of a DBMS are not required for this type of data a set of flat files on a standard file-system will give the greatest performance.

These two conflicting requirements suggest a two-tier architecture consisting of a DBMS to store meta-data to enable users to find artefacts and a set of flat files on disk to store the actual data. All data in the DBMS should also be stored in the flat files so that in the event of DBMS errors the repository can be repaired transparently from the data held in files. Similarly the DBMS's transaction services shall be used to prevent concurrency problems (many-writers) from corrupting the disk data store. This mirroring approach will not noticeably affect the performance of OSCAR since whenever the meta-data on disk is changed the (much larger) artefact data will also require modification.

5.1 Artefact Description Requirements

OSCAR shall represent all the information contained within it as a set of artefacts all of which possess properties, some common across all artefacts and others possessed by specialised artefact types. This will allow the repository to treat artefacts either as generic pieces of information (for high-level searches etc) or take advantage of additional operations for specialised artefact types.

5.1.1 Artefact Structure

Artefacts shall be structured recursively; at the lowest level every file is an artefact. Additionally, artefacts shall inherit properties from their parent artefacts: consider the example of an XML document with associated image and other data files. At the highest level the document will be represented by a single artefact containing several sub artefacts for each of the document components. Assuming the document was produced by a single actor the author and other such information will be inherited from the parent artefact. See figure 3 for a representation of such an artefact.

In light of this design decision there are some necessary constraints on the structure of artefacts:

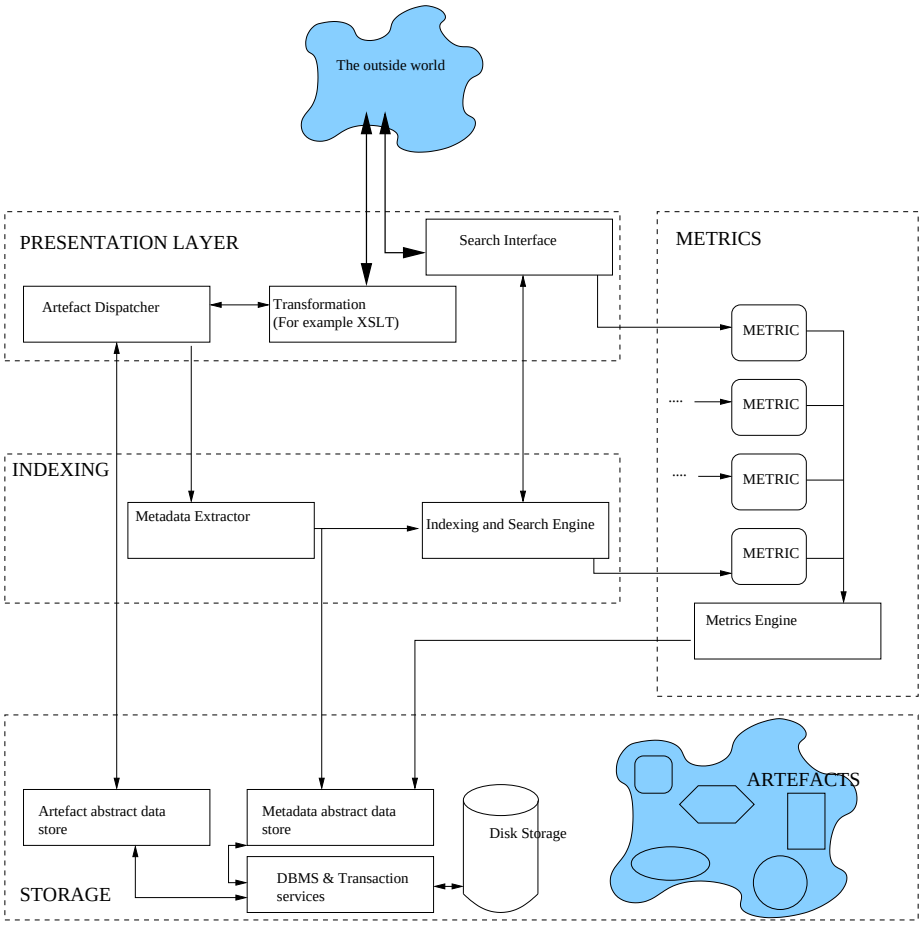


Figure 2: Overall Artefact Management System Architecture

- No two artefacts may "own" a single sub artefact. Put another way, artefacts may not inherit properties from more than one artefact. This decision has been borrowed from the design of the Java programming language and is intended to avoid the so called "Diamond Of Death" caused by conflicting properties. Though dual-ownership of an artefact is not possible, artefacts may be re-used by linking from another artefact and the appropriate construction placed upon the relationship when the artefact is presented.

