Part 2 : Scientific Information

Main applicant:	Nierstrasz, Oscar
Project title:	Agile Software Analysis (ASA2)

Contents

1 Summary of the research plan

Software developers actually spend much of their time not producing new code, but analysing the existing code base. Integrated Development Environments (IDEs) however are mostly glorified text editors, and offer only limited support for developers to query and analyse software systems. In this continuation of our running SNF project[1], we proceed to explore new ways to enable developers to efficiently answer detailed questions about the software system under development.

Like its predecessor, this project is designed as four related PhD tracks (including two end-of-thesis subprojects). We avoid any critical dependencies between the tracks, while offering many possibilities for fruitful collaboration.

— **Agile Model Extraction.** Here we address the question: *"How can we rapidly extract models from unknown source code?"* The key insight is that we do not need precise parsers to extract useful models from software. By using *approximate parsing* techniques combined with heuristics to automatically recognise common programming language features, we expect to be able to iteratively construct analysable models of complex software systems in a fraction of the time it would take to build a conventional model importer.

— **Context-Aware Tooling.** In this track we tackle the question: *"How do we close the abstraction gap between application domains and IDEs?"* We plan to extend and generalise our previous work on highly configurable tools, such as debuggers, browsers and inspectors, that can easily be adapted to the context of a given project, and enhance both the source code and the IDE itself with domain-specific and project-specific information. In particular, we will focus on domain-specific ways to present and query software systems.

— **Ecosystem Mining.** Here we ask: *"How can we mine the ecosystem to automatically discover intelligence relevant to a given project?"* When evolving source code, developers commonly make use of dedicated social media (*i.e.*, question and answer fora) as well as general search engines to search for answers to technical questions. In addition to these sources, the *software ecosystem* of related software systems can be a rich source of useful information, if properly analysed. We plan to mine these ecosystems to locate common, reusable code examples, opportunities for refactoring, common bugs, and bug fixes.

— **Evolutionary Monitoring.** Finally we address the question: *"How can we steer the evolution of a software system by monitoring stakeholder needs, technical debt, and ecosystem trends?"* By monitoring the activities of the stakeholders (*i.e.*, both users and developers), we hope to infer their needs and the chronic problems of the system. By monitoring "technical debt" (*i.e.*, pain points where effort should be invested to avoid the future escalation of these problems), we expect to be able to better prioritise future development activities. By monitoring technical trends from the ecosystem, we expect to be able to detect new opportunities and also better estimate the cost of future change.

In addition to disseminating the results of the research through academic publications, we will continue to release software artifacts (tools, IDE extensions *etc.*) as part of the Pharo development environment and the Moose analysis platform used by a wide community of academic and industrial partners.

[1]`http://scg.unibe.ch/asa`

2 Research plan

This project seeks to better support software developers in developing and maintaining large, complex software systems by offering several ways to close the abstraction gap between software source code and the application domain. The project is constructed as four related PhD research tracks that share common themes to foster interaction and collaboration.

2.1 Current state of research in the field

The topic of this proposal reflects trends in software engineering research over the past twenty years. These trends include increased interest in software modeling and analysis, including large-scale analysis of software corpora and ecosystems, empirical studies to understand developer needs, novel tools and high-level languages to better support the development process, and increased focus on software architecture and "technical debt". In this section we highlight recent progress related to the goals of this project.

Software Modeling. Modern Integrated Development Environments (IDEs), such as Eclipse[2] or NetBeans[3], provide a number of tools for browsing, querying and developing software source code. Generally the querying facilities of such IDEs are rather limited, so to perform useful analyses, one must either resort to developing a custom-made tool that can interpret the internal data structures used by the IDE to represent source code, or one must use a dedicated software modeling and analysis platform. Moose [NDG05] and Rascal [KvdSV09] are examples of two such platforms. The former employs an extensible object-oriented metamodel and the latter an equivalent relational metamodel. Both environments support the formulation of high-level analyses and the computation of metrics. In both cases, however, the bottleneck is to generate the model from source code. This entails either the creation of a custom-built parser and model generator for the programming language used, or a model transformer that translates models imported by an existing tool (such as the IDE). Building a custom model generator implies significant development effort, while model transformers are rarely a feasible option.

Numerous *language workbenches*, such as Spoofax [KV10], have been developed to support the rapid development of new languages and language-based tools. Even with the help of such advanced tools, recovering the grammar of the host language and building a full parser can require weeks or months of effort [LV01]. Some interesting directions are to recover grammars from existing parse trees [KDM09] or to mine them from human-readable specifications [Zay12].

A key insight is that we do not necessarily need a full parser for a programming language to build a useful model. *Island grammars*, pioneered by Moonen [Moo01], offer a form of *imprecise parsing* or "semi-parsing" [Zay14] that distinguishes "islands" (*i.e.*, parts of the source code of interest) from "water" (*i.e.*, the parts to ignore). They are especially useful for extracting selected information from complex languages, but they are unfortunately very fragile and difficult to specify correctly and robustly since the specification of the water depends on the islands of interest.

An interesting direction is to exploit commonly occurring artifacts in software code to infer language constructs. For example, it is known that indentation is a good proxy for complexity

[2] `https://www.eclipse.org`
[3] `https://netbeans.org`

metrics [HGH08] and reflects faithfully the structure of code. There has been increased interest in dedicated parsing techniques for indentation-sensitive languages [ERKO12] [Ada13].

Advanced IDEs. In recent years there has been considerable interest in augmenting the IDE with dedicated support for special tasks and activities. For example, dedicated support for collaboration has been a recurring theme [HCRP04] [GBRvD15].

IDEs such as Eclipse enable extensibility through the development of "plug-ins". Such plug-ins can be difficult to develop due to the complexity of the IDE infrastructure. IDE customization can, however, be facilitated with the help of a suitable architecture [CFJ+09].

Researchers have experimented with new kinds of user interfaces to facilitate code navigation. Code Bubbles [BZR+10] and Debugger Canvas [DBR+12] break away from the traditional file-based view of source code and allow structure to emerge from interconnected visual entities that contain source code and reflect the execution of the program. These experimental interfaces do not, however, support querying or alternative presentation interfaces.

Other research has investigated the benefit of linking source code to other relevant artifacts, such as e-mails [BLR10], documentation [SIH14], or relevant web pages [SMJ13], though most of this work has focused on recovering links rather than novel ways to integrate the content of the links into the IDE.

Further work has explored novel means to query a code base. S^6 is a code search engine that supports semantics-based queries (*e.g.*, input/output pairs) [Rei09]. Sourcerer offers a relational interface to query a large corpus of open-source software [BOL14]. Portfolio combines natural language processing and ranking algorithms to identify relevant functional abstractions [MGP+11].

