Part 2 : Scientific Information

Main applicant:	Nierstrasz, Oscar
Project title:	Bringing Models Closer to Code

Contents

1 Summary

Software models are notoriously disconnected from source code. At the start of a software development project, the emphasis is on eliciting system requirements and defining domain and architectural models. Once development commences, the original models become disconnected from the code that they describe. It becomes increasingly difficult to reestablish traceability links as crucial knowledge of design decisions and user features are implicit in the code.

This situation poses difficulties for developers who need to understand existing code before adding new features or making changes. Furthermore, developers need to communicate with the domain experts and stakeholders using a representation of the system that both parties can understand. The problem is particularly critical for today's software applications which are constantly evolving and being adapted to new requirements, platforms or services.

We propose to investigate four related research tracks that seek to bring models and code closer together:

1. Coordinating models and code. Many techniques exist to recover models from various software artifacts, but little has been done to capture implicit human knowledge and to make it explicit in the code. Similarly, little effort has been invested in coordinating different models of a software system and detecting patterns implicit in their interconnections. In this track we will (i) explore ways to embed multiple models in code (*e.g.*, by means of annotations) and navigate between these multiple views, and (ii) apply various clustering techniques to mine higher-level abstractions from code and models. The results of this work will yield useful techniques that are directly applicable in the context of the subsequent three research tracks.

2. Embedding domain models in the code. A well-designed and maintainable software system can be viewed as a set of configurations of software components at various levels of abstraction, each capturing different kinds of domain models. To make domain models explicit in the code, we need to raise them to the level of Domain Specific Languages (DSLs) within the host programming language. We plan to (i) develop an adaptive model for a host language to accomodate multiple DSLs, (ii) validate the model by showing how DSL-aware development tools (browser, debuggers *etc.*) can then both support and exploit domain models to raise the developer's level of abstraction, and (iii) expose implicit domain models by refactoring the APIs towards domain specific languages (DSLs).

3. Bringing dynamic models to the IDE. IDEs typically exploit the static software structure while ignoring run-time structures. In this track we plan to (i) explore how the results of run-time analyses can be brought to the IDE to provide a more effective and comprehensive development environment. Run-time information, usage scenarios, and features are examples of the types of information typically missing from an IDE. We will also (ii) carry out empirical studies to validate the effectiveness of run-time analysis techniques for supporting typical development and maintenance tasks.

4. Model-centric development. As a direct consequence of the preceding three research tracks, we believe development of new software should benefit from the interplay between the various models that are produced on the way to producing the software itself. Unlike model-driven and round-trip engineering approaches where models and code are seen as separate artifacts, we propose to develop an experimental programming environment in which *executable models* themselves are the primary artifact produced. Software developed in such an environment should be self-describing and self-aware — that is, all models are interconnected and are always available to both development and run-time environments. Traceability between models (*i.e.*, from requirements to design decisions) should be explicit in the software itself. As such a system will be self-describing, it will be model-driven at run-time, and can, for example, generate its own user interface and adapt it automatically.

2 Research plan

2.1 State of Research in the Field

The role of models in the software development process has attracted enormous interest in both research and industry in recent years. We briefly survey some of the most important developments relevant to this proposal, and some limitations of current approaches.

1. Coordinating models and code. Over the past decade substantial research effort has been invested in developing techniques for reverse engineering higher level information from software systems. However few researchers have attempted to correlate multiple models of software. Even less have explored the value of capturing and making human knowledge explicit, so as to reveal further implicit patterns in multiple views of a software system. With this premise in mind we review the literature most relevant to this topic that describes techniques for extracting high level models of software and those that rely on human knowledge.

Techniques based on Formal Concept Analysis (FCA) has been applied by various researchers to mine abstractions from code. Examples of these approaches detect structural design patterns in the relationships between classes [TA99], to maintain, understand and detect inconsistencies in the class hierarchies, [GMM$^+$98, ST98, HDL00], and to select an effective order for reading methods and reveal how the attributes of classes are used in methods [Dek03].

A variety of techniques focus detecting of crosscutting concerns in a program. Examples of these include studying which parts of the program change at the same time [BZ06], applying code clone detection techniques [BDET05], identifying commonalities in dynamic traces [BK04]. Marin *et al.* presents a technique for identifying aspects by using a fan-in analysis [MDM07].

Feature identification is a technique to identify which parts of the source code are activated when exercising a feature [WS95]. Eisenbarth *et al.* uses a semi-automatic technique based on a combination of static analysis, dynamic analysis and formal concept analysis to detect features in source code [EKS03]. The authors emphasize the need for the human input to refine the findings. A complementary approach identifies features at a fine-grained level of statements rather than methods [KQ05].

Čubranić *et al.* proposes a "group memory" in the form of a searchable database with artifacts related to a software system [CM03]. Their approach defines a structured meta-model, which relates bug reports, news messages, external documentation and source files. The goal of this approach is to exploit the information in the model so as to propose the reverse engineer the most relevant artifact for a specific task.

Latent Semantic Indexing (LSI) is an information retrieval technique to locate linguistic topics in a set of documents. Maletic and Marcus pioneered its use in the context of software analysis when they categorized the source code files of the Mosaic web browser [MM00]. LSI is employed in several similar analyses: to detect high-level conceptual clones [MM01], to recover links between external documentation and source code [MM03], to detect concepts in the code [MSRM04] and to compute the cohesion of a class based on the semantic similarity of its methods [MP05].

