Part 2 : Scientific Information

Main applicant:	Nierstrasz, Oscar
Project title:	Synchronizing Models and Code

Contents

1 Summary of the research plan

Successful software systems are under constant pressure to adapt to changing circumstances. Software adaptations take many forms, are of varying granularity, and may need to take effect over extreme variations in time scale. Running software systems are often subject to fine-grained, short-term adaptation to available resources and run-time context. Modest requirements changes typically provoke medium-grained, medium and long-term evolution of software source code, with consequent short-term adaptation of running software. Deeper requirements changes can provoke coarse-grained, long-term adaptation at the architectural level.

In each of these cases we are faced with the challenge of keeping the source code and the running software synchronized with changes in the higher-level domain and requirements models. This synchronization, however, is often difficult because current languages and runtime systems assume global consistency. They cannot cope with typical inconsistencies of systems with rapidly changing requirements, such as unpredictable variations in the execution environments, inconsistent versions of components, or dispersed code sources.

Our current SNF project, *Bringing Models Closer to Code*, has been exploring various ways to close the conceptual gap between models and the software systems they apply to. In this new project we propose to study novel techniques to keep software systems synchronized with models in the face of varying granularities of change over different time scales.

— *Dynamic meta-objects:* A running system should be able to dynamically respond to changes in its environment (fine-grained and short-term adaptations). Most common approaches to realize run-time adaptation are low-level, fragile, and unsuitable for composition. We propose to address these problems by means of *dynamic meta-objects*. These meta-objects will manipulate high-level representations of an object's behavior, they can be introduced on a per-object basis, and they can be composed to address multiple adaptations at a time.

— *First-class, active contexts:* A software system needs to be able to locally and incrementally update code and program state (medium-grained and medium-term adaptations). Running applications must increasingly cope with interface and data representation mismatches. Instead of placing strict barriers between software components of different versions, we propose to associate versions to *first-class, active contexts*. A running object that enters such a context may need to be dynamically updated to reflect different versions of interfaces, behavior, or even state. Objects may even be in multiple contexts at once, yet still behave in a predictable way.

— *Linked, active source code:* Current software development tools fail to address synchronization of code shared between independent systems (medium-grained and medium-term as well as long-term adaptations). As software evolves, libraries, components and even fragments are frequently duplicated, adapted and specialized across software projects. Instead of treating software source code as passive text, we propose a novel approach in which source code is linked to other source code, to other relevant semantic information, and to high-level models. We plan to draw inspiration not only from sites such as Wikipedia as possible models for linked software, but also from popular social networking sites.

— *Polyglot systems modeling and analyses:* Novel analyses are needed to help software architects assess the impact of changes (coarse-grained and long-term adaptations). These analyses need to account for sub-systems built with diverse technologies and programming languages. We propose to analyze these "polyglot" systems focusing on the technologies involved. To detect hidden architectural dependencies between diverse sub-systems we need to develop a new meta-model that captures and connects the idiosyncrasies of each involved technology. We then plan to research novel metrics-based visualizations to support analyses such as the detection of architectural patterns.

2 Research plan

2.1 Current state of research in the field

Effective support for software evolution entails the construction of detailed models of software at a variety of levels of abstraction. As the literature on software modeling is vast, we briefly summarize some of the most significant developments in recent years related to modeling of software evolution with particular focus on dynamic adaptation, context-dependent adaptation, modeling of change, and mining architectural information.

2.1.1 Dynamic meta-objects

Structural and behavioral reflection are well-known techniques to enable run-time change. Nevertheless, these techniques can be cumbersome and unwieldy in the face of complex and evolving domain knowledge, and unanticipated requirements changes. For this reason many new models of reflection have been developed over the past two decades.

Smalltalk provides a number of reflective features [Duc99], most of these being concerned with structure. Reflective facilities for changing behavior are only supported in a rudimentary way. There is no *meta-object protocol* to allow for fine-grained control of behavior. For example, we cannot re-define what a message send is, and we cannot hook into variable access easily.

The model of reflection offered by a programming language limits and restricts what we can express. If a language does not reify certain abstractions we will not be able to reflect on them. Although it is sometimes possible to add missing abstractions *post hoc*, in general it is not possible to provide a reflection model that can anticipate all potential uses [MW88, DS01, Tan09]. We therefore need an *open* approach to reflection.

Mirrors offer a first attempt to model the reflection problem domain. In this approach objects themselves do not have any reflective capability, but reflection is provided by *mirror objects* [BU04]. Mirrors offer a clear separation of the base level and the meta layer.

There has been some attempts to provide a meta-object model by selectively specifying what should be reified in an application in each particular case. Iguana/J [RC02, RC00] offers a form of selective reification which makes it possible to select program elements down to individual expressions. Other tools like Dalang [WS99], Reflective Java [Wu98], Kava [WS01], the ProActive MOP [CHV01], MetaXa [GK99] and Guaranà [OB99] are targeted specifically at controlling method invocation for Java.

Partial-behavioral reflection [TNCC03] restricts the introduction of reflective capabilities to the parts of a system where they are needed, thus eliminating the cost of reflection when these capabilities are not used. Partial behavioral reflection was conceived using bytecode transformation in Java.

Most reflective approaches have been developed in the context of class-based object-oriented languages. Binding reflective behavior to classes only delivers reflective models that can reflect on the structure and behavior of a class of objects instead of individual instances. To date there exists no thorough analysis of object-specific reflective systems. CLOS [BDG$^+$88] offered some object-specific behavior capabilities, however, they have not been adopted by other languages.

Self [US87] offers object-specific behavior since it is prototype-based rather than class-based.

Each object is built from a prototype with which it shares its behavior and structure. An object can evolve by differentiating itself from its base prototype. The behavior specification is obtained by the use of slots.

