Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 166 (2007) 81–91
www.elsevier.com/locate/entcs

Software Evolution from the Field

An Experience Report from the Squeak Maintainers

Marcus Denker [2,3]

Software Composition Group
University of Bern, Switzerland

Stéphane Ducasse [1,3,4]

LISTIC
Université de Savoie, France

Abstract

Over the last few years, we actively participated in the maintenance and evolution of Squeak, an open-source Smalltalk. The community is constantly faced with the problem of enabling changes while at the same time preserving compatibility. In this paper we describe the current situation, the problems that faced the community and we outline the improvements that have been introduced. We also identify some areas where problems continue to exist and propose these as potential problems to addressed by the research community.

Keywords: Squeak, open-source, Maintenance, Software Evolution, Tool support

1 Introduction

Over the few last years, we actively participated in the development and maintenance of the Squeak open-source project [13]. We were responsible for the the 3.7 and 3.9 official releases and participated in the releases 3.6 and 3.8. During this activity we faced the typical situations that developers are facing daily: bloated code, lack of documentation, tension between the desire of improving the code and

[1] Email: sduca@univ-savoie.fr

[2] Email: denker@iam.unibe.ch

[3] Denker and Ducasse gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

[4] Ducasse gratefully acknowledges the financial support of the French National Recherche Agency (ANR) the project "Cook: Rearchitecturing Object-Oriented Application 2005-2008".

doi:10.1016/j.entcs.2006.08.003

providing an unchanging basis for all developers. Furthermore, Squeak suffers from some of the problems of a real open-source project: lack of man-power, distributed development with contributors at multiple locations and lack of long term commitment.

In this paper we present the problems that the Squeak community has been facing and describe the improvements that have been introduced. Moreover we sketch future approaches we would like to put in place. Finally we analyze the open problems and propose these as future research topics that may be of interest to the research community on software evolution.

Some researchers analyzed software evolution and its challenges and proposed taxonomy of changes [6,16,17]. There has been other research into open source development and evolution [18]. In this paper we want to give a summary of our own experiences as developers and researchers maintaining a large and complex open-source environment.

First we present Squeak [13] by providing some measurements that characterize it and its evolution. Then we shed some light on the philosophy of Squeak by describing the current forces in the Squeak community and we present the overall development process. We outline the common problems the community has been faced with and the improvements it has adopted both from an implementation and from a more process oriented point of view. Subsequently we present the open problems that still exist. We analyze the opportunities we missed retrospectively and sketch the approach we envision to improve the situation.

2 Squeak

While the appearance of Squeak may mislead a novice user, Squeak is a really large and complex system containing more than 1600 classes and 32,000 methods in its latest public release (3.8 basic). Squeak includes support for different application domains:

- two large graphical user-interface frameworks, Morphic and MVC,
- an IDE, including an incremental compiler, debugger and several advanced development tools,
- a language core and all the libraries enabling it (both simple and complex) including concurrency abstractions,
- multimedia support including: images, video, sound, voice generation,
- eToy [1], an advanced scripting programming environment for children,
- various libraries such as compression, encryption, networking, XML support.

All this code is part of the main Squeak distribution. In addition to that, there is a growing number of external packages available on the *SqueakMap* repository (over 600 packages), thus showing the vitality of the Squeak community.

Release	Year	classes	methods	LOC	testCase classes / testCase methods	Introduced Functionality
3.5 Full	2003	1811	41408	322656	20 / 158	
3.6 Basic	2003	1338	33277	246752	0 / 0	
3.7 Basic	2004	1544	35526	261315	133/ 1180	Tests
3.8 Basic	2005	1659	37952	281694	148 / 1426	Internationalization Support
3.9beta	2006	2083	45842	334334	220 / 2506	Traits, Monticello, Services, ToolBuilder

Table 1
Overall Squeak growth.

2.1 Measurable Facts

The table 1 shows some metrics to give an idea of the size and growth of Squeak. The Squeak community realized that the system is beginning to contain too much unneeded functionality. Hence, in Squeak 3.6 certain functionality such as the web browser and email clients was externalized. In Squeak 3.9, an extensive amount of infrastructural work has been introduced: new language feature (Traits)[20], a versioning system (Monticello), a registration mechanism (Services), an abstraction layer for the UI framework (ToolBuilder).

To understand the Squeak development dynamics we have to consider its open-source nature and also to analyze not only its growth, but the actual changes taking place. Indeed some actions (addition and removal of functionality) may result in the numbers of methods and classes remaining constant. One good starting point when considering the evolution of Squeak is to consider the approximate number of patches that have been integrated between releases. These patches vary in size and complexity, so the number can only be taken as a hint of the change that is done:

Version	3.5	3.6	3.7	3.8	3.9
Patches	10	240	560	600	n/a

Table 2
Number of patches applied.