While LSI is a powerful technique, it is still of limited use as the concepts it identifies lack structure. Raţiu and Deissenboeck address this as their approach seeks to bridge the gap between existing ontologies and source code [RD06b]. Their methodology is iterative and requires human intervention to grow both the ontology and the mapping. Several analyses are built on top of this approach like: detection of semantic defects such as conceptual duplications [RD06a] or mismatches between described concepts and their implementation [RJ07].

Reflexion models are used for architecture recovery [MNS95, KS03]. With this approach, developer knowledge is encoded in a model and then manually mapped to the source code. Recently, this approach was complemented with automated clustering to infer candidate components detected as belonging together [CKS05]. The reverse engineer can then take these candidates as input and manually change them to fit the mental model.

Pinzger *et al.* proposes a lightweight approach to recover architectural elements by applying a pattern matcher [PFGJ02]. With their approach, an expert of a system under study is required to encode the rules of interest in queries. Another query-based approach is proposed by Mens *et al.* to recover links between source code and external information like architectural constraints [MKPW06]. They refer to the queries as *Intensional Views* and they are expressed in a Prolog-like engine by the expert.

2. Embedding domain models in the code. Unlike general-purpose programming languages, domain specific languages (DSLs) tend to be compact languages that provide appropriate notations and abstractions for a particular problem domain. It was shown that DSLs increase productivity and maintainability for specialized tasks [DK97]. DSLs are often categorized as being either homogenous (internal), where the host-language and the DSL are one and the same, or heterogeneous (external), where the two languages are distinct [She01]. Techniques are proposed to define language and semantics for new DSLs [KRV07]. The idea of designing languages that embrace adding new DSLs has been a focus of research in the past [Ode07, WP07, Tra08]. However integrating those languages into existing tools has been largely neglected.

Muller *et al.* [MSFB05] present a specific metamodel and associated DSL for the modeling of dynamic web specific concerns. Web applications are represented as three related models (business, hypertext and presentation). To specify constraints and behavior on the model, an action language (based on OCL and Java) is used. Xactium [CESW04] and Kermeta [MFJ05] are executable meta-languages that define new languages similar to work with EMOF [Gro04]. Both provide a dedicated language for specifying meta-level operations, however they lack appropriate tool support.

In the 1990s there was considerable interest in the development of architectural description languages (ADLs) [SG96] to capture and express architectural knowledge of a software system. ADLs can be viewed as DSLs for describing the architecture of complex software systems. Many DSLs formalize architecture in terms of components, connectors, and the rules governing their composition [SG96]. This idea is also implicitly contained in the notion of *scripting languages*, which can be seen as DSLs for composing applications from components written in another, usually lower-level programming language [Ous98]. Despite this, the interplay between conventional object-oriented languages, ADLs, scripting languages and DSLs has not yet been thoroughly studied nor has it been exploited in practice.

3. Bringing dynamic models to the IDE. There is a large body of research in the area of dynamic analysis as well as in the area of development environments, but only a few publications connect these two topics.

Dynamic analyses based on tracing mechanisms typically focus on capturing a call tree of message sends. Many of these approaches do not focus on bridging the gap between dynamic behavior and the static structure of a program [DRW00, HLBAL05, WH92]. IDEs traditionally deal purely with static source artifacts. To achieve a seamless integration of dynamic information into an IDE, we need to embed dynamic models of software in the static perspective of a program. A tight integration of dynamic models in a developer's environment renders them useful for more efficient navigation of a software system or for gaining an understanding of how objects of various classes collaborate with each other.

Reiss visualizes the dynamics of Java programs in real time, *e.g.*, the number of message sends received by a class [Rei03]. The visualizations and associated analysis techniques proposed are not tightly integrated in an IDE, but are provided by a separate tool. Consequently, a developer cannot directly exploit these analyses while working with source code.

While many IDEs such as Eclipse [Ecl03] or Squeak [Squ] include a debugger tool for analyzing the runtime of a system, debuggers themselves are not intended for supporting the navigation of the source space. Usually a developer initiates a debugging session to gain a detailed insight into a specific run of the system, observing and analyzing a slice of the program often to discover the cause for a specific defect [Wei81, Zel03]. However, such a restricted view does not reveal an overall picture of the run-time behavior of the system.

Some IDEs support the navigation of large software systems using techniques other than program analysis. For instance, NavTracks [SES05], an extension for Eclipse [Ecl03], keeps track of the navigation history of software developers. Based on this history information, NavTracks forms associations between related source files (*e.g.*, class files) and can thus present related entities to the developers. Mylar [KM05] monitors a programmer's activity in the IDE to obtain a degree-of-interest model for program elements scattered across a large code base. By doing so, the IDE can reveal code elements that are likely to be important for the task at hand [KM05].

In the future, the notion of integrating dynamic analyses and dynamic models in IDEs represents a promising research direction. In this context, several issues need to be addressed: (i) complete coverage of a system is difficult to achieve with dynamic analysis [Bal99], (ii) the efficiency of dynamic analyses is often poor, and (iii) it is challenging to find suitable visualizations for presenting the vast amount of information typically resulting from dynamic analysis.