Software visualization is another important recurring theme. Stasko distinguishes two major categories: *program visualization* and *algorithm visualization* [SDBP98]. Lanza and Marinescu have developed numerous lightweight visualizations based software metrics to help developers assess their code base and detect possible anomalies [LM06]. Storey *et al.* have surveyed the use of visualization to support human activities in software development [SvG05]. Parnin *et al.* have surveyed lightweight visualizations for highlighting "code smells" [PGN08]. Roassal is the latest in a series of dedicated Domain-Specific Languages (DSLs) to support the rapid development of custom visualizations [ABC+13]. It is used within Moose to visualize software. Despite Roassal's expressive power, it remains a non-trivial task to design and implement a suitable software visualization to support a given developer task. Moreover, once generated, such a visualization is normally used as a third-party tool.

Large Scale Software Analysis. The increased availability of open-source version repositories as well as the ever-increasing size of the software ecosystems inside large industrial companies [MH13] bring challenges such as the difficulty of being able to verify that the code one relies on can co-evolve with other systems, but also opportunities, such as useful redundant information across projects, the possibility of detecting emerging patterns from the source code [RBK+13], and increased opportunities for reuse.

One direction in which considerable work has been invested is the detection of techniques and tools for clone detection together with empirical studies that assess the prevalence of the phenomenon [HJHC10] [SLR12] [RZK14]. The majority of the research effort is directed towards performance and scalability [Kos13]. Some effort is targeted towards actual applications, such as refactoring a system to remove duplication.

A rich body of work is concerned with automatically extracting API protocols to build recommender systems for developers [RBK+13] [ZXZ+09]. Another approach is augmenting an existing library documentation with examples crawled from the clients of a library [MBDP+15] or even integrating web search features inside the IDE to help the developer avoid context switching [BDWK10]. What most of these approaches lack is a strong contextualization that can increase the relevance of the results for the individual developer.

A direction that has been developing rapidly recently is defect prediction based on source code, version history, and organizational information[ABJ10]. Somehow at odds with the reported success of various approaches, a recent study of deployed bug prediction algorithms at Google found no identifiable change in developer behavior in the presence of prediction systems [LLS+13]. After surveying many evaluation techniques for bug prediction models, Jiang *et al.* [JCM08] concluded that one of the problems with the evaluation is that the system cost characteristics should be taken into account. Arisholm *et al.* [ABJ10] introduced the Cost Effectiveness metric and Mende and Koschke [MK10] integrated it into their own evaluation method. There is a strong need for more cost-aware prediction models that work down to the lowest abstraction levels possible [PFD11].

Monitoring Evolution. Monitoring the evolution of a system, with the help of continuous integration and architectural constraint monitoring are common software engineering practices. However, current monitoring systems (*e.g.*, Travis, Jenkins, Sonar) generally focus on relatively low-level or syntactic aspects of the system such as open issues reported by static analysers [ZVI+14]. Multiple types of deeply contextual information can be taken into account during evolutionary monitoring to increase the usefulness of the process.

Some of the evolving stakeholder needs [SMDV08] [FM10] can be extracted and analyzed automatically by monitoring the behavioral developer meta-data [KM06] [MMLK14] while others must be empirically elicited by running empirical studies with the developers [MTRK14]. We argue that just as user needs are routinely collected and monitored, developer need elicitation and monitoring should be supported as first-class activities in software engineering.

Every software system accumulates *technical debt* when short-term goals in software developent are prioritized over the long-term health of the project [NVK11]. A recent study compared various approaches to computing technical debt and discovered that the results of different techniques are loosely coupled and therefore indicate problems in different locations in the source code [ZVI+14]. This indicates the need for combining detection strategies that are based on multiple sources of information, including static source code analysis [LM06] [YM12], simulations to estimate the benefits of different investment strategies, and information regarding the trends of usage of the used technologies [DRNN14].

An emerging challenge is monitoring the health and emerging evolutionary trends for an entire software ecosystem [MG13] [Jan14]. Ecosystem-derived information used in the context of a single system can augment a developer dashboard with intelligence about inter-related projects and collaborators [BZ10], and support solving a range of problems including breaking changes[RvDV14], library migration [TFPB13], or security vulnerabilities in the used components [CBVvD15]

References

[ABC$^+$13] Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, September 2013.

[ABJ10] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, January 2010.

[Ada13] Michael D. Adams. Principled parsing for indentation-sensitive languages: Revisiting Landin's off-side rule. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 511–522, New York, NY, USA, 2013. ACM.

[BDWK10] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.

[BLR10] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages 375–384. ACM Press, 2010.

[BOL14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259, January 2014.

[BZ10] Andrew Begel and Thomas Zimmermann. Keeping up with your friends: Function foo, library bar.dll, and work item 24. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 20–23, New York, NY, USA, 2010. ACM.

[BZR$^+$10] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512, New York, NY, USA, 2010. ACM.

[CBVvD15] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. Technical report, Delft University of Technology, March 2015.

[CFJ$^+$09] Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton Jr., Evelyn Duesterwald, and Jurgen J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. In Shail Arora and Gary T. Leavens, editors, *OOPSLA*, pages 191–206. ACM, 2009.

[DBR$^+$12] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: industrial experience with the code bubbles paradigm. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press.

[DRNN14] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM.

[ERKO12] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *SLE*, pages 244–263, 2012.

[FM10] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM.

[GBRvD15] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. Supporting developers' coordination in the IDE. In *Proceedings of CSCW 2015 (18th ACM conference on Computer-Supported Cooperative Work and Social Computing)*, page to appear, 2015. missing-doi.

[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.

[HGH08] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Indentation as a proxy for complexity metrics. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 133–142, Washington, DC, USA, 2008. IEEE Computer Society.

[HJHC10] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: incremental, distributed, scalable. In *ICSM*, pages 1–9, 2010.

[Jan14] Slinger Jansen. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology*, 56(11):1508 – 1519, 2014. Special issue on Software Ecosystems.

[JCM08] Yue Jiang, Bojan Cukic, and Yan Ma. Techniques for evaluating fault prediction models. *Empirical Softw. Engg.*, 13(5):561–595, October 2008.

[KDM09] Nicholas A. Kraft, Edward B. Duffy, and Brian A. Malloy. Grammar recovery from parse trees and metrics-guided grammar refactoring. *Software Engineering, IEEE Transactions on*, 35(6):780 –794, November 2009.

[KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM Press.

[Kos13] Rainer Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 2013. accepted for publication.

[KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *OOPSLA'10: Proceedings of the 25th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 444–463, Reno/Tahoe, NV, USA, October 2010.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, 2009.

[LLS+13] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? Findings from a Google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice.* Springer-Verlag, 2006.

[LV01] Ralf Lämmel and Chris Verhoef. Semi-automatic grammar recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[MBDP+15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can I use this method? In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, 2015.

[MG13] Tom Mens and Mathieu Goeminne. Analysing ecosystems for open source software developer communities. In S. Jansen, M. Cusumano, and S. Brinkkemper, editors, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry.* Edward Elgar Publishers, 2013.

[MGP+11] Colin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 111–120, May 2011.

[MH13] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems — a systematic literature review. *J. Syst. Softw.*, 86(5):1294–1306, May 2013.

[MK10] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 107–116, Washington, DC, USA, 2010. IEEE Computer Society.

[MMLK14] Roberto Minelli, Andrea Mocci, Michele Lanza, and Takashi Kobayashi. Quantifying program comprehension with interaction data. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 276–285, October 2014.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, October 2001.

[MTRK14] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4):31:1–31:37, September 2014.

[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, USA, September 2005. ACM Press. Invited paper.

[NVK11] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 1–8, New York, NY, USA, 2011. ACM.

[PFD11] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 362–371, Washington, DC, USA, 2011. IEEE Computer Society.

[PGN08] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 77–86, New York, NY, USA, 2008. ACM.

[RBK+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.

[Rei09] Steven P. Reiss. Semantics-based code search. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[RvDV14] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 215–224, September 2014.

[RZK14] Chanchal K. Roy, Minhaz F. Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future. In *Proceedings of the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 18–33, 2014.

[SDBP98] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization — Programming as a Multimedia Experience.* The MIT Press, 1998.

[SIH14] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[SLR12] Niko Schwarz, Mircea Lungu, and Romain Robbes. On how often code is cloned across repositories. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1289–1292, Piscataway, NJ, USA, 2012. IEEE Press.

[SMDV08] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34:434–451, July 2008.

[SMJ13] Nicholas Sawadsky, Gail C. Murphy, and Rahul Jiresal. Reverb: Recommending code-related web pages. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 812–821, Piscataway, NJ, USA, 2013. IEEE Press.

[SvG05] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis'05: Proceedings of the 2005 ACM symposium on software visualization*, pages 193–202. ACM Press, 2005.

[TFPB13] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migration in java software. *ArXiv e-prints*, June 2013.

[YM12] Aiko Yamashita and Leon Moonen. Do code smells reflect importat maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 306–315, September 2012.

[Zay12] Vadim Zaytsev. Notation-parametric grammar recovery. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 9:1–9:8, New York, NY, USA, 2012. ACM.

[Zay14] Vadim Zaytsev. Formal foundations for semi-parsing. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 313–317, February 2014.

[ZVI+14] Nico Zazworka, Antonio Vetro, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426, 2014.

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343. Springer Berlin Heidelberg, 2009.

2.2 Current state of own research

The PI, Oscar Nierstrasz, is a well-known expert in Software Evolution and Object-Oriented Languages. In 2013 he was awarded the prestigious Dahl-Nygaard Prize for contributions to Object-Orientation. Nierstrasz founded the Software Composition Group at the University of Bern in 1994 and has made numerous contributions to Component-based Software Development, Object-based Concurrency, Reverse Engineering, Programming Language Design, and Software Evolution. He has led an unbroken series of SNF projects since 1994, and has graduated 28 PhDs, many of which were funded from these projects.

In this section we mainly report on our progress in each of the four tracks of the precursor project, Agile Software Assessment (SNSF 200020-144126). PDFs of all published papers are available online.[4] The four tracks of the current proposal each follow logically from those of the precursor, though with a slight shift in emphasis.

Four PhD students (Jan Kurš, Andrei Chiş, Andrea Caracciolo and Boris Spasojević) were supported or partially supported from the SNF project. Additionally, three further PhD students (Nevena Milojković, Haidar Osman and Leonel Merino) working on related topics are funded by the University of Bern. We report on their progress as well insofar as it relates to this proposal.

Agile Modeling. In this track we are exploring techniques to rapidly construct models from source code by using imprecise parsers. The key insight we exploit is that we do not necessarily need a full and complete parser for a programming language to build useful models of systems written in that language.

An overview paper [NK15] describes both the prior work and the research plan to explore island parsing, indentation-aware parsing and automatic keyword recognition as steps towards rapidly building imprecise parsers for a given language.

Island grammars are very promising, but they are very difficult to specify correctly, and they are very fragile to change, since the specification of the "water" to be ignored depends on the islands to be extracted. We have therefore been developing a new approach to island parsing called *bounded seas*, in which the water is effectively computed from the context of the islands to be recognized [KLN14b] [KLN14a]. An extended journal paper on bounded seas has been submitted for publication.

A prototype of bounded seas has been implemented in PetitParser, a framework for building composable parsers developed at the SCG [KLR+13]. With the help of Masters and Bachelor students, we are carrying out case studies with Java and Ruby to determine the effectiveness and robustness of bounded seas vs. traditional island parsers.

Since we also want to exploit structural clues in code to infer model elements, we have been extending PetitParser to support indentation-aware operators. Numerous languages, like Python, Haskell and F#, rely on indentation to define structure, but their indentation-aware features differ in significant ways. An early prototype was developed as a Bachelors project [Giv13]. A full paper has been submitted for publication.

We have also started with the help of another Bachelors student to explore the use of heuristics to automatically identify different classes of tokens, such as keywords, in unknown programming languages. The initial results are quite promising, and we hope to extend the set of heuristics in a followup project.

Jan Kurš is expected to defend his thesis in the Spring of 2016. In the followup track on *Agile Model Extraction* a new PhD student will carry on this research.

Meta-Tooling. In this track we are investigating how to enable developers in rapidly adapting development tools to support specific application domains. To this end, we have produced a number of "moldable" developer tools. The Moldable Debugger [CNG13] [CGN14] is a debugging framework that can be easily adapted to different domains, such as event-driven systems, or

[4]`http://scg.unibe.ch/scgbib?query=snf-asa`

parsing. Andrei Chiş received the Best Student Paper award for his full paper on the Moldable Debugger at the Software Language Engineering (SLE) conference 2014. An extended journal paper has been submitted for publication.

Another example of such a tool is the Moldable Inspector, a configurable object inspector that can easily be adapted to a given domain [CNG14]. Andrei Chiş received a European Smalltalk User Group 2014 Technology Innovation Award (1st prize) for his implementation of the Moldable Inspector.