Aspect-Oriented Programming (AOP) [KLM+97] addresses the problem of adapting code to accommodate cross-cutting concerns (such as instrumentation, or persistence). Adapted behavior is packaged into an "aspect", which modularizes cross-cutting concerns of the applications. Although aspects can be dynamically enabled or disabled, they are specified statically.

For instance, these reflection models can be used for providing dynamic language reification like CodA [McA95]. CodA reifies the message send process from an operational decomposition point of view of the meta-level.

2.1.2 First-class, active contexts

Software applications increasingly need to adapt their behavior to changing context, and numerous programming constructs have been proposed to address this need. In its most basic form, we have object-oriented programming, in which *dynamic dispatch* allows different code to be evaluated depending on the context of the *receiver* of a message.

The idea of adapting the behavior of objects according to a more general notion of context was pioneered in *subject-oriented programming* [HO93]. The behavior of an object depends on the caller, and the system behaves differently under various perspective. The notion of multi-dimensional method dispatch is central to these approaches where the behavior of an object may depend on several contextual parameters. The *Us* system [SU96] is based on the Self programming language and supports subjective programming. In subjective programming, message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. Each perspective contains an ordered sequence of layers, which determines the currently active behavior. In AOP, join points define the context where a concern will be applied.

The mechanisms described above are all based on a structural notion of scope or context. To provide more flexibility at run-time, temporal mechanisms have also been proposed. Dynamic AOP allows aspects to be enabled or disabled at run-time. Lasagne [TJV02] is a model that supports the context-sensitive selection of aspects, enabling client-specific customization of systems. ContextL [CH05] is an extension of CLOS where co-related modification can be organized into layers that can be activated or deactivated dynamically. The term *context-oriented programming* [HCN08] has emerged to refer to such highly dynamic models where variations are applied selectively based on the changing context.

These approaches have focused mostly on behavioral variations. State adaptations have not been much covered from this perspective, apart from a proposal by Tanter to provide contextual values [Tan08]. There is, however, a large body of work covering the adaptation of *persistent object stores* resulting from schema changes. One of the most comprehensive platforms in this area is Gemstone [BOS91]. The right way to deal with mutable state and persistent entities in programming languages is still an open question, and interesting proposals continue to emerge. Examples include Clojure [Hic08], which takes a novel approach on shared state, or "Declarative object identity using relation types" by Vaziri *et al.* [VTFD07] who propose to introduce the notion of keys in the type system to solve the problem of object equality.

Another important issue is the consistent application of such contextual modification. Various

models were devised to reason about the correctness of object updates, as for instance in the work by Chandrasekhar *et al.* [BLS+03], who use ownership types for this purpose. The concept of transaction was also proposed as to delimit the safe update points [NHFP08].

2.1.3 Linked, active source code

In recent years there has been an increased interest in identifying links between artifacts, primarily between software clones. The conventional attitude towards code duplication and redundancy has been to avoid it [Kos08, FBB+99]. Xie and Engler go so far as to use the term redundant code as a synonym for superfluous code [XE02]. On the other hand, code duplication is ubiquitious [Kos08, Bak95], and has found a kinder judgement more recently as unavoidable [KSNM05] and even beneficial in certain situations [KG06]. Koschke gives an overview [Kos08].

Tools are developed that help cope with the downsides of software clones, without removing them. The tools aid in the following activities, which we will explain.

- detecting clones
- capturing and modeling clones (for further tracking)
- editing clones
- collecting information about clones

Bellon *et al.* [BKA+07] give an overview of how to detect clones. Automatic clone detection serves as a foundation for some tools that aid with software clones, while other tools require the users to manually specify all clones that they wish to be aided with.

Rather than being detected from the source base, clones can be captured at the time of creation in the IDE, and then tracked further. A tool called CloneTracker [ER08] keeps track of clones and notifies users in case any clone updates. However, it assumes that code duplication does not transgress project boundaries. Thus, code snippets found in a code search engine cannot be associated with their origin.

To edit cloned software snippets simultaneously in different places in one software repository, linked editing has been proposed [TBG04], a clone-specific adaptation of synchronous editing [MM01] which edits several places in a software repository at once.

Codebook [BD09] provides a Facebook-like platform to keep track of exchange notifications across different software projects. It is intended to connect developers to the originals of snippets that they cloned. However, Codebook has no automatic mechanism to follow clones as they continue evolving.

A number of tools visualize clones in the IDE to aid reasoning about them. The tool *CSeR* shows the same parts of clones in one color and the diverging parts in another [HJJ09]. Another tool, intensional views, accepts a query expression that defines any measure of structural similarity and then visualizes those parts of the software repository that are deemed similar according to the measure [MKPW06].

Since we plan to store the data on clones in a version control system and then leverage the captured information, we present some related work to leveraging version control systems. Version control systems have become very popular [CW98] to store professionally developed code. Tools

leverage the information stored in software control systems, *e.g.*, Mock *et al.* quantify a programmer's expertise [MH02]. More recent tools try to capture the development process at more fine grained level [RL07], which is used to improve programmer's productivity by improving code completion [RL08].

Git's submodule function allows developers to keep a modified library, and both integrate distant changes and publish local changes back into the project [Cha09].

The *Jazz* development environment integrates bug reports, online discussion and commit messages to a useful whole [HCRP04].

2.1.4 Polyglot systems modeling and analyses

Enterprise applications (EAs) are typically large, distributed, *polyglot* software systems implemented using multiple technologies and programming languages. Numerous architectural patterns for enterprise applications have been identified [Fow05, JPnMK$^+$09, ACM03] in an effort to help developers address common design problems arising in the development of these systems. The selection of suitable architectural patterns in the early phases of development can influence crucial decisions about further design or implementation stages. Therefore it is important to support the identification of those patterns to aid the evolution of the system.