Other ideas [KM05] aim at studying the working patterns of developers to eventually adapt IDEs to the context of the task a developer is currently performing. If for example, a developer is carrying out a maintenance task (such as correcting a defect), the IDE could present only the information required for the task at hand or could even suggest possible corrections for the defect. By mining the version history of a system, the IDE could for instance present what code developers changed in the past when working on a similar task [ZWDZ04].

4. Model-centric development. There is a long history of reflective programming languages, ranging from dynamic languages such as Lisp, Smalltalk, Scheme and CLOS, to static languages like C++ and Java, which provide a more limited form of run-time introspection rather than full intercession. All of these approaches are limited to models of the code base, and do not take models of requirements, design decisions or architecture into consideration.

Over the years various approaches have attempted to keep high-level knowledge about software in sync with the software itself. The earliest examples of these is probably Literate Programming [Knu92], in which documentation and source code are freely interspersed and maintained together.

In some cases Architectural Description Language (ADL) specifications are considered to be part of the running software system, rather than simply a higher-level description of it, as it is the case with Darwin [MDEK95]. Although ADLs provide a high-level interface for specifying and configuring components at an architectural level, there is not really any explicit representation of a model that is developed in tandem with the rest of the software.

Generative programming approaches [CE00] produce software from higher-level descriptions using such mechanisms as generic classes, templates, aspects and components. A general shortcoming of these approaches is that the transformation is uni-directional — there is no way to go from the code back to higher-level descriptions. Round-trip engineering refers to approaches in which transformations are bi-directional [AC06]. Models and code are still considered to be separate artifacts, so models are not available at run-time for making adaptive decisions.

Case tools and 4GLs represent an attempt simplify the generation and adaption of an application. However the main focus of these approaches was to generate code and not to consider models as executable artifacts of software development.

Model-driven engineering (MDE) [Sch06] [BG01] [MDA] refers to a more recent trend in which application development is driven by the development of models at various levels of abstraction. Platform-independent models are transformed to platform-specific models, and eventually to code which runs on a specific platform. Generally these transformations are performed off-line, so the models are not necessarily available to the run-time system, though some approaches support this [HP05].

The Eclipse Modeling Framework (EMF) [BSM$^+$03] provides facilities for manipulating models and generating Java source code from these models. Here too the focus is on models of source code, rather than on other views of a software system. The model and the code are still separate entities.

Naked objects [Paw04] is an approach to software development in which domain objects and software entities are unified. Business logic is encapsulated in the domain objects and the user interface is completely generated from these domain objects. In this approach the domain model and the executing runtime are tightly coupled. Although naked objects address the earlier complaint against approaches which separate domain models from the source code or the running system, they do not offer any help in integrating other views of the software as it is being developed (*i.e.*, such as requirements models, architectural views, and so on).

References

[AC06] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 692–706, Berlin, Germany, 2006. Springer-Verlag.

[Bal99] Thomas Ball. The concept of dynamic analysis. In *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, number 1687 in LNCS, pages 216–234, Heidelberg, sep 1999. Springer Verlag.

[BDET05] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. On the use of clone detection for identifying cross cutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings Automated Software Engineering (ASE 2001)*, pages 273–282, Los Alamitos CA, 2001. IEEE Computer Society.

[BK04] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of International Conference on Automated Software Engineering (ASE 2004)*, pages 310–315, 2004.

[BSM$^+$03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[BZ06] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language driven development, 2004.

[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.

[CM03] Davor Cubranic and Gail Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.

[Dek03] Uri Dekel. Revealing JAVA Class Structures using Concept Lattices. Diploma thesis, Technion-Israel Institute of Technology, February 2003.

[DK97] A. van Deursen and P. Klint. Little languages: Little maintenance? In S. Kamin, editor, *First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL'97*, pages 109–127, January 1997.

[DRW00] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[Ecl03] Eclipse Platform: Technical Overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.

[GMM$^+$98] Robert Godin, Hafedh Mili, Guy W. Mineau, Rokia Missaoui, Amina Arfi, and Thuy-Tien Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[Gro04] Object Management Group. Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group, 2004.

[HDL00] Marianne Huchard, Hervé Dicky, and Hervé Leblanc. Galois Lattice as a Framework to specify Algorithms Building Class Hierarchies. *Theoretical Informatics and Applications*, 34:521–548, 2000.

[HLBAL05] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 112–121, Los Alamitos CA, 2005. IEEE Computer Society Press.

[HP05] Stefan Haustein and Jörg Pleumann. A model-driven runtime environment for web applications. *Software and System Modeling*, 4(4):443–458, 2005.

[KM05] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[Knu92] Donald E. Knuth. *Literate Programming.* Stanford, California: Center for the Study of Language and Information, 1992.

[KQ05] Rainer Koschke and Jochen Quante. On dynamic feature location. *International Conference on Automated Software Engineering, 2005*, pages 86–95, 2005.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated definition of abstract and concrete syntax for textual languages. In *MoDELS*, pages 286–300, 2007.

[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.

[MDA] OMG model driven architecture. http://www.omg.org/mda/.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeffrey Kramer. Specifying distributed software architectures. In *Proceedings ESEC '95*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag, September 1995.

[MDM07] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[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.

[MM00] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000)*, pages 46–53, November 2000.

[MM01] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, November 2001.

[MM03] Andrian Marcus and Jonathan Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 125–135, May 2003.

[MNS95] G. Murphy, D. Notkin, and K. 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.