- Explicit properties assigned to sub-artefacts override the corresponding properties in their parent artefact.

Aside from inheritance of properties from another artefact and encapsulation of sub-artefacts, artefacts also possess linking relationships with other artefacts. Consider our XML document and its associated images again. A single image (for example a chart) within the document may be generated by some process described elsewhere in the repository and though the image does not inherit properties from this process it *depends* upon that process to exist at all. Several

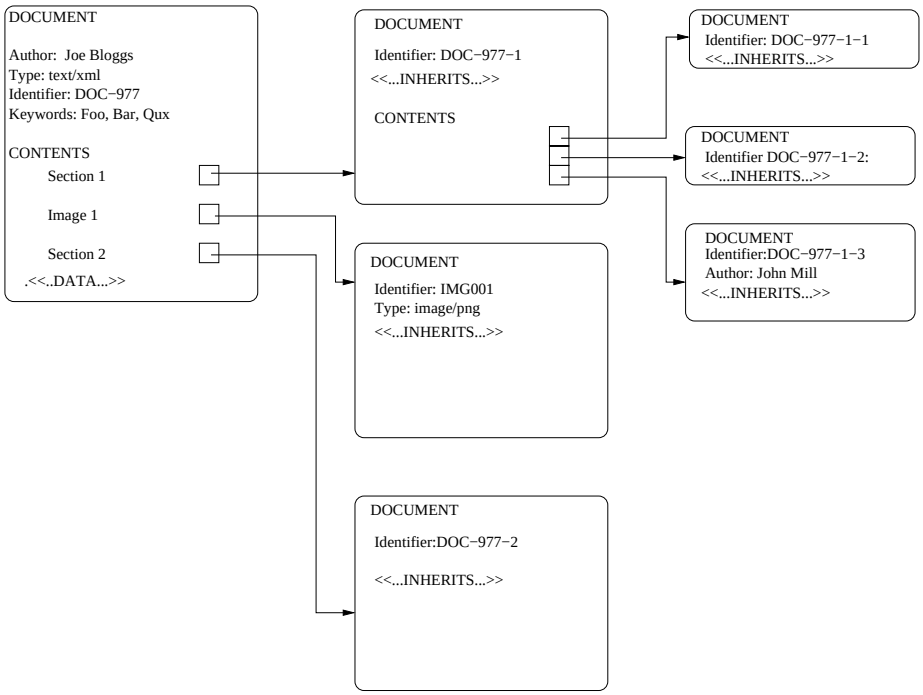


Figure 3: Structure Of An Example Artefact

types of linking relationship shall exist within the system:

Link Used to indicate artefacts associated with another by some informal link

Describes Used to indicate another artefact which documents the current one.

Suggested Indicates an artefact that is strongly associated with the current one but is not required to retain the current artefact. An example of such a relationship would be an artefact representing code for a client suggesting the related server-side software

Dependency Used to indicate an artefact that the current one depends upon

This resembles the Debian [Jackson and Schwarz, 1998] configuration management system for packaged software. Unlike the inheritance capabilities described above cyclic relationships are possible and must be detected and eliminated.

5.1.2 Data Formats

Any potential technologies for describing artefacts must satisfy several important properties:

Lightweight The barrier to entry for the technology (in terms of TCO, setup complexity etc) must be low enough to ensure OSCAR can be deployed quickly and easily.

Extensible Artefact descriptions including specialised artefacts should be extensible in a backwards compatible way at any time by the user

Open/Free and interoperable Since non-intrusivity is a requirement of the GENESIS system as a whole, OSCAR must employ interoperable technologies with excellent tool support to ease integration with existing systems.

Considering that OSCAR will already possess some form of DBMS to provide transaction services and store meta-data one potential technology is a relational database. In particular the entity relationship model supported by these databases will provide a powerful linking mechanism the integrity of which is enforced by the DBMS. Though the database schema is extensible on-the-fly without affecting data integrity, each specialised artefact type will usually require a new relation within the database. Finally not all RDBMSs deal well with large quantities of text data.