Detection of architectural patterns requires a meta-model that not only captures all structural elements but also conceptual elements (*e.g.*, layers) that constitute the design of the application.

Marinescu has proposed such a meta-model for enterprise applications [MJ06] including a meta-model for relational databases, and she has proposed a technique to enrich the latter with information contained in the source code [Mar07a]. She has also worked on quality assessment of EAs, proposing an approach to detect and remove discrepancies between the database schemas [Mar07b] and the source code and to assess design quality in EAs [Mar06].

Various research teams have worked on architecture recovery and validation using reflexion models [MNS95, MN97]. These models are used to recover architecture by capturing developer knowledge and then mapping this knowledge to the source code [KS03, CKS05]. Intensional views, proposed by Mens [MMW02, MPG03, MKPW06], codify the conceptual structure of software systems. A variety of software maintenance and evolution tasks can then be facilitated by checking conformance of intentional views against the source code. Some tools for architecture recovery and validation are commercially available, such as Sotograph [BKL04]. Sotograph is a software analysis tool that can check architectural conformance, visualize the static structure of the code and assist the user in evaluating the impact of potential modifications on the code.

Due to their complexity and variety, polyglot systems require a sophisticated build system to manage all compilation and deployment dependencies. MAKAO [Ada09, ADTM07] is a reengineering framework that analyses systems built using Makefiles. MAKAO constructs a dependency graph and uses it to support the identification of bad smells.

References

[ACM03] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies 2nd edition.* Prentice Hall, Sun Microsystems Press, June 2003.

[Ada09] Bram Adams. Co-evolution of source code and the build system - impact on the introduction of aosd in legacy systems. In *PhD Symposium at the 25th IEEE International Conference on Software Maintenance (ICSM)*, Edmonton, Canada, September 2009. To appear.

[ADTM07] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In Ladan Tahvildari and Gerardo Canfora, editors, *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*, pages 114–123, Paris, France, October 2007. IEEE Computer Society.

[Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second IEEE Working Conference on Reverse Engineering (WCRE)*, pages 86–95, July 1995.

[BD09] Andrew Begel and Robert DeLine. Codebook: Social networking over code. In *ICSE Companion*, pages 263–266, 2009.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonia E. Keene, Gregor Kiczales, and D.A. Moon. Common Lisp Object System specification, X3J13. Technical Report 88-003, (ANSI COMMON LISP), 1988.

[BKA+07] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[BKL04] Walter Bischofberger, Jan Kühl, and Silvio Löffler. Sotograph – a pragmatic approach to source code architecture conformance checking. In *Software Architecture*, volume 3047 of *LNCS*, pages 1–9. Springer-Verlag, 2004.

[BLS+03] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, October 2005. ACM.

[Cha09] Scott Chacon. *Pro Git.* Apress, xxii, 265 p. edition, August 2009.

[CHV01] Denis Caromel, Fabrice Huet, and Julien Vayssière. A simple security-aware MOP for Java. In *In Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001, volume LNCS 2192*, pages 118–125. Springer-Verlag, 2001.

[CKS05] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Equipping the reflexion method with automated clustering. In *Working Conference on Reverse Engineering (WCRE)*, pages 89–98, 2005.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[DS01] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. *Higher Order Symbol. Comput.*, 14(1):7–34, 2001.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[ER08] Ekwa D. Ekoko and Martin P. Robillard. Clonetracker: tool support for code clone management. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 843–846, New York, NY, USA, 2008. ACM.

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

[Fow05] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2005.

[GK99] Michael Golm and Jürgen Kleinöder. Jumping to the meta level: Behavioral reflection can be fast and flexible. In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 22–39, London, UK, 1999. Springer-Verlag.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.

[HCRP04] Susanne Hupfer, Li T. Cheng, Steven Ross, and John Patterson. Introducing collaboration into an application development environment. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 21–24, New York, NY, USA, 2004. ACM.

[Hic08] Rich Hickey. The Clojure programming language. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, New York, NY, USA, 2008. ACM.

[HJJ09] Daqing Hou, Patricia Jablonski, and Ferosh Jacob. Cnp: Towards an environment for the proactive management of copy-and-paste programming. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 238–242. IEEE, May 2009.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.

[JPnMK+09] Ricardo Jimenez-Peris, Marta Pati no Martinez, Bettina Kemme, Francisco Perez-Sorrosal, and Damian Serrano. *A System of Architectural Patterns for Scalable, Consistent and Highly Available Multi-Tier Service Oriented Infrastructure*, volume 5835 of *LNCS*, pages 1–23. Springer, 2009.

[KG06] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. *WCRE '06*, 0:19–28, 2006.

[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.

[Kos08] Rainer Koschke. Identifying and removing software clones. In *Software Evolution*, chapter 2, pages 15–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[KS03] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, page 36. IEEE Computer Society, 2003.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. An empirical study of code clone genealogies. In *Proceedings of European Software Engineering Conference (ESEC/FSE 2005)*, pages 187–196, New York NY, 2005. ACM Press.

[Mar06] Cristina Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 169–180, Los Alamitos CA, 2006. IEEE Computer Society Press.

[Mar07a] Cristina Marinescu. Discovering the objectual meaning of foreign key constraints in enterprise applications. *Reverse Engineering, Working Conference on*, 0:100–109, 2007.

[Mar07b] Cristina Marinescu. Identification of Relational Discrepancies between Database Schemas and Source-Code in Enterprise Applications. In *Symbolic and Numeric Algorithms for Scientific Computing, 2007. SYNASC. International Symposium on*, pages 93–100, September 2007.