[MP05] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frédérick Fondement, and Jean Bézivin. Independent web application modeling and development with netsilon. *Software and System Modeling*, 4(4):424–442, November 2005.

[MSRM04] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, November 2004.

[Ode07] Martin Odersky. Scala language secification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, March 2007.

[Ous98] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[Paw04] Richard Pawson. *Naked Objects.* Ph.D. thesis, Trinity College, Dublin, 2004.

[PFGJ02] Martin Pinzger, Michael Fischer, Harald Gall, and Mehdi Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*, pages 170–178, 2002.

[RD06a] Daniel Raţiu and Florian Deissenboeck. How programs represent reality (and how they don't). In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, Los Alamitos CA, 2006. IEEE Computer Society.

[RD06b] Daniel Raţiu and Florian Deissenboeck. Programs are knowledge bases. In *Proceedings of the 14th International Conference on Program Comprehension, (ICPC 2006)*, pages 79–83, Los Alamitos CA, 2006. IEEE Computer Society.

[Rei03] Steven P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.

[RJ07] Daniel Raţiu and Jan Juerjens. The reality of libraries. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering, (CSMR 2007)*, pages 307–318. IEEE Computer Society, 2007.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. Navtracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, sep 2005.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[Squ] Squeak home page. http://www.squeak.org/.

[ST98] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[TA99] Paolo Tonella and Giuliano Antoniol. Object oriented design pattern inference. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 230–238. IEEE Computer Society Press, October 1999.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM TOPLAS*, 2008. to appear.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WH92] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[WP07] Alessandro Warth and Ian Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.

[WS95] Norman Wilde and Michael Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[Zel03] Andreas Zeller. Program analysis: A hierarchy. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis*, pages 6–9, 2003.

[ZWDZ04] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.

2.2 Research Fields

The Software Composition Group carries out research in programming languages and software engineering methods to support the construction of flexible and open software systems. In recent years the research has focussed on techniques and mechanisms to support software evolution. In this section we review recent SCG publications that are especially relevant to this proposal.

All papers can be downloaded from: `http://www.iam.unibe.ch/~scg/cgi-bin/scgpubs.cgi`

1. Coordinating models and code. Over the past 11 years we have developed Moose[1], a platform for reverse engineering that provides a flexible framework for building complementary analyses [NDG05].

We have used Formal Concept Analysis to detect implicit collaborations between classes based on source code structure [ABN04]. Another use for FCA was to identify co-change patterns revealed by entities that change in the same time throughout history [GDK+07].

We have also used clustering techniques such as simulated annealing to detect implicit patterns in source code to attempt to improve its structure [PN06].

We employed Latent Semantic Indexing to go beyond the structure of the code and to retrieve implementation topics by analyzing the names of identifiers [KDG07]. We named the approach Semantic Clustering and used it to identify groups of classes that share similar vocabulary. Based on this information we were able to detect both concepts that are well encapsulated and those that are crosscutting.

Dynamic analysis exposes the runtime behaviour of the system. Our approach was to relate the runtime control flow with the code structure and thus to enrich the structure with collaboration [RD02] and feature information (*i.e.*, units of domain knowledge) [GD05]. Recently, we also tracked how objects are passed through the system and inferred higher level knowledge about how features depend on each other [LGN07].

To analyze the structure of software systems, we first need to parse the source code. However, building a parser is a difficult endeavor. That is the reason why we have investigated the use of examples to construct a parser automatically [NKG+07].

Duplicated code implies hidden dependencies between different parts of the system. We investigated a lightweight approach to detect code clones by means of string matching [DNR06]. We also correlated code cloning with author information and detected that cloning can lead to inconsistencies when several authors are involved [BGM06].

2. Embedding domain models in the code. Piccola is an experimental language for expressing applications as compositions of software components [NA05, AN05]. The guiding principle of Piccola is that components that support a particular domain should provide a high-level API that allows these components to be "scripted" using high-level operators. As such, each API defines a DSL as a set of operators. With Piccola, the resulting DSLs are separated from the host language. No special support was provided in Piccola to develop or use these DSLs, aside from the ability to overload syntactic operators.

Further research showed the advantages of using a dynamic, reflective language such as Smalltalk as a host for embedding DSLs [DG06]. Over the past years, SCG has built and collaborated on several systems that use a dedicated domain-specific language at its heart. Mondrian [MGL06] is a visualization engine providing a scriptable DSL (embedded in Smalltalk) to express complex visualizations. Many of our research projects use Mondrian to visualize our data.

Seaside [DLR07] is a dynamic web application framework. It implements different DSLs as part of the core framework: for example XHTML markup and JavaScript code snippets are generated programmatically, facilitating an almost declarative way to build user interfaces for the web. Seaside[2] is developed as an open-source project and used in many industrial scale projects.

[1]`http://moose.unibe.ch`

[2]`http://www.seaside.st/`

Magritte [RDK07] provides a generic meta-description framework. Several interpreters have been written that allow one to automatically generate user-interfaces or object-serialization mechanisms. Magritte offers a declarative DSL to dynamically describe classes and objects in the system. Moreover it enables end-users to change these meta descriptions on the fly, through a visual user interface that provides almost the same expressive power as the DSL to specify the meta-model. Magritte is used widely in industrial settings to facilitate run-time adaptive behaviour.

