Supporting Software Change in the Programming Language*

Oscar Nierstrasz and Marcus Denker
Software Composition Group
University of Bern
www.iam.unibe.ch/∼scg

Keywords: sofware evolution, languages, traits, classboxes.

1 Introduction

Software changes.

Software systems are constantly modified and extended. Versions are split and sometimes merged. Software is refactored and occasionally reengineered.

Programming languages, however, provide little support for software change. Pretty well every mainstream language (and the exotic ones too) adopt the viewpoint that a program specifies a single, consistent viewpoint of a software system. The fact that this system will change over time is not, and cannot be a part of this specification. Furthermore, support for dynamic change is typically limited to idioms and design patterns that encode run time flexibility, rather than by representing it directly in the program specification.

We are by now used to the response; "that's not a language issue." Indeed, this response has considerable merit — it is much easier to reason about programs and about the semantics of the programming language itself if we assume a consistent world-view for any given program, and defer issues of program change to the meta-level of programming tools and environments that manipulate these programs.

But let us suspend our disbelief for a moment and ask ourselves, *What would a programming language look like that offered support for software change?*

In this paper we briefly review various kinds of software changes and the issues that arise from them. As a consequence, we propose research into programming languages with explicit support for representing first-class changes, and for manipulating and merging multiple viewpoints of evolving software systems.

2 Kinds of Changes

Software systems undergo different forms of change. These range from routine, day-to-day changes, to more radical kinds of change. Let us briefly survey the broad categories of change and the kinds of problems they pose for programmers [LS80, DDN02].

- *Bug fixes:* Even mundane bugs pose the problem — *"Will fixing this bug cause something else to break?"* Programming languages do not help programmers keep track of implicit dependencies (which is why they are implicit in the first place). More deep-seated bugs may be hard to fix because they are a sign of architectural problems. Fixing them either entails a "hack" — which leads to further maintenance problems — or entails reengineering towards a better architecture.

*OOPSLA Workshop on Revival of Dynamic Languages, October 2004.

- *New features:* Many changes concern user requests for new features. The difficult features to add are those that cross-cut components of the existing system, impacting the entire architecture.

- *Split and adapt:* Entire applications or portions thereof are often cloned and adapted to meet the needs of new clients. Although this can be an effective way to deliver results quickly, the code base quickly gets out of hand, leading to maintenance difficulties. Programming languages do not really help to keep track of the relationships between separate projects. Attempts to factor out libraries and frameworks of common components can be frustrated by large numbers of minor differences between the various applications.

- *Merge and integrate:* New features developed for one client may need to be merged and integrated into applications for other clients. This is inverse to *Split and Adapt*: after splitting, the system needs to be integrated back into a consistent body of code.

- *Refactoring:* In an ideal world, bad code smells are routinely eliminated through refactoring steps [FBB+99]. As with routine bug fixes, how do we know that refactorings will not break other parts of the system? Unit testing is a well-established practice to help solve this problem, but programming languages do not offer much help for maintaining code and the corresponding tests. Many tools exist, but perhaps more could be offered at the language level?

- *Reengineering:* Here we are concerned with much more radical reworkings of applications, due to architectural drift, or radically new kinds of requirements [DDN02]. Reengineering entails many smaller kinds of refactoring steps, and may intail porting to new environments or migration to a new architecture. Here too, programming languages offer little in the way of managing the correspondence between the old and the new system.

Clearly there is quite a range of issues from the very low-level to the high-level.

- *Implicit architecture:* One of the biggest hurdles in most programming languages, and especially in object-oriented languages, is that the run-time architecture is not apparent from the source code. This makes many kinds of changes difficult to understand, assess and realize.

- *Implicit dependencies:* A slightly different point is that dependencies in general may be hard to identify, understand and assess, for example, because many kinds of dependencies are built up at run-time, and are therefore not apparent in the source code. Not all dependencies are at an architectural level, but any kind of dependency can introduce fragility in the face of change. files which have no connection to the entities that are described.

- *Multiple viewpoints:* Many kinds of change reflect shifting viewpoints of the code. "Glue code" is a symptom of inconsistent viewpoints colliding. Difficulties in realizing certain kinds of change (such "merge and integrate", or porting to a new platform) have to do with reconciling these different viewpoints.

- *Third party software changes:* An additional level of difficulty is introduced by changes to software developed by a third party. Changes between versions can affect client code in unexpected ways, when interfaces or contracts are changed or deprecated.