[McA95] Jeff McAffer. Meta-level programming with coda. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

[MH02] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM.

[MJ06] Cristina Marinescu and Ioan Jurca. A meta-model for enterprise applications. In *SYNASC '06: Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 187–194, Washington, DC, USA, 2006. IEEE Computer Society.

[MKPW06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006.

[MM01] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.

[MMW02] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of SEKE 2002*, pages 289–296. ACM Press, 2002.

[MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

[MNS95] Gail Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[MPG03] Kim Mens, Bernard Poll, and Sebastian Gonzalez. Using intentional source-code views to aid software maintenance. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 169–178, September 2003.

[MW88] Daniel P. Friedman M. Wand. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Lisp and Symbolic Computation*, pages 298–307, 1988.

[NHFP08] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *SIGPLAN Not.*, 43(1):37–49, 2008.

[OB99] Alexandre Oliva and Luiz Eduardo Buzato. The design and implementation of Guarana. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 203–216, San Diego, California, USA, May 1999.

[RC00] Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for java. In *Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures*, 2000.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.

[RL07] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, January 2007.

[RL08] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*, pages 317–326, 2008.

[SU96] Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.

[Tan08] Éric Tanter. Contextual values. In *Proceedings of the 4th ACM Dynamic Languages Symposium (DLS 2008)*, Paphos, Cyprus, July 2008. ACM Press. To appear.

[Tan09] Éric Tanter. Reflection and open implementations. Technical Report TR/DCC-2009-13, University of Chile, November 2009.

[TBG04] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.

[TJV02] Eddy Truyen, Wouter Joosen, and Pierre Verbaeten. Consistency management in the presence of simultaneous client-specific views. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 501, Washington, DC, USA, 2002. IEEE Computer Society.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

[VTFD07] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *ECOOP*, pages 54–78, 2007.

[WS99] Ian Welch and Robert Stroud. Dalang – a reflective Java extension, 1999. In Proceedings of the OOPSLA 99 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, Oct.

[WS01] Ian Welch and Robert J. Stroud. Kava — using bytecode rewriting to add behavioural reflection to Java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology (COOTS'2001)*, pages 119–130, San Antonio, Texas, USA, February 2001.

[Wu98] Zhixue Wu. Reflective Java and a reflective component-based transaction architecture, 1998. In Proceedings of the ACM OOPSLA 98 Workshop on Reflective Programming in Java and C++, Oct. 1998.

[XE02] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 51–60. ACM Press, 2002.

2.2 Current state of own research

The Software Composition Group has extensive experience with reflection in programming languages, software (meta-)modeling, static and dynamic analysis of complex software systems, and reverse and re-engineering. We briefly summarize some of the more significant research results and publications which form the background to this proposal. In particular, we point to the paper *Model-centric, Context-aware Software Adaptation* [NDR09] (included with this proposal), which summarizes the research in our current SNF project, *Bringing Models Closer to Code*[1], and anticipates the research proposed here.

All papers are accessible online from: `http://scg.unibe.ch/publications/`.

2.2.1 Dynamic meta-objects

Piccola is a simple, formally specified composition language [AN05] for building applications from components. Piccola makes use of reflection to build up component models at run-time. Since Piccola provides a bridge to Java, it makes use of dynamically generated adaptors to communicate between Java objects and Piccola components [NAK03].

ByteSurgeon [DDT06] is a framework for transforming and annotating Smalltalk bytecode at runtime. Geppetto [Röt06] is a dynamic runtime meta object protocol for behavioral reflection that has been built on top of ByteSurgeon. Geppetto allows for a very fine-grained control of behavioral reflection. Since bytecode adaptation has several limitations, tools which are closer to the source code were developed. Persephone [DDLM07] introduced an approach to sub-method reflection using a model of methods based on abstract syntax trees (ASTs). Persephone works by annotating the AST nodes with metadata that can later be used for modifying execution. One application of Persephone's sub-method reflection was TypePlug [HDN09], a pluggable type system for Smalltalk.

[1] `http://scg.unibe.ch/research/snf08`

ByteSurgeon, Geppetto and Persephone were used to develop Reflectivity [Den08], which provides unanticipated partial behavioral and structural reflection at the sub-method level, using ASTs rather than source code or bytecode as underlying model of the software.

Dynamic Synchronization [RN09] provides first-class synchronization specifications which express safety requirements, and a mechanism built on top of Reflectivity which dynamically adapts objects to different runtime situations.

Pinocchio[2] is an open language system bootstrapped from a fully reflective runtime. A key goal of Pinocchio is to eliminate the barrier between the virtual machine and the run-time system of a programming language. Pinocchio is still in an early stage of development.

Although none of the previous work listed above fully addresses the problem of dynamic adaptation of individual objects, sub-method partial behavioral reflection offers the finest degree of control. For this reason we are proposing to extend this approach with dynamic meta-objects to run-time object adaptation.

2.2.2 First-class, active contexts

The semantics of Piccola (see above) is based on the notion of a *form*, which serves not only as an extensible record (*i.e.*, a primitive form of object), but also as a kind of first-class dynamic namespace [AN00]. This simple mechanism turned out to be extremely useful for constructing multiple contexts for code to execute. By controlling the visibility of services within a namespace, one could, for example, define a sandbox for running untrusted code, or provide special namespaces with limited or extended resources. At the core, Piccola is purely functional, so it is not possible for components to move from one context to another.

Classboxes [BDNW05] addressed the problem of extending existing software in a controlled way. Classboxes form a module system in which imported classes may be extended with purely local adaptations. Classboxes provide so-called local rebinding of method changes. Objects of the same class instantiated from other classboxes will not see those extensions. Classboxes were first used for structural scoping of the extensions, but could easily be extended to provide dynamic adaption of the behavior of objects upon a particular context.