3. Bringing dynamic models to the IDE. We have analyzed how objects flow over their whole life cycle at runtime through the execution of a software system by taking into account object aliasing [LDGN06]. This is complementary to trace-based approaches because the runtime of a software is heavily dependent on the flow of objects, which is not visible in traces. We have created visualizations highlighting the flow of objects at runtime.

Various visualizations can help to convey large amounts of information very compactly. Polymetric views [LD03] and class blueprints [LD01] have been used largely to convey static information, but they can also be used to express evolution over time, or dynamic information [DLB04]. With integrated feature-centric views [RGN07] we have made features explicit in the IDE. These feature views give us insights in the dynamics of features and allow us to quickly navigate static source artifacts constructing specific features. In the context of a maintenance task, we extended the feature-centric views to highlight candidate classes or methods that might cause features to be defective. We evaluated this work by means of an empirical study.

The backward-in-time debugger we have implemented enables us to navigate back in the history of a system [HDD06]. While traditional debuggers just reason about one slice of a program's execution (*i.e.*, the stack trace), the backward-in-time debugger remembers the whole history, *i.e.*, all states of an application. This debugger is hence well-suited to locate causes for defects that are hard to correct using a traditional debugger.

We have also identified other candidate enhancements to IDEs, *e.g.*, information missing in current development environments [SB04]. For instance, having readily available key information about dynamic collaborations between classes in *e.g.*, aggregation and delegation relationships enhances program understanding and efficient navigation in the IDE. In the past, we implemented dynamically computed virtual categories for the methods of a class, *e.g.*, a method category holding all methods that a class needs to implement to be able to handle all messages sent to instances of it [SB04].

4. Model-centric development. Piccola [NA05, AN05] was designed to be a pure composition language in which applications could be expressed purely as compositions of software components bound together by high-level connectors. Although it is possible to demonstrate this ideal with the implemented Piccola language, it is nevertheless non-trivial to develop the component interfaces and the high-level connectors, as Piccola provides no special support for this. Notably, Piccola itself provides no language mechanisms for components or connectors, but rather much lower-level concepts (agents and channels) which have to be used to define a component composition style.

Context-Oriented Programming (COP) [HCN08] refers to programming language support for developing applications whose behaviour depends on the run-time context. Present prototypes of COP languages focus on mechanisms for adapting behaviour to context, but provide little support for reasoning about context at the model level.

Changeboxes [DGL$^+$07] provide a mechanism to control the scope of change in a running system. Deployment and development versions of a running software system can co-exist without interfering. Mechanisms for merging differences and resolving conflicts, however, must be handled in *ad hoc* fashion, as no fully general approach exists for all usage scenarios. Changeboxes currently operate at the level of source code changes. There is no notion of changes to higher-level models.

Sub-Method Reflection [DDLM07] is an approach we have developed in which a fully abstract representation of source code is available at run-time to support dynamic adaptations. Source code is adapted using a high-level reflective API (*i.e.*, at the level of the abstract syntax tree (AST)), and methods are dynamically recompiled on-demand. This approach has practical applications for dynamic analysis tools [RDT08]. We also believe that this offers a good starting point for the run-time infrastructure of a model-centric development environment in which code and models co-evolve.

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.

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

[BGM06] Mihai Balint, Tudor Gîrba, and Radu Marinescu. How developers copy. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 56–65, 2006.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):231–251, October 2007.

[DG06] Stéphane Ducasse and Tudor Gîrba. Using Smalltalk as a reflective executable meta-language. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 604–618, Berlin, Germany, 2006. Springer-Verlag.

[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)*. ACM Digital Library, 2007. To appear.

[DLB04] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 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.

[GD05] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.

[GDK$^+$07] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Daniel Raţiu. Using concept analysis to detect co-change patterns. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2007)*, pages 83–89, 2007.

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

[HDD06] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.

[KDG07] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.

[LD01] Michele Lanza and Stéphane Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of 16th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01)*, pages 300–311. ACM Press, 2001.

[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.

[LDGN06] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[LGN07] Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC'07)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.

[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.

[NA05] Oscar Nierstrasz and Franz Achermann. Separating concerns with first-class namespaces. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*, pages 243–259. Addison-Wesley, 2005.

[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.

[NKG$^+$07] Oscar Nierstrasz, Markus Kobel, Tudor Gîrba, Michele Lanza, and Horst Bunke. Example-driven reconstruction of software models. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 275–286, Los Alamitos CA, 2007. IEEE Computer Society Press.

[PN06] Laura Ponisio and Oscar Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.

[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.

[RDK07] Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte — a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer-Verlag, September 2007.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.

[RGN07] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Feature driven browsing. In *International Conference on Dynamic Languages (2007)*, 2007. To appear.

[SB04] Nathanael Schärli and Andrew P. Black. A browser for incremental programming. *Computer Languages, Systems and Structures*, 30(1-2):79–95, 2004.

2.3 Detailed Research Plan

We are proposing four research tracks that seek to bring models closer to code. The overall objective is to strengthen the role of models in the development process by (i) integrating them more closely with code, (ii) offering more opportunities to link models together, (iii) exploiting these links during development and run-time, and (iv) even to shift the emphasis from code towards models as the target artifact being produced.

In each track we build on our previous work, develop innovative prototypes to experiment with new techniques and approaches, and carry out case studies to validate the ideas. Generally speaking, we combine both fundamental and empirical approaches to ensure relevance of our research.