XML [Bray et al., 2000] offers another potential technology for artefact description. In particular its modular construction allows features from a variety of other XML document types to be imported with minimal effort. Additionally, XML-Schema provides a validation system equal to that provided by many databases. Finally XML is an extremely portable technology with excellent tool support across all platforms, easing the interoperability burden of the system.

5.1.3 Generic Meta-Data

All artefacts will possess *meta-data* to aid indexing and searching. As described earlier each artefact will have some generic properties and some specialised properties according to its role. The generic meta-data included/inherited by each artefact shall include:

- A unique identifier
- A descriptive title
- Original creator/actor with primary responsibility for the artefact
- Contributing authors
- Date created/modified
- Validity over time
- Link relationships, such as dependencies and reuse related links
- Subject keywords
- Information on the process used to create the artefact (if any)
- Access permissions.
- Versioning information

Additionally, change information and other logs shall be treated as meta-data wherever possible. This will provide a notion of presence as "footsteps in the sand"

Most of this generic meta-data can be structured at the presentation level using a specialised version of an extensible meta-data vocabulary. The necessary schema may be imported as a name-space into the schema or DTD used to structure the artefact descriptors.

5.2 Dynamically Created Meta-Data Requirements

This type of "documentation" within the GENESIS system is generated on demand and associated with its source artefacts. It shall be used to store items of information that are not necessary for a software artefact to be used in a software process but provide useful perspective or helpful secondary information. Such artefacts shall consist of:

- API/technical documentation of software,
- User comments and other informal documents,
- Change logs,
- Metrics output such as the level of interaction by other users with the target artefact and
- Organisational knowledge such as interesting references, best practice recommendations, anecdotes etc.

5.2.1 Styling

At the presentation layer the contents of the repository shall be transformed into information appropriate for the various users of the system whether machine (agent) or human. Since the data will be initially presented as XML, XSL and an appropriate transform language is appropriate to conduct this activity.

The behaviour of OSCAR at this level resembles existing document/content management systems that apply a stylesheet to content before transmitting it to (usually web-based) clients. Ideally styling and transformations should be entirely transparent to the user though since OSCAR cannot know all a user's requirements beforehand a facility should be provided to indicate to the system exactly what a user requires.

5.3 Indexing and Retrieval Requirements

OSCAR shall permit clients to search the system to retrieve artefacts that match their needs. Two methods to do this shall be presented to the user, both relying on one underlying search technology:

1. An Internet search-engine style interface permitting searching by keyword, boolean connectives and ordering of results by a variety of metrics. As most users are familiar with Internet search engines this interface will flatten the system learning curve.

2. An interface which allows an artefact descriptor to be passed in as a *comparator* Artefacts that match the comparator shall be returned The other search interface shall generate a comparator encompassing the search properties.

6 Proposed Classification Of Artefacts

Though each artefact may be treated as a generic object, instances of artefacts will be of specialised types with a variety of properties.

6.1 Software

The software artefact will represent both requirements and design artefacts, for example the information generated by CASE tools as well as source code for software modules. The *Software* artefacts should possess the following additional meta-data:

Tools The tools used to create the artefact (as references to a *Tool* artefact).

Re-use Information Any reuse specific information associated with the artefact. In particular, references to the *ProcessInstances* where this artefact is used and information relating to the composability of artefacts as reusable software components should be here.

Services Required Other software artefacts that the artefact depends on. Effort should be made here to prevent cyclical dependencies as outlined in section 5.1.1

Services Provided In the context of software code, this includes interface details for the module or component. In the context of a design or requirements software artefact this attribute should be null.

Both the Services attributes should wherever possible be described using standard Interface Description Languages such as the Web Services Description Language [Christensen et al., 2001].

6.2 Documentation

Documentation artefacts in OSCAR are any items of supplementary information added by users or automatically generated by internal OSCAR processes. They are not intended output from any software process but rather incidental information produced while a process is being enacted. All components of documents such as images should also be classified as *Documentation*.

Any documents produced as part of a formal software process such as Requirements or Design documents shall be designated a *Software* artefact.

Documentation generated by OSCAR's internal systems (such as the metrics engine) produce a special type of *Documentation* artefact known as a *Journal*. See section 6.8 for more information on this artefact type.

Documentation shall be stored as XML for which stylesheets and formatting objects exist to transform the content into distribution formats such as PDF or HTML.