These results have contributed to the development of the PPBrowser tool, a dedicated environment for developing and debugging parsers built with PetitParser [KLR$^+$13]. Andrei Chiş will continue to work on this topic in the Context-Aware Tooling track of this proposal, and is expected to defend his thesis in September 2016.

In related work, Leonel Merino (PhD defense planned for Spring 2018) has been investigating how to offer very high-level support for analysis and software visualization of large software corpora [Mer14] [Mer15].

Large-Scale Software Analysis. In this track we have been exploring numerous ways to exploit the large amount of data available in both the immediate ecosystem of a given application, and the broader context of open source software written in the same language.

In early work, we have analysed the legacy PL/1 ecosystem of a large Swiss financial organization [ALNW13] [Aes13]. This work has inspired many of the threads outlined below.

Pangea[5] is a workbench for analysing multi-language software corpora [CCSL14]. Pangea provides a parallel infrastructure together with a specialized DSL that eases the analysis of a large number of software models interpreted by the Moose analysis platform.

We have been exploring numerous applications of large-scale software analysis. By ranking most commonly accessed methods based on their popularity in the ecosystem, we can offer developers a more productive browsing experience that leads more quickly to relevant source code [SLN14c] [SLN14b]. With Ecosystem-Aware Type Inference (EATI), we analyse the ecosystem of code written in Smalltalk, a dynamically-typed programming language, to determine to which concrete types given polymorphic variables are most commonly bound, and exploit this information back in the IDE [SLN14a]. We have analysed the prevalence of polymorphism in open-source Smalltalk and Java systems [MCL$^+$15], and we are investigating how to combine static and dynamic analyses to detect polymorphic usages cheaply and efficiently [Mil14].

Recently, we have started mining the creation of objects from large corpora to create a so-called “object repository” — an ecosystem-wide database of object creation code snippets. We are exploring the potential of such a repository to support testing, documentation, and program comprehension.

We are also analysing software corpora to mine which kinds of bug fixes occur most frequently. Interestingly, in Java code the most common bug fix is to insert a “null-check” to ensure that a variable being accessed is properly initialised [OLN14]. In followup work, we are proposing a new approach to bug prediction which is based on a model of profit and loss to better identify those portions of code most likely to contain critical bugs. A full paper is currently in submission

[5]`http://scg.unibe.ch/research/pangea`

and further empirical studies are being launched. We are also investigating the opportunities that come with data mining other types of languages beyond Java (*e.g.*, XML).

We have also carried out empirical studies with developers to better understand their needs with respect to the upstream (*i.e.*, providers) and downstream (*i.e.*, users) of ecosystem resources (*i.e.*, libraries and tools) [HLSN13] [HLSN14].

Finally, in work initiated earlier but completed last year, we have developed novel techniques to efficiently detect software clones in very large corpora (*i.e.*, all open-source Java code) [Sch14].

Boris Spasojević will continue his PhD in the successor project, and is scheduled to defend his PhD at the end of 2016 or early 2017. Haidar Osman and Nevena Milojković are both expected to defend their theses in the Spring of 2017.

Architectural Monitoring. In this track we are exploring ways to track the evolution of a complex software system and monitor possible violations of architectural constraints.

In the conclusion of earlier work, we have developed tools and techniques to recover software architecture from an existing code base [LLN14] and we have developed techniques to predict dependencies in software systems using domain-based coupling [APL+14].

We carried out two extensive empirical studies of software practitioners to determine what kinds of architectural constraints are most important in practice. A first qualitative study identified which kinds of constraints and concerns arise in industrial projects in practice and a second, quantitative study measured the relative importance of these constraints [CLN14b]. An important outcome was to identify the areas practitioners consider critical and where tool support is lacking.

In subsequent work, we have developed *Dictō*, a high-level domain specific language for specifying and testing architectural rules [CLN14a] [CLN15]. Based on the results of the empirical studies, Dictō was designed to be able to easily express a large variety of architectural constraints. Dictō can be connected to various back-end tools to perform the actual analyses. Dictō has already been connected to a large variety of tools, and we are now carrying out some larger case studies with external partners to assess its effectiveness.

In related work, we have developed *Marea*, a tool to detect and break dependency cycles in software systems using an explicit profit and cost model to determine which dependencies are most "profitable" to break [Aga15].

Andrea Caracciolo, the PhD student working on this track, is scheduled to defend his PhD thesis in the Spring of 2016.

References

[Aes13] Erik Aeschlimann. St1-PL/1: Extracting quality information from PL/1 legacy ecosystems. Masters thesis, University of Bern, December 2013.

[Aga15] Bledar Aga. Marea — a tool for breaking dependency cycles between packages. Masters thesis, University of Bern, January 2015.

[ALNW13] Erik Aeschlimann, Mircea Lungu, Oscar Nierstrasz, and Carl Worms. Analyzing PL/1 legacy ecosystems: An experience report. In *Proceedings of the 20th Working Conference on Reverse Engineering, WCRE 2013*, pages 441 – 448, 2013.

[APL+14] Amir Aryani, Fabrizio Perin, Mircea Lungu, Abdun Naser Mahmood, and Oscar Nierstrasz. Predicting dependencies using domain-based coupling. *Journal of Software: Evolution and Process*, 26(1):50–76, 2014.

[CCSL14] Andrea Caracciolo, Andrei Chis, Boris Spasojević, and Mircea Lungu. Pangea: A workbench for statically analyzing multi-language software corpora. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 71–76. IEEE, September 2014.

[CGN14] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The Moldable Debugger: A framework for developing domain-specific debuggers. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 102–121. Springer International Publishing, 2014.

[CLN14a] Andrea Caracciolo, Mircea Lungu, and Oscar Nierstrasz. Dicto: A unified DSL for testing architectural rules. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW '14, pages 21:1–21:4, New York, NY, USA, 2014. ACM.

[CLN14b] Andrea Caracciolo, Mircea Lungu, and Oscar Nierstrasz. How do software architects specify and validate quality requirements? In *European Conference on Software Architecture (ECSA)*, volume 8627 of *Lecture Notes in Computer Science*, pages 374–389. Springer Berlin Heidelberg, August 2014.

[CLN15] Andrea Caracciolo, Mircea Lungu, and Oscar Nierstrasz. A unified approach to architecture conformance checking. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. ACM Press, 2015. To appear.

[CNG13] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. Towards a moldable debugger. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, 2013.

[CNG14] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. The Moldable Inspector: a framework for domain-specific object inspection. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.

[Giv13] Attieh Sadeghi Givi. Layout sensitive parsing in the PetitParser framework. Bachelor's thesis, University of Bern, October 2013.