- *Sealed components:* Worse, it may be impossible to make changes to third party software. When it is possible, we may have difficulty propagating these changes back to the supplier. Alternatively, we may have the additional burden of propagating our changes to future versions of the software.

- *Crosscutting:* Implementing a single new feature or fixing one specific bug often leads to changes that are spread amongst different parts of the program. So all these changes are connected, yet there is no way to record or encode this connection in the program.

- *Component granularity and shape:* When we identify reusable abstractions, we may be frustrated from realizing the needed component in code due to the lack of suitable abstraction mechanisms. Reusable synchronization policies are a well-known example of an abstraction difficult or impossible to implement in mainstream languages [Ach00].

3 Programming Language Support for Change

Existing programming languages offer only limited support for change. Let us briefly review the state of the art.

- *Modularity:* Pretty well every mainstream language offers some mechanism for packaging modules [Par72], however modules tend to have second-class status.
- *Namespaces:* Increasingly we are seeing explicit namespaces being added to mainstream languages like C++, but namespaces are also typically second-class. Piccola is an experimental language with first-class namespaces [AN00].
- *Explicit architecture:* Some recent languages support the specification of explicit component models [AN01, ACN02, MFH01, Zen02].
- *Aspects* provide a way to describe and change crosscutting concerns [KLM+97].
- *Classboxes* offer a packaging mechanism for class extensions with *local rebinding*, which means that extensions are only visible to the parts of an application that directly or indirectly import those extensions [BDNW04].
- *Traits* [SB03] provide composable units of behavior. Common behaviors can be factored into traits and then be used by multiple classes. So traits (like mixins) provide an abstraction that is larger than a method and smaller than a class.
- *OLE versioning:* Microsoft's OLE provides versioned interfaces to components. By using this, the programmer can specify if the interface will stay compatible after a change. This provides a first step into making change explicit, but the OLE versionen components are not integrated into a programing language (e.g. the language can't provide real type-checking based on version information).
- *Perspectives and Layers in PIE:* The PIE System [GB84] was an experiment to merge the features found in frame-based knowlede representation languages with Smalltalk. The system provided a new object-model based on objects with multiple views. PIE was described using itself and thus provided a language that treats its code as objects with multiple views. Single views of objects are combined into layers, which describe one version of the code. PIE provided the programmer with a way of having multiple concurrent versions of the system available and enabled merging these versions. PIE did not allow two layers describing different versions of a system to be active at the same time.

4 Research Directions

We propose to develop a programming language which facilitates experimentation with features to support software change. Since the language should facilitate evolution, it should also support its own evolution. For such a language it is important that parts of the existing system (both language and tools) can be used and adapted for building the new language.

We are building a prototype (based on Squeak [IKM+97]) that will enable multiple, different language kernels to be active at the same time. With this, we want to develop a minimal language kernel. Having such a pluggable kernel will enable experimentation that will allow us to understand what kind of mechanisms for supporting change of the language itself are needed.

Concretely, we plan to experiment with:

- Combining traits and classboxes — we can use classboxes to encapsulate sets of related traits.
- Leveraging traits and classboxes at the level of the language kernel.
- Providing versions and changes as first class objects.
- Providing viewpoint operators — viewpoints extend classboxes with run-time operations. Multiple, inconsistent viewpoints representing different change sets can be simultaneously active. By moving the boundary of a viewpoint, we can gradually extend the scope of changes to different parts of a running system.
- Making reflection pluggable. Start with a minimal non-reflective language kernel and then add reflection and a meta object protocol in a modular way.
- Leveraging this pluggable API for building the developer tools.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004).

References

[Ach00] Franz Achermann. Language support for feature mixing. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367, Malaga, Spain, June 2002. Springer Verlag.

[AN00] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[BDNW04] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. Technical Report IAM-04-003, Institut für Informatik, Universität Bern, Switzerland, June 2004.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[GB84] Ira P. Goldstein and Daniel G. Bobrow. A layered approach to software design. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 387–413. McGraw-Hill, New York, 1984.

[IKM+97] 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*, pages 318–326. ACM Press, November 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[LS80] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison Wesley, 1980.

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson Hsieh. Jiazzi: New age components for old fashioned java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, October 2001.

[Par72] David L. Parnas. A technique for software module specification with examples. *CACM*, 15(5):330–336, May 1972.

[SB03] Nathanael Schärli and Andrew P. Black. A browser for incremental programming. Technical Report CSE-03-008, OGI School of Science & Engineering, Beaverton, Oregon, USA, April 2003.

[Zen02] Matthias Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.