6.3 Tool

The *Tool* artefacts describe external programs that may be executed and may optionally act upon a set of external artefacts. Behaviours described by the tool artefact may include the following:

- Allows the invocation of a third-party tool by GENESIS to produce OSCAR artefacts. The *Tool* artefact takes care of translating the tool's input and output via the presentation layer.
- Describes a metric. "Executing" the metric produces the current results. In reality of course the metrics will be calculated behind the scenes by the metrics engine.

To permit these behaviours the tool artefact must possess the following properties:

Invocation information Details of how the tool should be invoked by GENESIS clients. This allows the addition of a tool-server system like that employed by Oz at a later date and the interfacing of OSCAR to external software repositories.

Services required A list of other *Tool* identifiers that this *Tool* requires to perform its tasks.

Service provided An identifier describing the service the tools provide. For example, `emacs`, `vim` and `pfe` would all be examples of tools providing the `TextEditor` service.

The service keywords used in each system should be specified at system deployment or allowed to evolve in an ad-hoc fashion. Generally of course artefact authors will not specify abstract tools but the specific tools they used to create an artefact.

Target types An optional list of artefact types that the tool may act upon

Output types An optional list of artefact types that the tool produces

6.4 Actor/User

The *Actor* (also synonymous with *User*) artefact type in OSCAR is necessary for two major parts of the system:

Security Without access to a valid *Actor* artefact users of the system will not be able to perform any activity. This ensures that all actions (even simple reads by anonymous users) may be tracked using the metrics engine.

Author/Role Information Actor artefacts are used to fulfil the creator attributes of all artefacts and the role attributes of *ProcessElement* and *ProcessInstance* artefacts.

To facilitate both of these activities the *Actor* must possess the following properties:

- Whether the *Actor* is human or machine. Machine actors (such as agents) differ from *Tool* artefacts in that the former act externally from GENESIS whilst the latter are invoked by GENESIS to perform a specific tasks.
- Authentication information.
- Permissions information.
- Contact details (for human actors).

6.5 Legacy

The *Legacy* artefact is essentially a plain artefact which encapsulates a quantity of legacy data that is not stored in a way that OSCAR can understand or index correctly. For example, a large set of documents in proprietary formats that have not been managed as *Documentation* artefacts from the outset may be treated as *Legacy* artefacts until such time as they can be entered as proper artefacts.

Consequently the only additional attribute of a *Legacy* is a date to indicate when it is planned that the contents of the Legacy artefact will be added as true artefacts. Absence of the date indicates the Legacy will not be integrated.

6.6 ProcessElement

A *ProcessElement* describes part of a work-flow process. Like other artefacts they are recursively defined, allowing easy re-use of any appropriate subset of a process. The process itself shall be described by the use of a formal or semi-formal Work-flow Definition Language (WDL)

A *ProcessElement* shall have the following additional properties to support these tasks:

Inputs The artefact types that the process will consume (if any) when run.

Outputs The artefact types that the process will produce (if any).

Roles The *Actors* involved in executing this particular work-flow step

Prerequisites The pre-requisites for this step to execute successfully, such as other processes completing successfully etc.

A process element itself cannot be executed to implement a particular work-flow. Instead, the top level *ProcessElement* must be *instantiated* as a *ProcessInstance* which is linked to a specific version of the *ProcessElement*. Therefore, abstract *ProcessElements* can be used to define re-usable processes whilst concrete *ProcessInstances* can be used to record the execution of each process, providing useful information for future process design and analysis of working practices.

6.7 ProcessInstance

Once a *ProcessElement* is instantiated, all its sub-elements (if any) are also instantiated as *ProcessInstance* artefacts, preserving the relationship between them. To differentiate multiple instances of a running process from their template *ProcessElement* a *ProcessInstance* must have the following additional attributes:

Instance Identifier For all sub-instances this attribute shall denote the top level *ProcessInstance* artefact in the running process.

Element Version The version of the template *ProcessElement* that spawned this process. This is to ensure that if the best-practice process changes in the future completed *ProcessInstances* may be studied in the context of the original process

State The current state of this instance:

- Ready To Run
- Running
- Successful completion
- Unsuccessful completion

6.8 Journal

The *Journal* artefact represents versioning/change information for each artefact in the system. Each artefact will have an associated *Journal* to "store"[2] this information and present it as a coherent document. In particular the *Journal* will be used as a means of conveniently packaging the output from metrics *Tools* applied to the artefact. Since *Journal* artefacts are virtual, they possess no additional attributes. The metrics and other information in each *Journal* depend on the base artefact type.