[HLSN13] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 1st Workshop on Ecosystem Architectures*, pages 1–5, 2013.

[HLSN14] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. A quantitative analysis of developer information needs in software ecosystems. In *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*, pages 1–6, 2014.

[KLN14a] Jan Kurs, Mircea Lungu, and Oscar Nierstrasz. Bounded seas: Island parsing without shipwrecks. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 62–81. Springer International Publishing, 2014.

[KLN14b] Jan Kurs, Mircea Lungu, and Oscar Nierstrasz. Top-down parsing with parsing contexts. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.

[KLR$^+$13] Jan Kurs, Guillaume Larcheveque, Lukas Renggli, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, September 2013.

[LLN14] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming*, 79(0):204 – 223, 2014.

[MCL$^+$15] Nevena Milojkovic, Andrea Caracciolo, Mircea Lungu, Oscar Nierstrasz, David Röthlisberger, and Romain Robbes. Polymorphism in the spotlight: Studying its prevalence in java and smalltalk. In *Proceedings of International Conference on Program Comprehension (ICPC 2015)*, pages 1–11, 2015. To appear.

[Mer14] Leonel Merino. Adaptable visualisation based on user needs. In *SATToSE'14: Pre-Proceedings of the 7th International Seminar Series on Advanced Techniques & Tools for Software Evolution*, pages 71–74, July 2014.

[Mer15] Leonel Merino. Explora: Tackling corpus analysis with a distributed architecture. In *SATToSE'14: Post-Proceedings of the 7th International Seminar Series on Advanced Techniques & Tools for Software Evolution*, 2015.

[Mil14] Nevena Milojkovic. Towards cheap, accurate polymorphism detection. In *SATToSE'14: Pre-Proceedings of the 7th International Seminar Series on Advanced Techniques & Tools for Software Evolution*, pages 54–55, July 2014.

[NK15] Oscar Nierstrasz and Jan Kurs. Parsing for agile modeling. *Science of Computer Programming*, 97, Part 1(0):150–156, 2015.

[OLN14] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. Mining frequent bug-fix code changes. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 343–347, February 2014.

[Sch14] Niko Schwarz. *Scaleable Code Clone Detection*. PhD thesis, University of Bern, February 2014.

[SLN14a] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Mining the ecosystem to improve type inference for dynamically typed languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM.

[SLN14b] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Overthrowing the tyranny of alphabetical ordering in documentation systems. In *2014 IEEE International Conference on Software Maintenance and Evolution (ERA Track)*, pages 511–515, September 2014.

[SLN14c] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Towards faster method search through static ecosystem analysis. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW '14, pages 11:1–11:6, New York, NY, USA, August 2014. ACM.

2.3 Detailed Research Plan

Each of the four tracks of this new project continues logically from one of the tracks of the predecessor project:

— *Agile Model Extraction* builds on the approximate parsing technology developed in the Agile Modeling track of the Agile Software Assessment project and seeks to develop a practical approach to extracting models from different classes of programming languages;

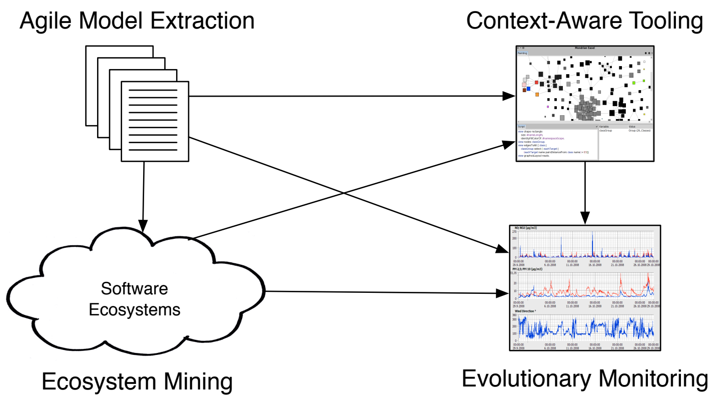
— *Context-Aware Tooling* builds on the "moldable tools" developed in the Meta-Tooling track and seeks to generalise these ideas to source code itself and to the IDE;

— *Ecosystem Mining* continues from Large-Scale Software Analysis to explore other kinds of information useful for the developer that can be extracted from the software ecosystem; and

— *Evolutionary Monitoring* broadens the scope of Architectural Monitoring to consider not just software architecture, but also stakeholder needs, technical debt in general, and technical trends in the ecosystem.

As in the previous project, each track is designed as a separate PhD topic, with plenty of opportunities for synergy and collaboration, but without any critical dependencies. This model of loose collaboration has been very successful in all of the previous SNF projects in this series.

The following figure illustrates the main synergies between the various tracks. Agile Model Extraction provides extraction services to the other tracks and can benefit from common case studies. Ecosystem Mining offers complementary information to Context-Aware Tooling and Evolutionary Monitoring, and Context-Aware Tooling provides infrastructure to Evolutionary Monitoring.



Two of the PhD students working on the previous project will complete their theses in the first year of the new project, with new PhD students picking up the continuation. As before, we will also enlist the participation of Bachelors and Masters students to explore new avenues of research and to reduce the risk for the PhD students.

As with our previous work, evaluation of all tracks and subtracks is an ongoing activity, so no separate "evaluation" activities have been listed. Case studies with available software, or empirical studies with developers and users, as appropriate, will be carried out in each of the tracks.

We plan to continue to integrate the tools that will have reached a certain level of maturity into the Pharo development environment[6] and the Moose analysis platform (as we did with PetitParser, the Moldable Debugger, and the Moldable Inspector in the current project).

Numerous collaborations with external partners in terms of research exchanges, joint development of the research platforms, and joint publications are planned:

— *Dr. Mircea Lungu*, the co-investigator on the previous project, has accepted a tenure-track position as Assistant Professor at the University of Groningen. We plan to continue to collaborate on the topic of Ecosystem Mining.

— *Dr. Tudor Girba* (CompuGroup Medical Schweiz) leads the development of the Moose analysis platform. We plan to continue to collaborate closely on the further development of Moose and specifically on the topic of Context-Aware Tooling.

— *Prof. Alexandre Bergel* leads the PLEIAD (Programming Languages and Environments for Intelligent, Adaptable and Distributed systems) group at the University of Chile. We plan to continue to collaborate through research visits and joint publications especially in the topic of Context-Aware Tooling.

— *Prof. Michele Lanza* leads the REVEAL (Reverse Engineering, Visualization, and Evolution Analysis Lab) at the University of Lugano. We plan to continue to collaborate through research exchanges especially in the topics of Context-Aware Tooling and Evolutionary Monitoring.

