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:

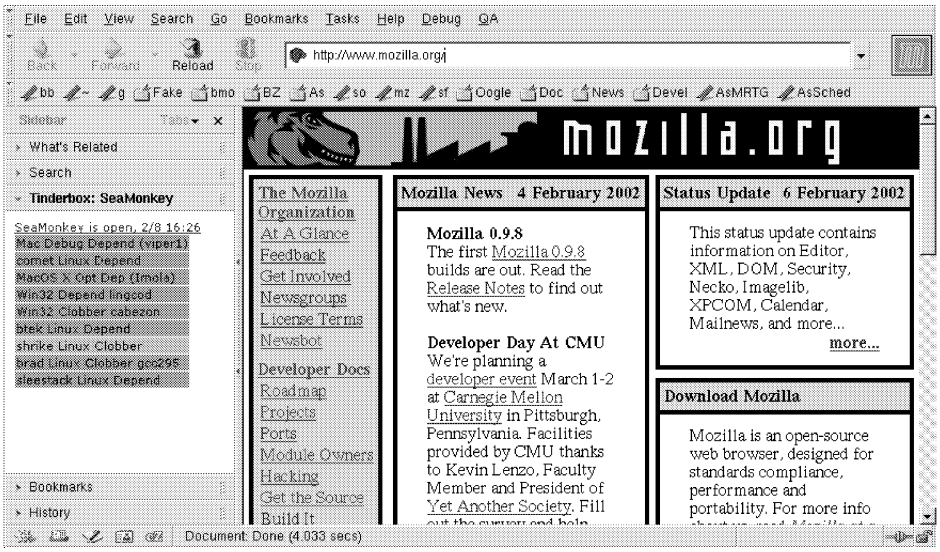

Figure 1: A screen capture of the Mozilla Web browser

- 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,

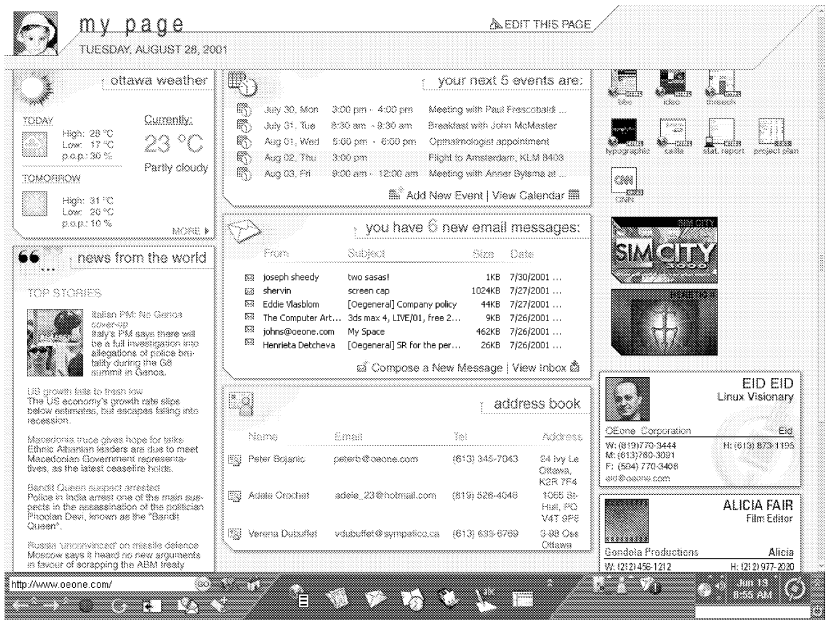

Figure 2: A screen capture of the OEone Personal Portal, a product derived from Mozilla source code.

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.

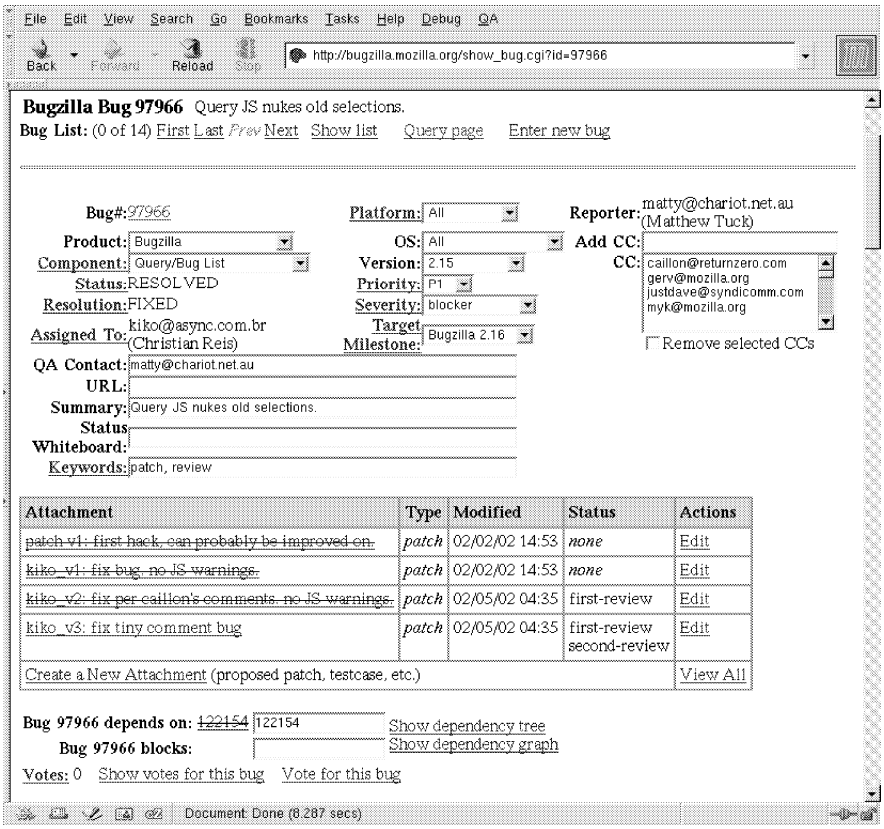

Figure 3: A screen capture of a Bugzilla bug form.