For this reason Journal artefacts cannot be recursively defined like most other artefact types. Should a more concrete representation of a *Journal* need to be stored in the repository, the results of examining a Journal at a particular time may be saved as a *Documentation* artefact.

7 Implementation and Exploitation Strategy

Open-source software is typically developed incrementally [Raymond, 1999]. In the GENESIS project, the plan is to develop the software in a closed-source style for the first year, and to release the software into the open at that point. From that point onwards, development will take place in the open. Initially (during the first year), the development will follow a traditional closed-source lifecycle, which is currently at the stage of gathering and analysing requirements.

[2] In reality this information will be generated automatically when the artefact is retrieved since storing such information in discrete artefacts is unlikely to be efficient in space nor easy to keep current.

7.1 Exploitation Strategy

There are two categories of members of the GENESIS consortium: universities and companies. The universities will use the results of the project for teaching and research purposes (publishing papers, *etc.*, while the companies will exploit the results by using the tools and expertise generated during the project. These results will enable them to improve their software engineering processes using both the tools (*e.g.*, OSCAR) and the methods for workflow and configuration management.

8 Conclusions

The motivation and requirements for OSCAR, an Open Source software repository, have been described and discussed. OSCAR is a part of the GENESIS project, an initiative to develop an enterprise class software engineering environment for distributed collaborative working. The next phase is to complete the design of the entire GENESIS system in collaboration with our European partners. Further information may be found at `http://www.genesis-ist.org`.

The key requirement for GENESIS as a whole is that it is lightweight and non-intrusive and therefore will not disrupt existing successful organisational practices and furthermore may be adapted to facilitate those practices in any organisational context. OSCAR is intended to provide a distributed and dependable repository with a global name-space for managing all artefacts related to software engineering including process models, source code and other kinds of documents.

We have proposed an architecture for such a system earlier in the paper. The architecture is intended to facilitate active artefact management, a collective term for several automated facilities including change traceability and logging, dependency analysis, dynamic artefact data transformation and history/presence as meta-data. The heart of this technique will be in OSCAR's indexing and metrics components.

References

[Bray et al., 2000] Bray, T., Paoli, J., Sperberg-McQueen, C., and Maler, E. (2000). Extensible Markup Language (XML) 1.0 (second edition). Technical report, The World Wide Web Consortium.

[Christensen et al., 2001] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., and Research, I. (2001). Web services description language version 1.1. Technical report, The World Wide Web Consortium.

[Flores et al., 1988] Flores, F., Graves, M., Hartfield, B., and y Winograd, T. (1988). Computer systems and the design of organizational interaction. *ACM Transactions on Office Information Systems*, 6(2):153–172.

[Gail E. Kaiser and Yang, 1997] Gail E. Kaiser, Stephen E. Dossick, W. J. and Yang, J. J. (1997). An architecture for WWW-based hypercode environments. In *1997 International Conference on Software Engineering: Pulling Together*, pages 3–13, Boston MA.

[Haake, 1999] Haake, J. M. (1999). Openess in shared hypermedia workspaces: The case for collaborative open hypermedia systems. *ACM SigWEB Newsletter*, 8(3):33–45.

[Jackson and Schwarz, 1998] Jackson, I. and Schwarz, C. (1998). The Debian policy manual. Technical report, The Debian Project.

[Jacobson et al., 1999] Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

[Jiang et al., 1997] Jiang, W., Kaiser, G. E., Yang, J. J., and en E. Dossick, S. (1997). Webcity: A WWW-based hypermedia environment for software development. In *7th Workshop on Information Technologies and Systems*, pages 241–245.

[Kaiser, 1998] Kaiser, G. E. (1998). WWW based collaboration environments with distributed tool ser vices. *World Wide Web Journal*, 1(1):3–25.

[Lai and Tait, 1998] Lai, T.-S. and Tait, J. (1998). General photographic image retrieval simulating human visual perception. In *Proceedings of the ACM SIGIR'98 Post-Conference Workshop on Multimedia Indexing and Retrieval*, pages 17–28, Melbourne, Australia.

[Leblang, 1994] Leblang, D. B. (1994). The CM challenge: Configuration management that works. In Tichy, W. F., editor, *Configuration Management*, Trends In Software, chapter 1, pages 1–37. John Wiley and Son.

[Oinas-Kukkonen and Rossi, 1999] Oinas-Kukkonen, H. and Rossi, G. (1999). On two approaches to software repositories and hypertext functionality. *Journal Of Digital Information*, 1(4).