— *Dr. Stéphane Ducasse* (INRIA Lille) leads the development of the Pharo Smalltalk development environment. We will continue to collaborate in the evolution of this platform, and we plan research visits and student exchanges related to all tracks of this project.

2.3.1 Agile Model Extraction

In order to effectively analyse a software system or to pose queries over it, one must first transform the source code of the system into a suitable data structure, or *model.* Unfortunately, developing an importer that will carry out this transformation is, in general, a non-trivial task requiring days, weeks or even months of effort, even if a formal specification of the host language is available. We consequently pose the following research question for this track:

RQ 1. How can we rapidly extract models from unknown source code?

Despite considerable progress in recent years, this research question remains open.

[6] `http://pharo.org`

We propose to tackle this problem from three perspectives, each building on previous work.

— **Language recognition heuristics.** We plan to continue to explore the use of heuristics to recognise common language elements, implicit structure, and language classes.

— **Approximate Parsing.** We will experiment with machine learning and evolutionary algorithms to iteratively improve approximate parsers based on our *bounded seas* approach to island parsing.

— **Approximate Parsing by Example.** Here we will combine approximate parsing with language recognition heuristics to our "parsing by example" approach and apply it to a large variety of different kinds of case studies.

Language recognition heuristics. There are many areas where heuristics may be effective either as an alternative to conventional parsing, or as an engine to produce a parser.

We plan to investigate further heuristics to distinguish keywords from identifiers, for example, the use of non-technical terms associated with application domains, natural grammar class of words (nouns, verbs *etc.*).

Many structural cues exist in program text, such as indentation, various kinds of parentheses, braces and brackets, and common keywords like `end`. We plan to explore how such cues can be used either in lieu of a conventional parser, or as an aid to detect potential grammar rules, for example, keywords frequently occurring at a given level in the hierarchical structure.

Most programming languages can be grouped into syntactically or semantically similar classes (*e.g.*, C-like languages; objected-oriented languages). We have experimented in the past with genetic algorithms to recover grammars, but with limited success since we did not exploit language similarities. We plan to use heuristics to identify language classes, and then apply genetic algorithms to mutate composable parsers for constructs unique to those classes. We especially plan to target language dialects, and languages with embedded domain-specific code or proprietary scripting languages to which we have access. The syntactic rules for comments, strings, escape sequences and even tokens are very similar in many programming languages, but they always differ in some small but significant ways. We expect that genetic algorithms applied at this level of abstraction hold better hope of success.

Approximate Parsing. The central insight of our approach is that we do not require a precise parser to initially build a useful model. Many useful analyses, such as establishing package dependencies or system complexity, can start with coarse information about a system. Approximate parsing techniques, such as island parsing with bounded seas (see section 2.2), can be used to recognise parts of the source code while ignoring others.

We plan to further experiment with the application of bounded sea parsers to various kinds of languages, to assess where island parsing works well, and where new techniques are needed.

Model extraction can be understood as transforming unstructured text into hierarchical data structures (trees). Instead of generating detailed models in a single transformation, this can be done iteratively, populating the tree as more structure is recognised.

We have been working extensively with composable parsers (mainly PEGs). We wish to experiment with the composition of conventional parsers (*i.e.*, for islands) and heuristics-based parsers (*i.e.*, for seas). For example, comments might be scavenged for domain knowledge in this way.

Approximate Parsing by Example. As our island parsing technology becomes more mature and robust, we plan to experiment again with and refine our parsing by example approach: use input from experts together with heuristics to generate an initial model extractor; then iteratively refine the extractor as counterexamples that cannot be recognised are identified.

This work will rely heavily on case studies, not only for evaluating the proposed model extraction techniques, but also to drive their development.

In addition to studying classical general-purpose languages, we will investigate various special-purpose languages. Modern software systems are typically implemented in multiple programming languages, which poses a particular challenge for model extraction. In particular, it is becoming very common for host language source code to contain embedded code written in a domain specific language. Extracting models from such a code base will only be possible with the help of composable parsers.

An important part of any code base consists of files that are not strictly executable, such as data files and configuration files. Such files often encode domain concepts as well as business logic. We will also use such examples as case studies.

This track is designed for a single PhD student. We also plan to involve Masters and Bachelors students in individual subprojects. The PhD student working on agile modeling in the predecessor project, Jan Kurš, will be completing his dissertation during the first year of the new project and will help to mentor the new student.

2.3.2 Context-Aware Tooling

A great deal of software development time is not spent producing new code, but in reading and trying to understand existing code. Unfortunately commonly available development environments focus mainly on supporting programming language features, while offering little or no support for the kinds of questions developers have concerning the system architecture, the domain concepts as represented in the code, or other aspects that are project-specific. We therefore pose the following research question in this track:

RQ 2. How do we close the abstraction gap between application domains and IDEs?

We propose to explore the following interrelated themes:

— **Meta-Tooling.** This *end of thesis* project will continue to explore how to support the rapid production of custom tools to support specific development tasks.

— **Context-Aware Code.** Here we explore how to enrich source code with project-specific information, such as mappings to domain concepts, to enhance the way developers see and explore software systems.

— **Context-Aware IDE.** Given the enriched model of software source code, here we explore new ways to support domain-specific queries and assessment tasks from the IDE.

Meta-Tooling. This activity continues and completes the "meta-tooling" (*i.e.*, tools for building tools) track of the predecessor project, which focused on the rapid development of custom developer tools, such as domain-specific debuggers, browsers and inspectors. Here we will focus on elaborating the specific mechanisms and techniques that enable domain-specific inspectors, debuggers, searches, *etc.*

We have also been developing several new tools and platforms to support meta-tooling. The *Spotter* is an experimental interface for navigating through a space of domain-related objects. Spotter is currently in Pharo and we are using it to understand how developers search in the IDE. We plan to experiment with better support for domain-specific querying and navigation by allowing each object to define multiple types of searches, and by providing a UI through which one can build searches on top of previous searches.

Phlow is a reimplementation of Glamour, the domain-specific browser framework, which relies on first class representations of attributes to propagate events between different views of the model being browsed.

These two currently ongoing activities will constitute an important component of the PhD thesis of Andrei Chiş to be defended during 2016.

Context-Aware Code. Classical IDEs treat source code essentially as structured text, and only offer limited means to navigate between related software entities. As a first step to better support developers in understanding and analysing code, we propose to enrich source code with contextual information that is motivated by established developer needs.