1. Coordinating models and code. Most existing reverse engineering techniques focus on extracting information from the code automatically and in the absence of external human knowledge. However, more often than not, external information is available (*e.g.*, developer knowledge or domain knowledge) and should be taken into account. We want to complement existing automatic techniques with annotations that encode different types of external knowledge, and provide mechanisms to enable multiple models to be effectively exploited.

- *Enriching code models with external knowledge.* We envision a system in which the model of the code is enriched with annotations. These annotations will not be just free text, but they will be structured according to an extensible meta-model. The body of annotations can thus grow and be refined as the reverse engineer learns about the system from various sources.

  Even though developer knowledge is present, it is not always explicit, and thus it may be difficult to encode it in annotations. We aim to investigate several techniques that can be used to guide the reverse engineer in defining and maintaining such annotations.

  To make external knowledge easily usable by reverse engineers, we intend to extend our Moose platform with the notion of annotation and to provide editors for manipulating both the actual annotations and their descriptions.

  To make effective use of multiple models, we need ways to navigate between different views. We will develop techniques to exploit the links between models so as to enable easy navigation between different views, and to support queries that span multiple models.

- *Mining higher-level views.* Once multiple views of a complex software system are available, and in particular source code models are augmented with human knowledge, it becomes possible to identify higher-level patterns that would otherwise only be implicit in the code.

  Clustering techniques can be used to identify parts that are similar based on a distance metric. However, the resulted clusters often contain many false positives. We want to develop an iterative approach that allows the reverse engineer to encode constraints while inspecting the results, and then to take them into account in a further iteration of applying the clustering. With this approach we can refine the clustering towards a more realistic state. The challenge in this part is how to devise the clustering machine to take the annotations into account.

  Semantic clustering can be used to identify implementation concepts in the code and to infer labels for describing these clusters. We want to use the labels to identify possible concepts that can be used in annotations.

  Feature location through dynamic analysis is a technique for recovering the mapping between source code artifacts and features (*i.e.*, behavioral units that encode domain knowledge). Once the mapping is recovered we can propose annotations based on the corresponding features.

  Formal Concept Analysis can be used to obtain a lattice of concepts formed by elements that have a set of properties in common. We want to use annotations to mark parts of the model, to then identify these parts in the lattice and then to use the lattice to discover further parts that are related to the marked parts. For example, we can mark parts of a known pattern (*e.g.*, design pattern) and then identify the rest of the pattern by navigating the lattice starting from the marked parts.

An ideal software system can be seen as a set of components that collaborate by means of APIs defined by clear interfaces. In real life software systems the interfaces may not always be clearly defined, and consequently, APIs are not easily detectable by automatic means. However, the correct way to use APIs can be recognized by developers during code inspection. We intend to allow reverse engineers to encode this information with annotations. At a later stage, we want to mine recurring patterns in the design of APIs, and we intend to use this knowledge as input for transforming these API into DSLs.

2. Embedding domain models in the code. Internal DSLs often lack expressiveness as they are strictly bound to the host language. External DSLs require the user to learn a new language and they miss sophisticated tools the host language should provide, such as native debuggers. Often DSLs are written in an *ad hoc* fashion, they may be poorly documented, hard to understand and impossible to debug as the necessary tools are missing.

We plan to provide an adaptive model for DSLs to be used by common tools such as the compiler, code editor, debugger, inspector, refactoring tools, and eventually also by the Virtual Machine (VM). Such a model should facilitate building DSLs by incrementally changing the model of the underlying language. Several language models should be executable and interoperable in parallel. We identify the following steps to reach this goal:

- *Seamless integration of DSLs in the IDE.* IDEs needs to provide additional tools to leverage the use of the DSL. For instance, when working in the context of a DSL, the IDE should enable dedicated syntax highlighting or code completion mechanisms. Switching between different DSLs should be possible on any level, even within a single method.

- *IDEs need to provide mechanisms for migrating code to use a specific DSL.* The system could suggest where and how the developer should refactor existing code, or automatically perform these refactorings based on a set of rules.

- *Advanced transformations for DSL code optimization.* DSLs provide a high level of abstraction, but often come at the cost of high performance and memory penalty. Therefore most DSLs involve some sort of code transformation, for example to avoid the creation of intermediate objects. We would like to integrate advanced transformations in our model, so that DSL code can be easily optimized without having its designers having to concern themselves with speed and performance issues. It is crucial that this transformation happen transparently, so that developers do not have to deal with transformed code but only with source actually visible in the code editor.

- *Development tool integration.*As a next step we plan to integrate other development tools with our model. Initially we will focus on the debugger. The lack of dedicated tools to find and fix bugs in a DSL is one of the major drawbacks when designing and using a DSL. The debugger should facilitate stepping through the code, inspecting and changing intermediate values, as well as modifying the code on the fly. Most important the debugger should work in the context of the underlying language, no matter if this is the host language or a custom DSL. Code transformations should not be visible, or only if the user explicitly wants to see them.

- *Validation.* In parallel to the above points, we plan to apply our approach to existing and new DSLs. We plan to analyze the transition of the Seaside web application framework from version 2.6 to 2.8, where the XHTML generator changed from a traditional API to a powerful DSL. As we are actively involved in the Seaside and several other open-source communities, we would like to implement their DSLs with our model. Subsequently we would like to release our model, so that we can collect feedback from industrial users.