[Raymond, 1999] Raymond, E. S. (1999). *The Cathedral and the Bazaar: Musing on Linux and Open Source by an accidental revolutionary*. O'Reilly and associates.

[Sachs, 1995] Sachs, P. (1995). Transforming work: Collaboration, learning, and design. *Communications Of The ACM*, 38(9):36–44.

[Selvin, 1999] Selvin, A. M. (1999). Supporting collaborative analysis and design with hypertext functionality. *Journal Of Digital Information*, 1(4).

A Glossary

Artefact A product, whether formal or informal from the process of software engineering.

DBMS Database Management System. Various types exist.

Free Software A more rigorous type of Open Source, with the intention of producing of "Free Speech" (rather than "Free beer") software.

GENESIS Generalised Environment for process management in cooperative software engineering.

Meta-data In the context of OSCAR supplementary data stored for the purpose of indexing and searching the repository.

OSCAR Open Source Component Artefact Repository

OSS Open Source Software. Specifically software that complies with the Open Source Definition.

SCM Software Configuration Management

TCO Total Cost Of Ownership

WDL Work-flow Description Language. An example is the Web Services Description Language [Christensen et al., 2001]

KeyMan: Trust Networks for Software Distribution

Ben Laurie <ben@algroup.co.uk>
Matthew Byng-Maddick <mbm@aldigital.co.uk>

February 28, 2002

1 The Problem

Software, particularly open source software, is vulnerable to attack by distribution of maliciously altered versions of it.

The standard solution to this attack is to either distribute checksums (traditionally MD5 but SHA-1 would be better these days) or to PGP sign the distribution or its checksum.

Clearly distributing checksums is a very weak solution, particularly since these are usually obtained by downloading them from the same site as the software itself, which makes the checksum as vulnerable as the software, and hence of no value.

PGP signing is a better solution, but neither of the current ways of associating the signing key with the project are particularly strong.

One is to use identifiers with the appropriate email domain, or some kind of role email address. This is not a particularly strong way of connecting the package signer, because you can never be sure of the key signers' own authority to mark a key as authorised to sign the particular software. This is especially true where one authority has many projects that it manages.

Another common scheme is to include a list of valid signing keys in the software distribution - this does have some merit for validating future distributions from the same source, but clearly is of no value whatsoever in validating the first one someone downloads.

In both cases, the sheer difficulty of actually checking PGP signatures, which is a tedious manual process, is a major barrier to any rigour in this scheme.

2 Why is it hard to validate the signer?

As one of the authors is a director of the Apache Software Foundation (ASF), it will be used as an example throughout this paper.

In the ASF, there are around 50 different projects. Each one has a number of contributors, and typically any of those contributors can build any particular release of the software. Clearly, only the builder of the release is in a position to sign the software. This means that there are literally hundreds of people who can sign software that is distributed by the ASF.

But how does one check that they are entitled to? This is really quite troublesome. You might check that their key has been signed by some well-known member of the ASF, perhaps. But what does that actually mean? It means that well known person has validated their identity and does not imply that they can necessarily sign a release of Apache!

A solution that has been suggested to solve this problem is to have an ASF-wide signing key. Although this clearly helps, it just changes the problem from one of validation to one of management. Who has control of the key? If it is too many people, then the key is both reduced in value, because we have the original problem - there are people empowered to sign things they should not be - and also a new one - the key will presumably have to be revoked and replaced

on a regular basis, as the pool of eligible signers changes, if too few, then the problem is that the signer may no longer be the person who made the release.

It was thinking about these problems that led to the idea of KeyMan.

3 What is KeyMan?

KeyMan is a piece of software that permits the management of keys, certificates and signatures in a distributed and exportable network of trust.

The idea is relatively simple. First, there's the concept of domains. Every object can have its trust evaluated in a domain, the domain being an indication of who controls it. Sub-domains are considered to be contained within their super-domains, so, for example, the domain "apache" might be the domain for the whole ASF, and the domain "httpd.apache" the domain for the Apache Webserver.

The next idea is a trust level - this is simply a number between 0 and 1 inclusive, where 0 means "no trust" and 1 means "complete trust". Numbers in between do not have any strict meaning, for example .5 doesn't mean "I half trust this" - they are just meant to be an indication of degree of trust, providing at least some rough idea of how much trust should be placed in a thing. Their interpretation will probably be a matter of personal preference (the most obvious being that only things with a trust of 1 should be trusted).