We envision an environment in which code is *live* and interconnected not only with other pieces of code but also a variety of other relevant software artifacts. In the software development process, programmers typically gather, modify and develop various snippets of code, documentation, knowledge (*i.e.*, questions and answers) from various sources. Initial experiments with an early prototype suggest that a tool that manages trees of linked snippets can be useful for both development and program comprehension.

Building on our experience with meta-tooling and with high-level specification of model visualization, we also plan to explore how to map software artifacts, both at the lowest level of snippets, and at high levels, to suitable visualizations corresponding to domain concepts or architectural components and constraints. (For example, in PPBrowser, we provide suitable graph-based visualizations of grammar rules.)

To drive this work, we plan not only to exploit previous research into developer needs (subsection 2.1), but we also plan to observe developers working in selected application domains, such as web development and mobile application development, to track the questions they typically pose.

Context-Aware IDE. Building on the notions of domain-aware code and meta-tooling, we plan to explore how developer questions can be supported by high-level querying tools and custom-built assessment tools. A proof-of-concept already exists in the PPBrowser tool, which offers dedicated support for browsing, developing, inspecting and debugging parsers.

Particular research challenges here include: (i) realizing a queryable metamodel for federated software information, and (ii) offering an efficient platform for developing and sharing simple assessment tools.

The Snippets model of live code outlined above should be based on an extensible metamodel that can integrate information from a large variety of sources: source code itself, domain models, architectural models and constraints, developer history, version history, documentation, developer questions and answers, and further information from the software ecosystem. We envision an open metamodel that supports dynamic and ad hoc extensions.

Querying and browsing can be seen as complementary activities. A query posed in a query language formally specifies how one navigates data conforming to a metamodel (*i.e.*, a schema) to arrive at query results. Similarly, a tool for browsing a model navigates according to the model's metamodel. We plan to exploit this synergy to develop a browsing and querying tool that allows us to store queries corresponding to an interactive developer browsing session and replay them at a later stage. Such queries can be stored as snippets together with the developer questions they are meant to answer.

The results of queries can be presented or visualized in a variety of ways. Many custom assessment tools can be seen as queries combined with a suitable presentation. We plan to build on our previous work on software visualization to develop a high-level language for specifying such tools as combinations for queries and visualizations. Tool descriptions can then themselves be stored and shared as snippets. We intend to explore the potential for enabling tool discovery by matching such descriptions to developer needs.

2.3.3 Ecosystem Mining

The chief advantages of systems depending on each other in an ecosystem is the possibility of reuse and the fact that together they are more than the sum of the parts. In this track we go beyond these conspicuous advantages, and further investigate the possibilities of leveraging the ecosystem context to provide intelligence[7] during the development process. The overarching research question that we will tackle is:

RQ 3. How can we mine the ecosystem to automatically discover intelligence relevant to a given project?

We plan to explore several paths in this direction:

— **Reuse Opportunities.** This *end-of-thesis* subtrack will explore how to discover opportunities for code reuse from the ecosystem.

— **Ecosystem-Aware Documentation.** Here we investigate mining of artifacts from the ecosystem to provide contextual and evolving documentation.

— **Intelligence for the Upstream.** In this subtrack we explore how recurring structures mined from downstream (*i.e.*, client) projects can provide critical information to the upstream (*i.e.*, libraries and frameworks).

[7]We use intelligence here in the sense of actionable information that supports decision-making

Reuse Opportunities. Sometimes shortcomings of a particular upstream require multiple clients to extend it in the same way (*e.g.*, in Javascript, literally thousands of projects and developers are forced to extend Array.prototype with the same methods). We plan to investigate automated techniques that can increase the awareness of such reuse opportunities *in-the-small* by detecting possible type extensions and automatically suggesting them to the developer at code-modification time. We will target multiple possible types of extensions such as subclasses, class extensions, prototype extensions and plugins. We will adapt existing code clone detection strategies to help detect equivalent extensions, and thus enable computing the popularity of different extensions.

One particular type of reuse that goes beyond source code reuse is the reuse of actual live objects. We have already developed a prototype of an "object repository" to mine, store and manage code snippets that create object instances. We plan to explore the potential applications of such an object repository that include easily generating test fixtures and being able to execute individual methods from the source code without running the entire system.

Ecosystem-Aware Documentation. In previous work we have shown that the usability of both code and documentation browsers can be increased by augmenting them with information distilled from the ecosystem, such as the relative popularity of different APIs. We plan to extend the current lightweight approach with a better contextualization of the analysis. Several ways in which the documentation augmentation can be contextualized are by taking into account: (1) the particular upstream configuration of the system in which the developer is working at the moment, (2) the experience of the developer, (3) the actual versions of the upstream, and (4) evolutionary trends in the ecosystem, for example, concerning the usage of a given API.

API usage examples and code snippets are some of the most sought after pieces of information in online fora like the currently popular StackOverflow. We plan to explore ways in which the peer-stream can be valorized for detecting API examples and relevant code snippets and integrating these examples in the documentation browser. In particular we plan to investigate the possibilities of improving the documentation and the browsing experience for dynamically typed languages, an area which has been traditionally underdeveloped.

We expect to exploit the synergy with the Context-Aware Tooling subtracks on domain-aware code and domain-aware IDEs to enable individual developers to effortlessly share the code snippets they consider have reusable value.

Intelligence for the Upstream. There is a strong agreement between upstream developers that (1) they desire to improve the usability of their code, but (2) they lack systematic techniques and mechanisms to discover the way the downstream uses their code. We plan to investigate opportunities for improving the intelligence gleaned from the downstream by automatically extracting and reporting usage statistics and emerging patterns in the downstream systems. Especially interesting are three types of information:

1. *Usage information from the downstream*, will provide the upstream with information about which parts of the code are more frequently used, and thus more critical for the clients.
2. *Repetitive structures, and boilerplate code* will provide feedback to the upstream on opportunities for increasing the expressiveness, understandability, and stability of their code.

3. *Recurring defect patterns*, detected through the analysis of downstream bug fixes, can automatically suggest which parts of a given APIs are problematic.

To detect such patterns from the downstream we plan to build on our previously developed code cloning techniques, and combine dependency information as it is specified in configuration files, with details extracted by static analysis from the source code of the corresponding downstream systems. Moreover, we plan to integrate these techniques with the existing monitoring infrastructure used currently to regularly rebuild our Ecosystem Aware Type Inference database. As a validation for this track we plan to implement prototypes that we will test with developers.

This track is designed as a combined end-of-thesis and handover between Boris Spasojević, working on the corresponding track in the predecessor project and a new PhD student.