- **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.

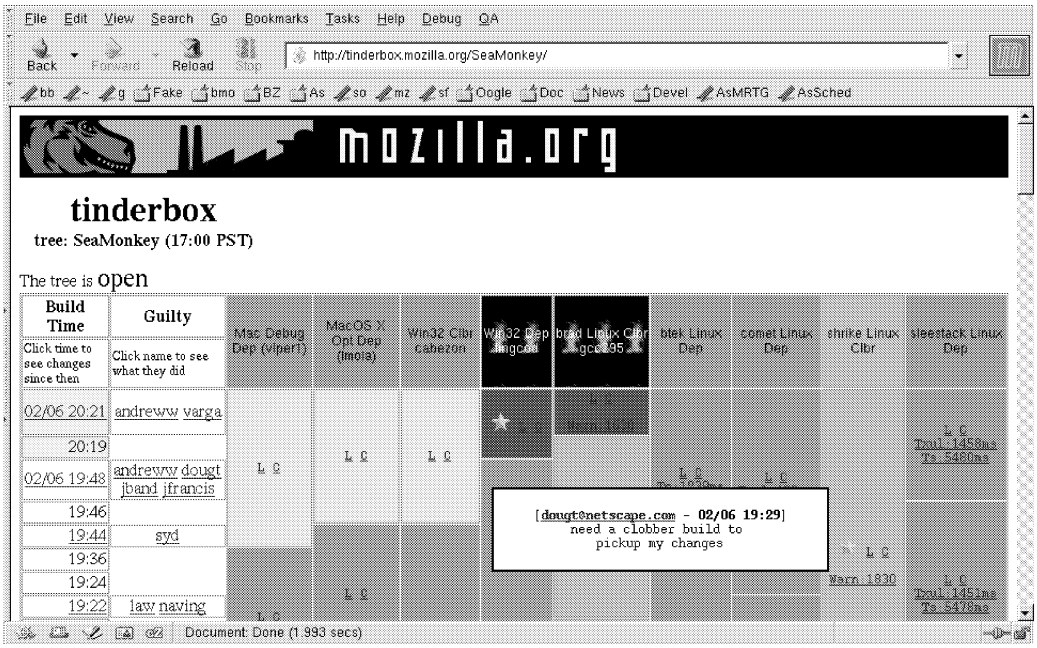

Figure 4: A screen capture of Tinderbox output.

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.

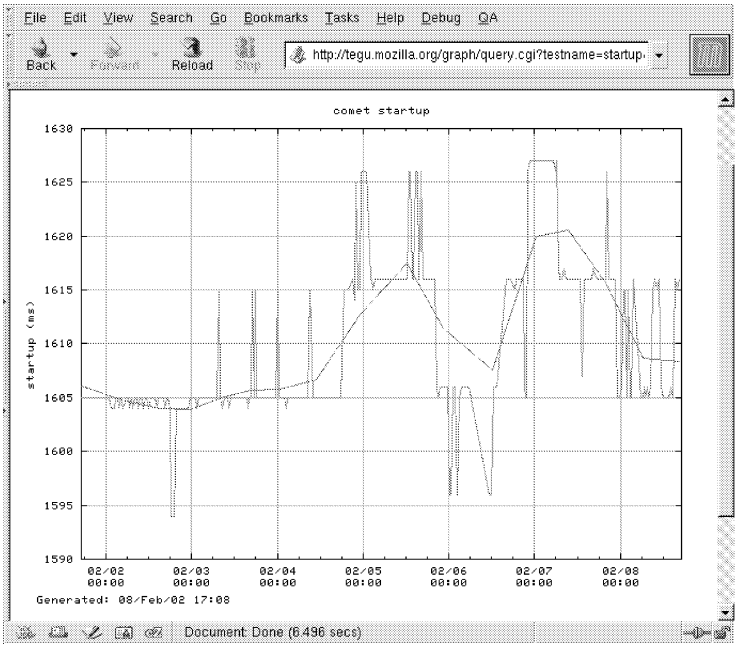

Figure 5: A graph of the time it takes to start the Mozilla browser, varying over days.

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.

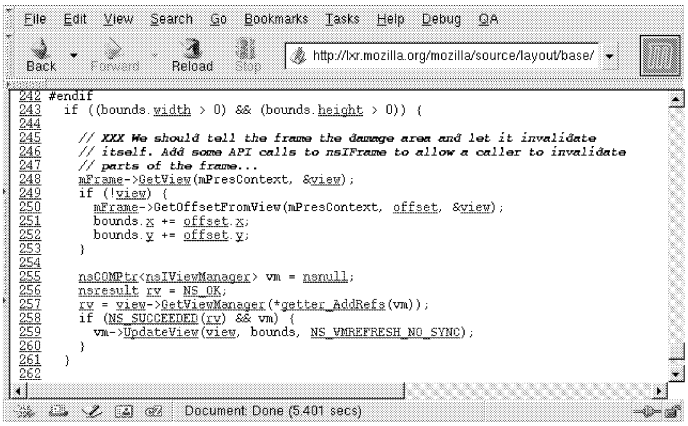

Figure 6: A screen capture of LXR displaying a page of the Mozilla code.

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.