Changeboxes [DGL$^+$07] are loosely inspired by classboxes, but they address several different issues within a consistent model: (i) they capture the semantics of changes as they happen, (ii) they limit the scope of changes to a particular changebox and (iii) they are activated and deactivate dynamically on the context. Unlike classboxes, which focus on class extensions, changeboxes treat changes as first class entities. Although changeboxes can be activated dynamically, objects cannot migrate from one changebox to another. Context-dependent adaptations are therefore not supported.

Object Flow Analysis [LGN08] uses first-class aliases to keep track of of the flow of objects during execution. As a side effect, the complete historical context of objects is maintained, thus enabling back-in time debugging [LFN09].

An initial attempt has been made to develop a formal model of evolving objects that adapt to changing context [DCGN08].

[2] `http://scg.unibe.ch/research/pinocchio`

2.2.3 Linked, active source code

Clone detection has been intensively studied by the Software Composition Group. Research at the SCG has compared techniques for clone detection [DNR06, RDL04], and found a lightweight method that is fast and easy to implement and provides state of the art accuracy in detecting clones.

Hermion [RGN08] and Senseo [RHV+09] augment the traditional static source code view provided by an Integrated Development Environment (IDE) with dynamic information in an attempt to support development and maintenance tasks that can benefit from such precise information.

Moose [NDG05] is a platform for software reverse and re-engineering based on an extensible, language-independent meta-model. It provides basic infrastructure for modeling existing software projects, querying models, collecting software metrics, and generating visualizations of models. Moose has been used as the basis for a large range of research projects in program comprehension resulting in numerous publications[3].

2.2.4 Polyglot systems modeling and analyses

Moose has been used to develop various techniques to recover high-level information implicit in complex software systems. A technique for iteratively recovering implicit information concerning roles and collaborations was developed, exploiting the correlation between static and dynamic information [RD99, RD02]. In another project, formal concept analysis was used to recover collaboration patterns from static source code [ABN04]. Yet another project addressed the problem of understanding how features evolve in a software system by recovering the relationship between features and software components [GDG06].

Software metrics and visualization play an important role in reducing the large amount of data in a complex software system and extracting useful information from it. Polymetric views [LD03] offer a lightweight technique to map various direct metrics to simple visualizations that provide compact, high-level views of a software system. Mondrian [MGL06] is a component-based visualization framework that allows polymetric views to be interactively scripted using an internal domain specific language.

Tool-building is an important but often underrated aspect of software engineering. Glamour [BGR+09] is a component-based framework for building model browsers. Like Mondrian, Glamour offers an internal domain specific language for scripting browsers. In this way, a dedicated browser to support specialized development or analysis tasks can be quickly and cheaply produced.

We have already spent initial effort to analyze technical problems in polyglot systems [Per09], such as the identification of inconsistent transaction scopes inside Java Enterprise Applications (JEAs). In this work we identified that the FAMIX Meta-Model [TDDN00] is not expressive enough to capture the idiosyncrasies of JEAs. Therefore, to perform high-level analyses of polyglot systems, we need a more expressive meta-model.

[3] `http://www.moosetechnology.org/publications/list`

References

[ABN04] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, November 2004.

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

[AN05] Franz Achermann and Oscar Nierstrasz. A calculus for reasoning about software components. *Theoretical Computer Science*, 331(2-3):367–396, 2005.

[BDNW05] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.

[BGR+09] Philipp Bunge, Tudor Gîrba, Lukas Renggli, Jorge Ressia, and David Röthlisberger. Scripting browsers with Glamour. European Smalltalk User Group 2009 Technology Innovation Awards, August 2009. Glamour was awarded the 3rd prize.

[DCGN08] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. A calculus of evolving objects. *Scientific Annals of Computer Science*, XVIII:63–98, 2008.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[DGL+07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.

[DNR06] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(1):37–58, January 2006.

[GDG06] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006.

[HDN09] Niklaus Haldimann, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Journal of Computer Languages, Systems and Structures*, 35(1):48–64, April 2009.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[LFN09] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. Flow-centric, back-in-time debugging. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009*, volume 33 of *LNBIP*, pages 272–288. Springer-Verlag, 2009.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[NAK03] Oscar Nierstrasz, Franz Achermann, and Stefan Kneubühl. A guide to JPiccola. Technical Report IAM-03-003, Institut für Informatik, Universität Bern, Switzerland, June 2003.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[NDR09] Oscar Nierstrasz, Marcus Denker, and Lukas Renggli. Model-centric, context-aware software adaptation. In Betty H.C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 128–145. Springer-Verlag, 2009.

[Per09] Fabrizio Perin. Enabling the evolution of J2EE applications through reverse engineering and quality assurance. In *Proceedings of the PhD Symposium at the Working Conference on Reverse Engineering (WCRE 2009)*, pages 291–294. IEEE Computer Society Press, October 2009.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.

[RD02] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02)*, page 34, Los Alamitos CA, October 2002. IEEE Computer Society.

[RDL04] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 100–109. IEEE Computer Society Press, November 2004.

[RGN08] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[RHV+09] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Augmenting static source views in IDEs with dynamic metrics. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 253–262, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[RN09] Jorge Ressia and Oscar Nierstrasz. Dynamic synchronization — a synchronization model through behavioral reflection. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2009)*. ACM Digital Library, 2009. To appear.

[Röt06] David Röthlisberger. Geppetto: Enhancing Smalltalk's reflective capabilities with unanticipated reflection. Master's thesis, University of Bern, January 2006.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167. IEEE Computer Society Press, 2000.

2.3 Detailed Research Plan

This project addresses synchronization of source code and running software with changing requirements at various granularities and time scales.