The only other thing KeyMan uses is a trust depth. This is used when making a certificate of a key - one signs the key with a domain (or domains), a trust metric, indicating one's personal trust in that key in that domain and a depth, indicating how far the trust extends in signatures or certificates made by the key being signed. It acts rather like a Time-To-Live metric, in that things can reduce it by arbitrary amounts, but must always decrement it.

KeyMan objects are implemented as XML data structures, parts of which are signed, and parts of which are not. These will be described in a future paper. Being XML, they can easily be integrated into a larger XML export document, so multiple items (either objects or certificates) can appear as one document.

3.1 Why use a non-1 trust metric?

The principle reason we have thought of for using a metric that isn't 0 or 1 (clearly 0 is the same as not signing at all, of course) is to indicate that you have some confidence in a key, perhaps through circumstantial evidence, but not complete confidence.

How might you get circumstantial evidence about a key? One way is through repeatedly seeing it over a period of time signing software you use. Another is to see a certificate or set of certificates from keys you trust in other domains or even in the same domain but with inadequate depth. Of course, should this happen, it might be better to re-evaluate your trust in those certificates (including domains and depths), rather than signing the key itself with partial trust.

3.2 Why use a non-infinite trust depth?

The most obvious reason is because you want to give someone the ability to certify objects in a domain without giving them the power to delegate that ability. In this case, a depth of 1 is appropriate.

Another case is where an organisation wants to limit complexity in signing and certification, by only granting limited power to delegate - for example, the ASF board might sign the keys of the management committee for one of its projects with depth 2, allowing only people directly signed by that committee to do releases within that domain.

An infinite depth is appropriate where a key is trusted completely in a domain. For example, the ASF board members would probably sign each others keys in the Apache domain with infinite depth.

3.3 Evaluating Trust

In each installation of KeyMan, there exists a "root key", which is a key used by the owner of the installation. Any non-zero trust in any object will be traceable by a chain of signatures and certificates back to this root key. There are other models for doing this, but we feel that this is the most appropriate.

So, to evaluate the trust in an object, we must first know the domain of that object. We then find all possible paths of signatures or certificates, back from that object to the root key, where every certificate is in the domain of the object or a super-domain of it, and the depth on each certificate is sufficient to reach the object. Then all the trust metrics on each path are multiplied together, giving the path trust for that path. The highest path trust is then the trust in the object.

The existence of multiple paths at reasonable trust levels can be an example of the circumstantial evidence mentioned above, which might make the user more likely to make their own certification of a given object.

4 KeyMan: the software

So now we know how KeyMan works, what does the software do? It manages all of the above objects, and their associated signatures and certificates, calculates the trust in supplied domains, allows you to sign and export objects, and also to download KeyMan objects from URLs and check them in one operation (note that the object can be a software distribution plus associated KeyMan signatures).

It displays the chains of signatures and certificates leading to trust on an object graphically, with helpful information available for all of the objects involved.

It allows you to sign things, and to export your certificates, signatures and objects themselves in a format suitable for other KeyMan users to import.

And it does this all both graphically and from the command line.

Incidentally, all signatures are currently actually PGP signatures, using GnuPG to manage them, but there is no reason they should not be any kind of public key signature. The architecture has been designed to allow for this. It is also possible to import pre-existing PGP signatures to be checked as part of the certification chain. Because these are signatures rather than certificates, they default to a trust metric of 1, an infinite depth, and a domain of ".", the root-level domain.

5 Usage Scenarios

How KeyMan would be used in practice is probably best illustrated by considering four different types of user, who conveniently span more or less all possible uses of KeyMan. Even more conveniently, each type of user builds on the previous one, each using the software in a slightly different way.

These four types of user are software developers (assumed to be members of a large team), sophisticated end users, package maintainers and naive end users.

5.1 Software Developers

Software developers really have two things they have to do with KeyMan - the first is to sign each others keys. This is normally best managed by first signing in the usual PGP web-of-trust way, indicating their trust in the key being owned by its claimed owner, then signing in KeyMan with appropriate domain (i.e a sub-domain of the overall project), trust metric (1, usually) and depth (depending on project's policies) in order to include them as a member of a project.

