Putting change at the center of the software process⋆

Oscar Nierstrasz

Software Composition Group, University of Bern
www.iam.unibe.ch/∼scg

Introduction

For over thirty years now, software components have been perceived as being essential stepping stones towards flexible and maintainable software systems. But where do the components come from? Once we have the components, how do we put them together? And when we are missing components, how should we synthesize them?

Lehman and Belady established in a classic study that a number of "Laws" of Software Evolution apply to successful software projects [10]. Of these, the two most insightful are perhaps:

– *Continuing change:* A program that is used in a real-world environment *must change*, or become progressively less useful in that environment.
– *Increasing complexity:* As a program evolves, it becomes *more complex*, and extra resources are needed to preserve and simplify its structure.

In this light we can observe that many recent trends in software engineering can actually be seen as *obstacles* to progress, since they offer metaphors that do not help address these issues [11]. "Software Engineering" itself can be seen as a dangerous metaphor that draws too strong an analogy between engineering of hardware and software. Similarly "software maintenance" is clearly a lie when we consider that real maintenance tasks are actually continuous development.

We know that successful software systems are doomed to change. But our programming languages and tools continue to focus on developing static, unchanging models of software. We propose that change should be at the *center* of our software process. To that end, we are exploring programming language mechanisms to support both *fine-grained composition* and *coarse-grained extensibility*, and we are developing tools and techniques to *analyse and facilitate change* in complex systems. In this talk we review problems and limitations with object-oriented and component-based development approaches, and we explore both technological and methodological ways in which change can be better accommodated.

Language support for composition

What programming languages provide specific mechanisms that either take into account or support the fact that programs change over time? It is notable that mainstream programming languages largely emphasize the construction of static software structures, and disregard the

fact that these structures are likely to change. We have been experimenting with various programming languages and language extensions that address certain aspects of change.

Piccola is a small language for composing applications from software components [1,13]. Whereas we have many programming languages that are well-suited for building components, few focus on how components are put together. Piccola provides a notion of *first-class namespaces* that turns out to be immensely useful for expressing, packaging and controlling the ways in which software components are composed [12].

Traits are a fine-grained mechanism for decomposing classes into sets of related methods [15]. Traits overcome a number of difficulties with single and multiple inheritance, while avoiding the fragility inherent in mixins by sidestepping traditional linearization algorithms for composing features. Traits have proven to be extremely useful in refactoring complex class libaries [6].

Classboxes offer a minimal module system for controlling class extensions [4]. Class extensions support *unanticipated change* to third-party classes where subclassing is not an option. In classboxes, as in traits and Piccola, we note that the notion of first-class namespaces is an important means to manage and control change. We conjecture that programming languages that better support change will place more emphasis on such mechanisms.

Mining components

Support for change is clearly not just a language issue. We also need good *tools* to analyze and manage code.

We have been developing a reengineering platform called *Moose* that serves as a code repository and a basis for analyzing software systems [7]. In this context we have developed a series of tools to aid in the understanding and restructuring of complex software systems.

CodeCrawler is a software visualization tool based on the notion of *polymetric views* — simple graphical visualizations of of direct software metrics [8]. One of the most striking applications of polymetrics views is in analyzing the evolution of a software system [9]: an *evolution matrix* quickly reveals which parts of a system are stable or undergoing change. We are further exploring the use of historical data to *predict change* in software systems [14].

We are also exploring ways to mine recurring structures from software systems. *ConAn* is a tool that applies *formal concept analysis* to detect recurring "concepts" in models of software. We have applied this approach to detect implicit contracts in class hierarchies [3] and to detect recurring "software patterns" [2]. We are now exploring ways to assess and improve the quality of the module structure of applications with respect to various reengineering operations.

Where are we? Where do we go?

To conclude, we would do well to note that change is inevitable in software. As a consequence software components, being the *stable* part of software systems, can offer at most half of any equation that would help to improve software productivity.

There is a need for both languages and tools that offer better support to help us cope with and even exploit change.

Nevertheless, we should beware that any new techniques or methods carry some danger with them. Not only do metaphors sometimes blind us, but, as Berry points out [5], any technique that addresses a key difficulty in software development typically entails some painful steps that we will seek to avoid. To achieve any benefit, we must first overcome this pain.

Acknowledgments

References

1. Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.
2. Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Software pattern detection using formal concept analysis. Submitted for publication, March 2004.
3. Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. X-Ray views: Understanding the internals of classes. In *Proceedings of ASE 2003*, pages 267–270. IEEE Computer Society, October 2003. short paper.
4. Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003. Best award paper.
5. Daniel Berry. The inevitable pain of software development: Why there is no silver bullet. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002. preprint.
6. Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA '03, ACM SIGPLAN Notices*, volume 38, pages 47–64, October 2003.
7. Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.
8. Michele Lanza. Program visualization support for highly iterative development environments. In *Proceedings of VisSoft 2003 (International Workshop on Visualizing Software for Understanding and Analysis)*, page To appear. IEEE Press, 2003.
9. Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.
10. Manny M. Lehman and Les Belady. *Program Evolution — Processes of Software Change.* London Academic Press, 1985.
11. Oscar Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002. preprint.
12. Oscar Nierstrasz and Franz Achermann. Separating concerns with first-class namespaces. In Tzilla Elrad, Siobán Clarke, Mehmet Aksit, and Robert Filman, editors, *Aspect-Oriented Software Development.* Addison-Wesley, 2004. To appear.
13. Oscar Nierstrasz, Franz Achermann, and Stefan Kneubühl. A guide to JPiccola. Technical Report IAM-03-003, Institut für Informatik, Universität Bern, Switzerland, June 2003.
14. Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*, 2004.
15. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.