- *Dynamic meta-objects* will support fine-grained behavioral adaptation of individual objects or groups of objects
- *First-class, active contexts* will enable dynamic adaptation of running applications to multiple software versions
- *Linked, active source code* will support medium and long-term synchronization of multiple versions of evolving source code

- *Polyglot systems modeling and analyses* will target coarse-grained, long-term evolution of complex, polyglot systems

2.3.1 Dynamic meta-objects

Applications today are subject to many forms of dynamic adaptation, but these are usually restricted to limited policies that are fixed in advance. Consider the classical cases of concurrency and persistence. Multithreaded applications need to synchronize their use of resources in order to avoid data races and other safety and liveness issues. Synchronization code is typically interspersed with functional code, thereby impacting understandability and maintainability of the code base. The synchronization policy is fixed, and cannot be dynamically enabled or disabled when needed. Transactional behavior for persistence is similar. The transactional mechanisms to be used as well as the policy in place (*e.g.*, pessimistic or optimistic) is hard-wired in the source code, and cannot be adapted at run-time depending on the current platform or the persistence infrastructure available at different sites in a distributed environment.

Existing approaches to support run-time adaptation are generally low-level, fragile with respect to source code evolution, and are not designed to enable composition of multiple adaptations. Bytecode manipulation, for example, works at a much lower level of abstraction than source code. With AST annotation approaches we make the semantic gap smaller but we only rely on a *link* abstraction which will delegate responsibility for intervention to a meta-object. Both of these approaches are difficult to use and at a low level, since we either have to know how and where to manipulate the bytecode, or we have to explicitly add links to the AST representation. Some approaches like Iguana provide an meta-object abstraction on top of bytecode manipulation, but they limit which reifications you can reflect on.

We propose to develop an approach to run-time adaptation based on dynamic meta-objects. Dynamic meta-objects will not be restricted to reifications offered by the host language, but will be able to reify arbitrary domain abstractions. Furthermore, they can be assigned to individual objects or groups of objects, not just classes.

Meta-object framework. We plan to extend the behavioral reflection framework offered by Reflectivity with dynamic meta-objects. Each meta-object will maintain a set of links that adapt the base object behavior and structure. The framework will also extend the meta-object model of Iguana by allowing meta-objects to be composed of other meta-objects. For example, a meta-object to provide CodA-like message sends will be composed of an existing message send reification meta-object. To provide dynamically synchronized access to objects we will use the meta-object that reifies the instance variable accesses and then synchronizes the process of accessing the instance variable.

Case studies. We will use the domain of synchronization as a case study for validating dynamic meta-objects. Synchronization policies can be modeled as meta-objects shared by the concerned objects. Meta-objects can abstract from the functional details of the base-level objects and encapsulate the technical details of a particular synchronization policy. Given a change in the the runtime environment, the meta-objects may be dynamically enabled or disabled. For example, if the concerned objects are in a single-threaded environment, synchronization may be safely switched off. We also plan to explore other suitable case studies to assess the

composability of dynamic meta-objects for concurrency with others for, say, transactional behavior, authorization, or localization.

Object-specific behavior. Reflection facilities need to be able to specify behavior and structure for a single object as well as for a group of objects. Instead of binding meta-objects strictly to classes, as is common in most reflective approaches, we propose to introduce meta-objects that can adapt the structure and behavior of an arbitrary group of objects, even if they do not belong to the same class. Furthermore, meta-objects should be dynamically composable to address dynamic run-time changes of, say, policies or service availability.

Language design. In the long term, we plan to explore the design and implementation of a core language in which the state and behavior is no longer defined by a class but by dynamic meta-objects. Such a language model would be maximally dynamic, leaving an object to be nothing but an identifier controlled by multiple meta-objects. Classes could be modeled by a specific meta-object that defines the structure and behavior of its instances. Although we plan to use Smalltalk as the platform for modeling, implementing and validating dynamic meta-objects, Smalltalk's virtual machine inherently assumes a class-based language model. We therefore plan to use Pinocchio as a platform for experimenting with the language design, since Pinocchio offers a fully bootstrapped open language system in which there exists no separation between the virtual machine and the runtime system.

2.3.2 First-class, active contexts

Whereas dynamic meta-objects enable fine-grained object adaptation, they do not offer a sufficient mechanism to synchronize modules or whole programs with changing contexts and requirements. Similarly, existing context-oriented approaches offer mechanisms (based on layers) to adapt code to contexts, but they do not model context directly. The notion of a declarative context switch has scarcely been investigated. Instead, context is typically hard-coded in rules that sample and test arbitrary environmental attributes.

Existing context-oriented approaches are also limited to certain types of behavioral variations. Modification of *state* is either missing (as in Classboxes and Changeboxes) or is supported only to a limited extent (as in ContextL).

We plan to develop a comprehensive context-oriented platform based on *first-class contexts* that play an active role in triggering changes in state and behavior of concerned objects. This work will build on the infrastructure offered by dynamic meta-objects, especially during the first year where these two tracks will be developed in close cooperation. The main goal of this research is to elaborate a comprehensive set of abstractions to express and reason about contextual modifications, in particular context switching. The impact of context switching on existing objects will be studied, and this should in particular bring state modification to the area of possible contextual modifications. A particularly compelling use case is to dynamically adapt the state and behavior of objects to contexts in which the software versions change. Not only the object, but its clients and providers may behave differently in a heterogeneous, distributed, and evolving system. Objects undergoing context changes should nevertheless behave predictably.

The research will consist of the following tracks:

Model. We will develop a model where the behavior, interface, and state of an object is *contextual*. We will also design a declarative language to express context switches, inspired partly by AOP point-cut languages. Context switches will occur at certain points in the execution, resulting in object behavior to vary over time. In case of a context switch, an object may be adapted lazily using context-dependent adaptation logic. The model will focus on objects instead of classes. The identical object may be seen as an instance of a different class (or different class version) depending on the context.