2.2 Communities and Different Agendas

The Squeak.org distribution is used by a wide range of diverse projects. Some of them have their own open-source communities, other are closed research or commercial projects. Here we list the most important projects that are based on the common distribution.

Squeakland. The main focus of this distribution is educators of primary schools. The main interface is the Etoy system [1]. Squeakland is widely distributed in Japan and US. The end user does not interact directly with Squeak but the Etoy layer. Code quality, ease of extension, untangling and tested code are not the main focus of development.

SmallLand. It is a community similar in scope to Squeakland: the children are older, and the project directly manages an installed base of its own squeak dis-

tribution on 80,000 PCs in schools in Spain. SmallLand has been completely localized in Spanish: one of the largest contributions of this work was the framework for translating Squeak to other european languages, a change that affected a large percentage of the code base. Lot of changes to the graphical user interface were made to support a better desktop. They are concerned with the code quality.

Seaside. It is a framework for developing sophisticated dynamic web applications. Seaside combines an object-oriented approach with a continuation-based one[8]. Seaside offers a unique way to have multiple control flows on a page, one for each component. This community is concerned with the robustness and scalability of the libraries and the tool support. The design and ease to extend the system is also one of their concerns. This community developed a package and version control system (Monticello), that is used in the latest version of Squeak. This enables the developer to write components that are highly reusable and that can be used to compose complex web applications. Seaside is used by a number of companies for commercial development.

Croquet. It is a 3D shared multi-user environment. Users on different machines can use Croquet to explore a three dimensional space together, collaborate and share. Croquet uses a new Peer-to-Peer model for synchronization, allowing the system to function without needing central servers.

Tweak. Tweak is a next generation implementation of a UI and scripting system based on the lessons learned with both Morphic and EToys. Tweakers are concerned with code quality but in general are building layers on top or besides the base system.

We have learnt while maintaining Squeak that there are two main driving forces, and this has lead us to coin the phrase *'egocentric syndrome'*: frequently one group of developers within the community wants to have problems that affect its interests fixed but at the same time demands that no other changes should be made for fear of breaking existing code.

2.3 The Squeak Development Process

As with a number of open-source projects, the Squeak community is an open-source movement that lacks real financial support or a sponsoring organization. From its inception to the release 3.4, Squeak was mainly developed by SqueakCentral, a single group of researchers around Alan Kay. From 3.4 to 3.8, the development community grew and became geographically distributed with many developers and small teams.

The general process is the following: the developers produce fixes or enhancements that are sent to the mailing-list and are collected in a central database. Then, the group of maintainers responsible for a given release look at the changes and decide whether to integrate them in the current release.

3 Common Problems

In this section, we elaborate on some code level problems that face the Squeak community. We are aware that such problems are neither exceptional nor specific to Squeak. Our goal is to document them for researchers who focus on the area of software evolution. The following section shows the improvements that have been applied and an evaluation of the obtained results.

Tangled Code. Historically, and as with any original Smalltalk implementation, Squeak was developed as one single system. The fact that subsystem boundaries were not explicit (for example by means of a package system) leads to a system with a lot of unneeded (and often surprising) dependencies. For example, the compiler depends on the GUI. These dependencies hamper modularisation and evolution: there is a high risk of introducing bugs, as a change in one part of the system can break another unrelated part.

Dead Code and Prototype Code. Squeak was originally intended as an environment for prototyping new ideas. Very little effort was spent on refactoring [11] and it was never systematically cleaned. Often the experiments and their extensions have remained in the system resulting in dead code and often incomplete functionalities.

Evolution Dilemma. The previous problems are minor compared to the evolution dilemma. As multiple stakeholders exist for Squeak, they require that it is a *stable* basis for development. However at the same time, those same projects also carry out refactorings to improve and bugfix the base system, and by doing they generate instability.

Thus the evolution dilemma is to preserve stability, while at the same time ensuring that it is possible to make changes.

The first two items are what Lehman [14] called entropy: The entropy of a system increases with time, unless specific work is executed to maintain or reduce it. The dilemma comes from the fact that reducing the entropy will have impact on the clients using the system.

4 Language-Level Improvements

One way to address the problem of evolution is by using well-known software engineering techniques and object-oriented design heuristics. [2]. In this section, we give examples of how these techniques were advantageous for Squeak. A challenging problem with most software engineering improvements is that they imply refactoring the system towards a better, more evolvable architecture. Hence we need to change the system to facilitate future changes.

4.1 Deprecation

Deprecation mechanisms are a well known technique to allow for gradual adoption of new interfaces. The idea is that all public methods that are no longer needed (*e.g.*,

because of adopting a new, different API) are not deleted immediately. Instead, a call to the deprecation method is added with the indication to the user that another method should be used:

```
Month>>eachWeekDo: aBlock
    self deprecated: 'Use #weeksDo:'.
    self weeksDo: aBlock
```

This method raises a warning in debug mode by calling the #deprecated: method, but allows the original method to execute when in production. In Squeak, deprecated methods are retained for one major release: methods that were deprecated in the development of 3.8 will be removed in 3.9.