2.3.4 Evolutionary Monitoring

As a software system evolves, technical debt typically accrues. Existing tools for monitoring the health of a software system generally focus on relatively low-level or syntactic aspects such as dependency violations or the list of open issues. We posit that an effective monitoring system should take into account the *context* of a specific project and its ecosystem to help developers and managers take informed decisions about where and when technical debt should be paid off. We accordingly pose the following research question:

RQ 4. How can we steer the evolution of a software system by monitoring stakeholder needs, technical debt, and ecosystem trends?

— **Tracking Stakeholder Needs.** Here we ask how we can effectively track developer needs and user needs by augmenting the IDE or the application software itself. What questions are stakeholders asking? What parts of the system are chronic "pain points"? Who has the needed expertise? How do these aspects evolve over time?

— **Tracking Technical Debt.** Technical debt concerns not only architectural drift but any form of software ageing that requires investment to repair, such as reliance on out-of-date libraries, code bloat, or obsolete documentation. In this subtrack we develop methods to *identify* different forms of technical debt, to *track* them as the system evolves, and to *assess* them in the context of stakeholder needs and priorities.

— **Monitoring Technical Trends.** Ecosystem trends form an important part of the context of technical debt. Are there important changes to upstream resources (*e.g.*, libraries) that must be considered? How much will it cost to integrate them?

Tracking Stakeholder Needs. A crucial part of the context of technical debt of a software system is formed by the needs of the various stakeholders of that system, most particularly the developers and the users. User needs are conventionally considered during requirements collection and analysis and in usability studies, but not during normal usage. Developer needs have scarcely been considered outside of specific research studies, and are best supported by dedicated fora, rather than through IDE support.

We plan to explore how stakeholder needs can be effectively mined directly from the IDE and the application. It is important that such observation be non-intrusive, transparent and respect privacy. We see two important sources of information to be tracked. The first is the set of questions posed by the stakeholder and the answers obtained. The second is the trace of interactions with the software or with the IDE. The first source can be used to identify experience, expertise and thematic pain points. The second can be used to infer application features that are problematic, missing, or unused, and to infer possible pain points in the development process.

To enable such tracking, applications and IDE need to be augmented with features that are typically considered out-of-scope, such as an integrated forum offering the ability to pose questions directly from with the system, or to browse past questions and answers. Tracking user or developer actions can be a rich source of useful information, but it must be optional, and only with the consent of the user or developer.

Both kinds of information can provide useful feedback to the user/developer (where have you been spending most of your time? which features have you not used yet?) as well as to the project management in general (what are the evident pain points?).

Tracking Technical Debt. There are many different categories of technical debt, which may be more or less important at a given stage in a project, for example, architectural drift, general code quality (*e.g.*, incidence of duplicated code, commenting quality, readability), incidence of defects, performance shortcomings, reliance on aging infrastructure or platform, obsolete documentation, and so on. Locating these different forms of technical debt similarly entails monitoring of different sources of software artifacts, such as the source version repository, the issue tracker, and the project management database.

To identify different forms of technical debt, we plan to explore a number of technical areas: benchmarks and detection strategies to locate technical debt; metrics to assess potential profit and loss; exploitation of historical data from the same project or the ecosystem; simulations to assess the benefits of different investment (repair) strategies. Here we expect some synergy with the Ecosystem Mining track in mining ecosystem data concerning typical costs of upgrades.

We envision a form of customisable "developer dashboard" to monitor different forms of technical debt. In addition to tracking the state of the system, it should be possible to quantify the technical profit and loss associated with technical debt.

To assess technical debt, one must evaluate it in the specific context of a given project and the stakeholders' needs. For a given system component, it is important to not only quantify its technical debt (how much will it cost to repair or replace it), but also the cost of not repairing it (what maintenance costs does it accrue?).

Monitoring Technical Trends. Suppose open source Platform X is an integral part of our development project. We would like to know: Where has most of the development activity concentrated in Platform X over the past year? How healthy are other projects based on Platform X? How difficult is it to migrate to version 3.0 of Platform X? How many projects are abandoning Platform X for System Y?

This kind of information is not always easy to obtain, but it can be discovered by data mining a variety of sources, such as the commit histories of open source repositories of the infrastructure itself and its downstream ecosystem, Q&A fora, and selected social media. We envision some close collaboration with the Ecosystem Mining track.

Here we envision a variety of streams of data that can feed into the "developer dashboard" and plug into metrics and visualization widgets to enable the user to observe trends. The selection of displayed feeds could be manually configured, or it could be automatically prioritised based on past interest or the tracked developer needs.

2.4 Schedule and milestones

Here we provide a coarse timeline for each of the planned research tracks. Each row corresponds roughly to one person-year of a PhD project. Please note that P2 = Andrei Chiş and P4 = Boris Spasojević are end-of-thesis projects, followed respectively by P3 and P5, for a total of 4 PhD tracks or 12 person-years.

Year 1	
Agile Model Extraction	Language recognition heuristics (P1)
Context-Aware Tooling	Meta-Tooling/ Context-Aware Code (P2/P3)
Ecosystem Mining	Reuse Opportunities (P4)
Evolutionary Monitoring	Tracking Stakeholder Needs (P6)
Year 2	
Agile Model Extraction	Approximate Parsing (P1)
Context-Aware Tooling	Context-Aware Code (P3)
Ecosystem Mining	Ecosystem-Aware Documentation (P5)
Evolutionary Monitoring	Tracking Technical Debt (P6)
Year 3	
Agile Model Extraction	Approximate Parsing by Example (P1)
Context-Aware Tooling	Context-Aware IDE (P3)
Ecosystem Mining	Intelligence for the Upstream (P5)
Evolutionary Monitoring	Monitoring Technical Trends (P6)

2.5 Importance and impact

The results of this project (as with previous projects) will be disseminated primarily through peer-reviewed full papers in top-ranked international conferences, such as ICSE (International Conference on Software Engineering), ICSME (International Conference on Software Maintenance and Evolution), SANER (International Conference on Software Analysis, Evolution, and Reengineering), ICPC (International Conference on Program Comprehension), OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) and ECOOP (European Conference on Object-Oriented Programming), and in top international journals such as IEEE TSE (Transactions on Software Engineering) and ACM TOSEM (Transactions on Software Engineering and Methodology).

We also collaborate regularly with industrial partners to apply our techniques in extended case studies, or to carry out empirical studies (*i.e.*, usability studies, interviews and surveys). In many cases the results of our research take the form not only of academic papers, but also tools or platforms (such as Moose and Pharo, both of which started as internal SCG projects) that have a considerable academic and industrial user and contributor base. Many of the tools we build are are used in teaching at universities worldwide.