Semantics. The concrete model will be formalized with operational semantics to reason about its soundness. A first goal is to prove that the model preserves the semantics of existing programs. We then plan to prove two important properties: (i) existing thread-safe program remain thread-safe in our model and (ii) the behavior of an object is always predictable. A second goal is to prove that custom adaptation logic (which can be user-provided) can be safely composed. Programs making explicit usage of contexts will be highly dynamic. Therefore, even if the model is sound, a program may still contain subtle bugs (in a similar way that a lock acquired and never released is likely a bug). A third goal is then to carry out static or dynamic analysis of contextual programs in order to increase the confidence in the program correctness. We plan in particular to extend dynamic type inference to dynamic *context* inference and be able to identify which sections of the code run in which contexts.

Run-time support. We expect the execution and memory overhead of contextual method dispatch and contextual state to be large unless implemented efficiently in the VM. Memory management will be addressed first. We will devise an efficient memory management scheme to reduce the memory footprint, for instance by implementing a "copy-on-write" strategy. We expect most objects to "die young" (as assumed by generational garbage collection), which would imply little or no overhead for such objects. The memory usage of long-lived objects in our model will need to be studied first to devise further optimizations. In a second step we will implement an efficient multi-dimensional dispatch to support active contexts with low performance overhead.

Case studies. We will assess the applicability of our model to solve well-known problems at different levels. At the code level, we plan to show how selected idioms and design patterns, such as Memento and Adaptor, can be re-implemented with contexts. At the module level, we plan to study and assess which incompatibilities between modules can be accommodated with active contexts. At the system level, we will show that context also provide a foundation for incremental dynamic software update of whole programs. We will evaluate these case studies according to two criteria, namely *practicability* and *performance*.

2.3.3 Linked, active source code

As complex software systems evolve and similar but distinct versions of software components proliferate, it becomes harder to keep track of the origins of software artifacts and how changes may impact their descendants.

We plan to investigate how collaboration takes place and how it can be aided across project boundaries. As a platform for experiments we plan to implement a prototype of a scheme to

initially create and then maintain links between clones of software snippets. In this prototype, a code search engine will assist the developer by integrating its results into the source code. The IDE then remembers the origin of the code snippet and informs the repository that a clone was created, thus creating a link between original and copy. We will refer to clones created in this way as *hot clones*. Whenever a hot clone changes, the linked clones' developers are informed and offered the option to update their instance. Also, whenever a method is inspected, its clones will be inspectable, too, thus providing valuable information for developers. The connections between clone instances are therefore proactive and bidirectional.

Specifically, we plan to follow the following research tracks:

Hot clone model. We will develop a model of hot clones and a prototype implementing the model. We plan to increase the value of locally fixing bugs by extending the IDE so that the provider of a library is informed when local changes occur. This should increase the chance of the revision being incorporated into the supplier's version. We plan to use the same approach to establish links between software clones across project boundaries, thus making code duplication visible, and trackable. Thus, when code is changed by supplier or consumer, the other party is informed of the change and asked whether they wish to integrate the change or not.

Publishability. In our view, the tight integration between supplier and consumer of code snippets will benefit both parties. It may then become interesting to allow suppliers to take special care that components of their application can be found and copied into other applications. We plan to investigate how suppliers can mark individual components of their applications as copyable, and integrate that information with code search engines. If the supplier and consumer of a code snippet use different coding conventions, integration may pose difficulties in ensuring that the respective local conventions are followed. We plan to investigate how the integration of updates in distant clones can be aided programatically.

Clone usage analysis. Establishing a link between consumer and supplier of a code snippet will create data on code duplication that will give us greater insight into how developers use code duplication. Conversely, we plan to investigate whether knowledge of the cloned usage of their own software snippets may aid developers understand possible flaws in their own design, because clones may be modified to circumvent possibly needless limitations of the original code.

Linking other software artifacts. Once we're able to link software clones, we should also link other kinds of software artifacts. Discussions of a code snippets will be linked with these code snippets, bug tracker entries that refer to specific software artifacts should be linked to these software artifacts. Even design sketches and drawings of a user interface could be linked to the code that generates that user interface. Thus, browsing the source code should reveal more information than it classically does.

Trustability. We plan to build a central repository of example domain objects and methods. Going beyond cloned code snippets, entire cloned libraries will be integrated. Developers can mark fragments of their projects as shareable and then these fragments can serve as

examples of common domain objects. We plan to allow developers that only copied the example domain objects to easily modify, discuss them and set a level of trustability. We plan to investigate how the trustability of examples can be verified and maintained despite easy accessibility. A promising path is to draw inspiration from Wikipedia and Web 2.0 by letting the community verbally critique and modify (as in Wikipedia) or vote on (as in many Web 2.0 applications) these provided examples. We plan to extend the code search engine to run as the developer is typing and suggest implementations of classes and methods from the examples repository.

Each of the research tracks we plan to assess by qualitative user studies, where we monitor the usage of our prototype and assess whether our tools suit the developer's mental model and whether it aids the engineering process.

2.3.4 Polyglot systems modeling and analyses

Large software systems such as enterprise applications are generally built using multiple technologies and languages. The polyglot nature of these systems makes it difficult to obtain an overview that accurately captures dependencies between the various parts, and consequently to evaluate their evolution and assess the impact of changes.

In order to enable such analyses we need a unified description of polyglot systems that incorporates not only structural elements but also higher-level, architectural concepts. We propose to model such systems from the perspective of the technologies used to build them. We plan also to develop new types of analyses in terms of software visualizations, metrics, and architectural pattern detection, and to validate the meta-model through real-life case studies.