3. Bringing dynamic models to the IDE. Current IDEs focus on providing the developer only with a static view of the source code, and this deprives the developer of information about how the code is actually executed at runtime, about why a bug occurred, about whether there are performance issues or memory consumption problems. Furthermore, valuable information which could help the developer to navigate the source code to those artifacts that are relevant to the task at hand (such as which components support a given feature) is simply missing.

Thus, we want to focus on bridging the static and dynamic views of the system by bringing the dynamic information to the IDE:

- *Make dynamic communication between classes and methods explicit.* Dynamic communication between objects occurring in a software system is often only implicitly visible when browsing the static source code. Due to polymorphism and late binding in object-oriented languages, a developer is unable to detect from the code to what objects messages are sent at runtime, or what type of objects are stored in variables. We want to make explicit in the IDE how classes and their objects dynamically collaborate with each other, *e.g.*, to which objects another object delegates.

- *Enrich the source code with the result of dynamic analysis (*e.g.*, exact types of variables).* The developers need the runtime information readily available where and when they actually work with the system. Nowadays, developing software predominantly means working with a textual representation or model of a system, *i.e.*, source code. Thus, we propose to enrich this source code with information gained by means of dynamic analysis. For example, we depict variables written in source code with the type of objects those variables get at runtime.

- *Support for understanding by presenting the dynamic information to the developer through the means of visualizations.* Visualization such as polymetric views can convey much information to the developer that is otherwise hard to grasp. But these visualization are either not integrated in the IDE or are not taking runtime information into account. We thus plan to first integrate useful visualizations such as class blueprints in the IDE, second to extend those visualizations with dynamic information and to finally invent new visualizations supporting the developer in understanding, maintaining and evolving software systems.

- *Automatic execution of software systems.* Most current approaches to dynamic analysis rely on the human to execute the system explicitly. Continuous integration goes a step forward providing an automatic process of running the system. However, none of these approaches bring the resulting data in the IDE. We intend to build an IDE that runs the system in the background and presents the information resulting from these runs. During these runs, the IDE checks for instance all specified pre- and post-conditions. All failing conditions are immediately reported to the developer working with such an IDE.

- *Covering the entire system with dynamic analysis.* A well-known issue of dynamic analysis is that it normally does not cover the entire system, but only those parts that have been explicitly triggered manually by the user or by recorded scripts such as test cases. We hence intend to work on an IDE capable to cover all aspects or features of a software system. The long-term goal is to let the IDE automatically cover all parts of a system that have to be covered for the current task the developer is working on.

As an IDE is a complex tool used by human beings, the effect of extensions or applied improvements to this tool need to be assessed with humans actually working with it. Upfront it is unclear what tools, mechanisms or enhancements actually contribute to a more efficient use of the IDE or to a better understanding for a software. It is hence crucial to perform elaborated empirical experiments with real users to gather qualitative and quantitative feedback on the effects of enhancements to the IDE. We plan to conduct several empirical studies with developers from both academia and industry to collect valuable and informative data validating the resulting impacts of all the extensions to the IDE we are going to implement.

In particular, we want to assess the effect of our work in the IDE on program comprehension, software maintenance and software evolution. We want to gather qualitative feedback from the studies subjects (*e.g.*, how the personally assess the effect of an implemented enhancements) and quantitative data (by *e.g.*, measuring the time to perform a certain task) to get a reliable measure for the effect of a specific enhancement.

4. Model-centric development. There is increasing pressure in many application domains (*e.g.*, financial instruments, health insurance, e-commerce) to quickly adapt software to changing business concerns. The required turnaround may be in the order of days or even hours. The adaptiveness of applications is hampered by the fact that the only reliable view is its source code. Developers need to efficiently transform new and changing requirements of stakeholders. Adapting views of the application that are easily understood by both developer and stakeholder would promote shorter development cycles.

With the preceding three activities we have focussed on how to bring models closer to source code for software that already exists. In a very real sense, many of the problems we have in keeping models and code synchronized arise from the fact that we tend to view "code" as the final product of the software development lifecycle. Since code is typically manifested as source files which are technically divorced from the models that drove their development, we arrive at a situation where disconnected artifacts slowly drift way from each other, and implicit knowledge is gradually lost over time.

In this activity we propose to explore a fundamentally different approach to developing software in which *executable models*, rather than code, are the end product. Rather than being "model-driven" and merely generating code from models, we propose a *model-centric* approach in which models themselves are the key software artifacts. As with the other activities, models offer multiple, coordinated views of a complex software system. Here, however, rather than producing source code in the end, we propose to produce executable models as the run-time artifact, which will represent just one model amongst many coordinated models. Our coordinated models provide a sound basis for program comprehension, thus easing the adaptive capabilities of the system.

We propose the following tasks to build an experimental model-centric development environment supporting our proposal of model-centric development.

- *Demonstrator use cases.* We will identify a number of plausible usage scenarios to demonstrate model-centric development. The use cases should emphasize very different kinds of models, and should range from easy to more difficult applications. For example, state-transition modeling of web applications should be a natural domain that lends itself well to generation of executable models. Modeling of pre- and post-conditions or safety and liveness conditions for concurrent applications would be far more ambitious. The demonstrators will serve (i) to establish requirements, (ii) as case studies for validation of the research.