Version	Number deprecated methods	Version	Number deprecated methods
3.6	100	3.8	24
3.7	104	3.9	55

Table 3
Number of deprecated methods.

The usage of the deprecation mechanism decreased over time (Table 3). This could indicate one of two things: it may be an indication that a better mechanism is needed or that the system is in fact stabilizing.

Next Steps. It is not possible to tag an entire class as deprecated. Another disadvantage is that the mechanism as a whole is very course grained: evolution is happening all the time and is not restricted to major releases. Deprecation does not scale towards short cycles of change.

In the future, the authors would like to explore how refactorings could be stored and then be available for automatic replay when a deprecated method is called. We also want to address the issue of deprecation of classes.

4.2 Divide and Conquer: Modularizing the System

A large effort is underway to modularize the system. The latest 3.9 beta version is composed of 49 packages excluding tests with an average of 40 classes per package.

Next Steps. The modularization was not driven by advanced analysis[21]. Having additional tool support to analyze existing dependencies is a first important step.

4.3 Registration Mechanisms

Modularization requires that a system be divided into units to be able to be built from a subset of its actual components. In earlier versions of Squeak there were many cases where simple configuration required that the code be changed: for example, when adding a tool, the code that built the menu needed to be extended to handle the new tool. The community successfully applied Transform Conditional to Registration [7] *i.e.*, *registration mechanisms* were introduced allowing for registration of new services without having to modify code. This reduced considerably the friction between different tools and facilitated the dynamic loading and unloading of tools.

4.4 New Abstractions

Squeak's multiple GUI frameworks proved to overcomplicate the building and maintaining of tools. There were interdependencies between MVC/Morphic and some features are only available in one framework. The solution the community adopted was to build an abstraction layer, named ToolBuilder, representing the common framework elements and to rewrite the tools to use the ToolBuilder framework, which encapsulates the GUI framework used.

4.5 Refactorings

A lot of small refactorings were carried out over the recent years, leading to iterative improvements. Some large refactorings were supported such as a complete rewrite of the network subsystem. Large refactorings were possible when the changes did not cross-cut a lot of packages but were localized in one package.

4.6 Enabling Changes: Tests

Over the last years *unit testing* has gained a lot of recognition, as it facilitates change [3]. Tests support the developers ability to change code as they can quickly identify the code that they broke as a result of their changes. Squeak has for a long time contained SUnit, a testing framework. However, it took time for developers to recognize the importance of tests and adopt the practice of writing unit tests. To increase the awareness of unit testing, the distribution of Squeak 3.7 included not only the framework, but the tests themselves. This increased the general awareness among Squeak developers. Over the last releases, test coverage increased, with over 6 times more tests in 3.7 than in 3.6, and nearly doubling in later versions (see Table 4).

Version	Number tests	Version	Number of Tests
3.5	171	3.8	1347
3.6	0(external package)	3.9	2157
3.7	1124		

Table 4
Number of tests.

Next Steps. Having tests is a first step towards enabling changes and documenting the system in a synchronized way. The next step is to assess the quality of tests using test coverage tools. We plan to use tests to support the development by having a test server providing continuous feedback on the state of the latest versions [4].

5 Process Improvements

Besides strictly technical improvements on the code level, process-oriented approaches have been adopted. The community started to adopt a number of process

improvements such as better tools and real bug tracking. Other promising improvements, for example an automated build system, will be implemented in the future.

5.1 Better Packaging Tools

Squeak development used to be based on *change sets*: simple lists of modified or added methods and class definitions. Changesets can be saved into text files and send around (*i.e.*, via email). However, as soon as multiple developers are involved and are required to get synchronized, changesets do not scale: resolving conflicts is very tedious and turnaround time for integration is very large.

In contrast many of todays development systems use elaborate versioning systems like CVS[9]. This inspired the commercial sub community around Seaside to develop a versioning system called Monticello that uses a powerful merge algorithm, and a server, SqueakSource, that projects can use for managing and storing their code base in a distributed setting.

SqueakSource and Monticello have been proven to be very successful. SqueakSource counts over 420 registered projects and 480 developers. Several sub communities started to have their own SqueakSource servers to manage their projects. Starting with the development cycle for 3.9, Squeak uses a dedicated SqueakSource installation and Monticello for managing the complete squeak code base.

Next Steps. Having packages that are first class entities and better tool support for packages is desirable.