Consider, for example, Java Enterprise Applications (JEAs) composed of Java source code, various kinds of Enterprise Java Beans (EJBs), XML configuration files, SQL, and various subsystems to support persistence, transactions and physical distribution. In such a system it can be hard to manually determine the scope of existing transactions. As a consequence, changes to code that accesses the database layer may incorrectly impose redundant or lacking transactions. Similar difficulties may arise when attempting to reason about other non-functional aspects that may be specified in a distributed or heterogeneous fashion, such as security and access rights, reliability, or testability.

The previous example illustrates the difficulties that developers face when trying to understand dependencies in heterogenous systems like JEAs, for instance to fix a bug or introduce a new feature. We plan to develop a unified model of polyglot systems to be able to automatically detect previously hidden dependencies. By raising the level of abstraction our analysis will help software developers and architects to detect inconsistencies and support the navigation of relationships between components.

The research will consist of the following tracks.

Polyglot systems modeling. We will develop a technology-centric approach to modeling polyglot systems. We will research ways to connect the technology-specific aspects by representing them in a unified model. We plan to integrate the new meta-model into Moose to be able to use our existing infrastructure. To drive the modeling effort, we will identify potential case studies based on open-source and industrial software, and establish modeling

requirements for querying, navigation and analysis. We plan to seek out partners who can provide objective analysis requirements for actual applications in use. (We are currently collaborating with various academic and industrial partners who would be candidates for this role.). We plan to validate the effectiveness of our approach using the technical issues provided by our partners as case studies.

Information extraction. Based on the identified case studies and analysis requirements, we will develop the needed static and dynamic techniques to extract and synchronize models from the heterogeneous sources of polyglot systems. We plan to collect information from sources such as Java (DTOs, JSPs, EJBs, Distributed Objects etc.), Smalltalk, SQL, and XML configurations files of different applications like Hibernate, Toplink, Maven, and Ant.

Analyses. Initially we plan to focus on the analysis of the persistency system and its relations to the other components. To extract dependencies between system components we plan to also investigate the build system. In a second step we will focus on the identification of architectural patterns and violations of architectural constraints, and on change impact analysis. This research will entail the development of dedicated metrics, queries, and visualizations. We will develop experimental prototypes supporting the analyses on top of Moose, using Mondrian and Glamour.

Case studies. We will experimentally validate our approach using the open source and industrial case studies identified earlier. In controlled experiments we want to evaluate how effective our approach is to solve the concrete problems that our industrial partners have reported. Furthermore, based on large open-source systems, we want to evaluate how well our approach supports developers understanding how changes in one part of the system may affect other components.

2.4 Schedule and milestones

Year 1	
Dynamic meta-objects	– Meta-object framework: extend Reflectivity with dynamic meta-objects – Case studies: analyze existing approaches to dynamic adaptation, and explore application of dynamic meta-objects to overcome limitations
First-class, active contexts	– Model: develop a model of active contexts and implement a prototype in Smalltalk – Semantics: formalize operational semantics of active contexts
Linked, active source code	– Hot clone model: Develop a model of linked software clones. Implement a prototype that keeps track of clones introduced by copying from software searches – Publishability: Extend the prototype to allow sharing of common business domain objects and procedures. Carry out case study to establish how people clone, share, and exchange code snippets
Polyglot systems modeling and analyses	– Polyglot systems modeling: Develop a meta-model to capture technological aspects of polyglot software systems, based on identified case studies – Information extraction: develop static and dynamic techniques to extract model data from polyglot software
Year 2	
Dynamic meta-objects	– Object-specific behavior: explore application of dynamic meta-objects to adapt individual objects and groups of objects. – Language design: explore dynamic meta-objects as a foundation for a reflective language without built-in classes
First-class, active contexts	– Run-time support: investigate time and space-efficient sharing between contexts and efficient context-dependent dispatch – Case studies: develop case studies to assess performance and practicability of active contexts
Linked, active source code	– Clone usage analysis: support analysis of clone usage based on gathered code duplication data – Linking other software artifacts: the model and approach will be extended to accommodate other artifacts – Trustability: extend hot clone model to incorporate means to assess trust. Implement IDE tools that propose snippets as one is editing.
Polyglot systems modeling and analyses	– Analyses: develop analysis techniques, metrics and visualizations for polyglot systems – Case studies: carry out empirical studies on open source and industrial systems previously identified

2.5 Importance and impact

Software continues to become more complex and heterogeneous, while pressure is increasing for software to be adapted to changing conditions, both in the long term and in the short term. Despite this pressure, few novel mechanisms have been developed to support dynamic and disciplined adaptation of software to continuously changing requirements. This project aims to make several advances of both academic and industrial interest.

The research proposed here is foundational, and will be mainly disseminated through scientific venues (*i.e.*, high-impact, peer-reviewed journals and conferences accepting full papers).

The first two tracks focus on aspects of programming language design and run-time support, and are primarily of interest to language designers and researchers. The second two tracks target more directly support for developers, and promise to produce not only results of academic interest, but also prototypes of tools that may influence future development environments.

We are already collaborating informally with partners in the Bern area on analysis of industrial software, and hope to intensify these partnerships in the context of the proposed project.

We are also actively involved in the development of *Pharo*[4], an open-source Smalltalk development environment which aims to offer a clean and stable platform for both academic research into dynamic languages and professional development with Smalltalk.

Moose is an open-source reengineering platform original developed at the University of Bern in the context of several EU and SNF projects. We are continuing to develop Moose together with academic and industrial partners[5]. The research carried out in this project will influence the development of Moose, and promote its further dissemination.

[4] `http://www.pharo-project.org`
[5] `http://www.moosetechnology.org`