In the case of the ASF, the way this would probably work is that the ASF board would all sign each other's keys in the "apache" domain with infinite depth, and get as many of the other ASF members and contributors as possible to sign their keys in the same way (and, indeed, anyone else that can be persuaded). The board members would then sign the keys of each project management committee (PMC) in that PMC's domain (which would be a sub-domain of "apache", for example, "httpd.apache"). The PMC would, in turn sign keys of the project maintainers in appropriate domains (which might be sub-domains of the PMC domain, or the PMC domain itself). These maintainers can then sign project releases.

The net effect of all this activity is that anyone who can find a path of trust to any ASF developer should end up being able to trust all software released by the ASF (via developer → Board → PMC → maintainers → developer), but in a way that is scalable and maintainable.

The other thing a developer has to do is release software. This is really just a matter of signing the tarball, setting its domain and setting a URL where it (and future versions and the certificates) can be downloaded, and then putting the resulting KeyMan exports at that location.

5.2 Sophisticated End Users

A sophisticated end user will want to make their own decisions, control their own life as far as trust is concerned (see Naïve End Users). This means that they will try to find chains of trust that lead to the software they need to check, through friends or colleagues. It is typically this kind of user that will end up signing keys with partial trust, being unable to find routes that give them complete trust.

Of course, they can use KeyMan as a tool to help them explore the trust network and see if there are keys they may be able to validate to improve their position.

And if they have friends or colleagues who have trust that improves their situation (assuming they trust them, of course), then they can get them to export their trust graphs and send them (by email for example).

5.3 Package Maintainers

A package maintainer is in much the same position as a sophisticated end user, from the point of view of checking signatures and certificates, except that they may have better contacts through their organisation to enable trust paths to be found.

Once they've verified the source, most package maintainers will then apply patches and other tweaks appropriate to their way of doing things, and then will probably produce some kind of rolled-up version of the modified/configured software. They will then sign this with a key that has been certified in the domain of their organisation. The idea behind this being, of course, that the distribution can have a single root key which, if trusted, will automatically certify all software in that distribution.

Naturally, the distribution will have hierarchy of domains similar to the ones maintained by software authors - reflecting their own internal network of packagers and their managers.

5.4 Naïve End Users

The typical naive user will most likely be using vendor-supplied packages on their system, and will be compiling very little from the original source. They would set themselves up to trust the vendor root key, to a depth such that they would trust the package maintainers of the vendor's packages (probably, in practice, infinite depth, since they should trust the vendor to manage their keys and certificates correctly anyway).

The vendor's key could well be distributed with the installation media for their distribution - including KeyMan, of course. KeyMan would then be used as part of their installation and upgrade process, and would be largely transparent to the user. The only thing they'd have to do in a normal situation is to generate their own root key and sign the vendor's root key with it - of course, the distribution would probably largely automate this process as part of the install.

6 Future work

One of the things we are still working on in KeyMan is certificate revocation. In essence this is simple, but there are a number of issues.

The main issue is that we would like to not invalidate all past certificates made by a key that has been revoked at a particular time. This requires us to have timestamps, signed by third parties[1]. In essence, timestamps are relatively simple - all one needs to do is send an object to someone, containing a hash of the thing to be timestamped and a claimed time[2], which the timestamper then signs if they believe the claimed time, including a time at which they signed[3].

It is clear that in the worst case we can abandon the idea of timestamps and simply rely on the time the object actually arrived at the relying party (that is, if there is a revocation that claims a key was revoked at time t_X, and we first saw an object signed by that key at time t_Y, then if $t_Y < t_X$, we can still believe the signature). Obviously this gives an attacker a denial of service attack, but it does prevent them from forging valid signatures.

Another thing we'd like is to allow anyone to revoke anyone else's certificates - this makes sense with the model of the network of trust - whether you believe the revocation depends on whether you trust its author, rather than the simplistic model where only the holder of the key can revoke it.

In any case, there are a number of complications resulting from revocation and timestamping which we are still working on, and which, no doubt, will be the subject of further papers.

7 Closing Remarks

An implementation of KeyMan written in Perl is available from an anonymous CVS download, described in `http://keyman.aldigital.co.uk/`.

KeyMan was designed and written by Ben Laurie and Matthew Byng-Maddick for A.L. Digital Ltd., and is an effort sponsored by the Defense Advanced Research Project Agency (DARPA) and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0537.

[1] this is needed otherwise a compromised key could obviously then be used to claim any time in the past

[2] this is needed because if the timestamp signer is human, they may not respond immediately

[3] this, we believe, is a good idea, because it allows the relying party to make their own decisions about how large a time window with which to trust the signer