5.2 Bug tracking

Earlier releases of Squeak were developed without any provision for bug-tracking. Naturally the Squeak community were faced with the problem of managing bugs: users did discover bugs, developers offered fixes, some of which were included in the next release. But there was no control over how many bugs had been identified, or to monitor whether or not they had been fixed, or in fact what the general overall state of Squeak was. This situation has been improved with huge success: Over 2500 bugs have been reported using Mantis [15], a web-based bug tracking system.

5.3 Future improvements

We have learned from experience that we should have invested more time and effort in tools that would support the development process of Squeak. Our goal now is to actively address this.

Automatic Build tools. Currently, the task of merging changes is labor-intensive and time-consuming. A goal is to leverage an automatic build system to eliminate all the tedious work that currently face maintainers. Automatic build systems are a well known technique and will be a valuable addition to the set of tools for Squeak development. Integration with an automatic test server is another major aspect of the build system that will have to be tackled.

Automatic test runs. Unless tests are run frequently, they are not very useful. Currently executing the entire testsuite takes well over 10 minutes, thus discouraging developers from running tests. We propose to solve this with a server that runs all the tests automatically, at least once a day, and sends reporst to the developers. Such a server is currently in development.

Even better tools. The current tools represent improvement in the development process of Squeak, but they need to be improved even further: *e.g.*, the response time of Monticello is slow for large code bases.

6 Language Design Dreams

Apart from our involvement in the maintenance and evolution of Squeak, we are primarily language design researchers. We try to step back from our research background and avoid to mix experiment in language design with robust, scalable solutions.

While we are skeptical that the complex problem of enabling smooth but fast evolution of a large system can be solved by just adding a couple of new language features, we think that new language features are a very promising field for future research on software evolution: How can a language [19] support change and evolution?

Here is a list of challenges for the software evolution researchers.

Better support for modularity. While Squeak now has a package mechanism, it does not have a real module system. Packages are only deployment time and code management artefacts. Squeak packages do not include any notion of visibility (contrary to Java packages) [5]. It would be interesting to evaluate how a module system, by providing a scoping mechanism, could help to better structure the system.

History as a first class objects. Squeak is a reflective language [10]: *e.g.*, classes and methods are objects, the system has a complete, object-oriented description of its own structure, available for introspection and change.

We would like to understand the costs of extending this reflective model to take into consideration the data that is important for evolution: the history of the system should be represented as a first class entitiy [12]. Having first class history should support the possibility to express the following questions: Why was this method changed? What else has changed when this method was changed? When did this test break for the first time? Which change affected the performance of the system?

Beyond Deprecation. Deprecation, as described earlier, does not really help with the evolution problems the Squeak community faces: the problems is that some clients may want to migrate to a new release but only incrementally for certain parts of their applications. If we have the complete history of the system available, do we really need to force clients to use the latest version only? Would it be possible to have different clients using at the same time different versions of the

same components? The operating systems offer the possibility to specify which library version to use. Can such a model be realized at the level of a programming language so that multiple versions are able to co-exist?

7 Conclusion

The evolution of Squeak with its diverse community of developers and open issues (*i.e.*, decentralized development, modularization need) is a challenging task. The Squeak community achieved lot of progress over the last years such as a first cut at modularizing the system and the increase of unit tests. However, apart from environment enhancements such as Monticello few tools have been built to analyze the code base of Squeak. We have identified the need for them to help improving the system.

In addition, over the last few years, we have experienced the problems of having to introduce changes while at the same time supporting stability for existing users. We highlight the fact that there is no language support for the notion of evolution in todays programming languages. Current languages are only capable of describing one version of the system. Scoping changes and allowing multiple versions while retaining a simple language is an open challenge that the research community is facing.

Acknowledgments.

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1) and french ANR for the Cook: rearchitecturing object-oriented applications. The authors would like to thank all the Squeak developers and Orla Greevy and Adrian Lienhard for the reviews on the early versions of this paper.

References

[1] B.J. Allen-Conn and Kimberly Rose. *Powerful Ideas in the Classroom.* Viewpoints Research Institute, Inc., 2003.

[2] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion.* Addison Wesley, 1998.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.

[4] Kent Beck and Martin Fowler. *Planning Extreme Programming.* Addison Wesley, 2001.

[5] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, 2005.

[6] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005.

[7] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2002.

[8] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of ESUG Research Track 2004*, pages 231–257, September 2004.

[9] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Coriolis, 2001.

[10] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, October 1989.

[11] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[12] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[13] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

[14] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[15] Mantis. http://www.mantisbt.org/ (last accessed 8/8/2006).

[16] Tom Mens, Juan F. Ramil, and Michael W. Godfrey. Analyzing the evolution of large-scale software: Issue overview. *Software Maintenance and Evolution: Research and Practice*, 16(6):363–365, November 2004.

[17] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2005)*, 2005.

[18] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[19] Oscar Nierstrasz and Marcus Denker. Supporting software change in the programming language, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[20] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[21] Daniel Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.