- *Run-time for executable models.* We will develop a run-time system to support executable models. We propose to take our Reflectivity framework [DDLM07] as a starting point, since it already supports a higher level view of source code than the pure textural perspective by providing support for annotations and fine-grained structural reflection. At present Reflectivity models methods as abstract syntax trees, and compiles methods on-the-fly when reflective adaptations are performed. To achieve our vision of model-centric development, we would need to extend the framework to support models at a higher level of abstraction than ASTs, and at arbitrary granularities (*i.e.*, not just at the method level, but also that the level of user features, requirements and architecture). Furthermore we would need to introduce more advanced mechanisms for synchronization and merging of reflective adaptations, to avoid inconsistencies that can arise when adapting models that are currently executing. We plan to extend our previous work on first-class contexts to control the scope of changes. Models will need to be self-describing so that tools can manipulate them. Reflectivity currently supports the use of objects as annotations. We plan to exploit this feature to enable models to be fully self-describing.

- *Model-aware tools.* Model-aware tools should offer the user the means to interact with and manipulate models. Transformations performed on these models should then be reflected back into the run-time environment where these models reside. Initially we will develop proof-of-concept tools driven by the demonstrator applications. For example, a simple state-transition editor could be used as a front-end to manipulate models whose behaviour is expressed as a state machine. We expect to leverage the results of tasks 2 and 3 as well. Specifically, we would use specialized editors for DSLs to manipulate models that are expressed in a textual language. By the same token, rather than developing a new IDE, we would leverage the work being done in bringing dynamic models to the IDE to not only display those models, but to manipulate them and reflect changes back into the system.

  Not all models are executable. A part of the work will address the expression of models at the level of requirements, domain knowledge, architectural constraints and design knowledge. These models should be linked into the executable models so that they are available at run-time for tools that can use this knowledge (*e.g.*, to generate suitable user interaction dialogues), and available to the developer, as described in the third activity above. Since essentially all models can be represented as graphs, we envision a generic environment to manipulate self-describing models as graphs. Building such an environment would be beyond the scope of this activity, but we expect to make some steps in this direction through the development of a series of simple prototypes.

- *Transformation engine.* Actions performed with model-aware tools will need to be reflected back to the run-time. For the first simple prototypes, these transformations will necessarily be *ad hoc*. In the long-term, a general environment to support model-centric development should provide generic support for propagating action on models back to the run-time. A general transformation engine would also be needed to translate between models, to navigate and query them, and to translate between representations, such as graphical and textual views). Existing refactoring engines represemt a plausible starting point for this work. We plan to start with concrete demonstrator applications, and explore the development of a more generic transformation engine based on our initial experiences with the concrete examples.

2.4 Timetable

Mr. Lukas Renggli started his PhD in August 2006, and will continue to work on the topic of *Embedding domain models in the code.* Mr. David Roethlisberger started his PhD work in October 2006, and will continue to work on the topic of *Bringing dynamic models to the IDE.* Two new PhD students will be engaged to start work on the remaining topics. (We are currently interviewing candidates.)

We expect to obtain the following results over the two years of the project. We expect a rough correspondence between bullet items below and publishable units.

Year 1	
Coordinating models and code	– Extending Moose to support multi-model annotations. – Applying clustering techniques to extract higher-level abstractions from multiple models.
Embedding domain models in the code	– Case studies: analyze the relationship between APIs and DSLs. – Develop a model for specifying DSLs in host language. – Develop (adapt) debugger to work with this model. – Demonstrations applied to Mondrian and Seaside.
Bringing dynamic models to the IDE	– Make dynamic communication between classes and methods – explicit in the IDE. – Enrich source code with dynamic information. – Develop and embed visualizations in the IDE.
Model-centric development	– Extend Reflectivity framework to different classes of models. – Develop prototype model-aware tools for demonstrator applications.
Year 2	
Coordinating models and code	– Inferring domain models from API usage. – Combining user annotations and pattern recognition techniques to identify idiomatic API usages.
Embedding domain models in the code	– DSL transformation to host language, for optimization. – Refactoring APIs towards DSLs.
Bringing dynamic models to the IDE	– Develop techniques to execute software systems automatically – in the IDE. – Covering the entire system with dynamic analysis and integrate the results in the IDE. – Conduct empirical experiments validating the implemented techniques with users in the field.
Model-centric development	– Develop context mechanism to control scope of change. – Develop transformation engine to support arbitrary models.

2.5 Significance of the Research

Software is becoming longer-lived due to improvements in processes, languages and tools. As a consequence, the proportion of the software lifecycle devoted to "maintenance" (*i.e.*, further development after initial deployment) has increased to anywhere from 50% to 70% of overall effort. Technological advances are constantly leading to new business models and hence to increasing demands to adapt existing software to meet these new requirements. As a consequence there is increasing pressure for new techniques to shorten the length of time from change request to deployment of new features.

Further advances in this area are only possible by lessening the gap between models and code. Model-driven techniques are a step in this direction, but they are still mired in the focus on code as the primary artifact to be produced. Fundamental research is needed on many tracks to develop the next generation of model-centric programming languages, tools and environments. We estimate that the research tracks presented in this research proposal will have a positive impact on technology in the next 5-10 years. (By analogy model-driven approaches have taken about that length of time to mature and achieve some level of industrial adoption.)

As this proposal consists of foundational research, we focus on the traditional means to disseminate results, namely publications in high-impact, peer-reviewed, international venues, participation in key workshops, internet presence and establishing industrial contacts through tutorials, presentations and eventual collaboration on case studies.