Agile Software Assessment (Invited Paper)

Oscar Nierstrasz
Software Composition Group
University of Bern
Switzerland
http://scg.unibe.ch/oscar

Mircea Lungu
Software Composition Group
University of Bern
Switzerland
http://scg.unibe.ch/staff/mircea

***Abstract*—Informed decision making is a critical activity in software development, but it is poorly supported by common development environments, which focus mainly on low-level programming tasks. We posit the need for *agile software assessment*, which aims to support decision making by enabling rapid and effective construction of software models and custom analyses. Agile software assessment entails gathering and exploiting the broader context of software information related to the system at hand as well as the ecosystem of related projects, and beyond to include "big software data". Finally, informed decision making entails continuous assessment by monitoring the evolving system and its architecture. We identify several key research challenges in supporting agile software assessment by focusing on customization, context and continuous assessment.**

PREPRINT: In Proceedings of International Conference on Program Comprehension (ICPC 2011), p. 310, 2011. DOI:10.1109/ICPC.2012.6240507

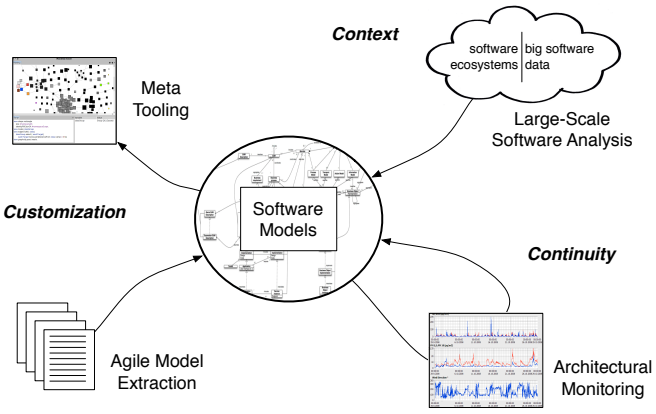

Fig. 1. Agile software assessment entails customization, context and continuity

I. Introduction

Developers are under constant pressure to assess the state of the system at hand in a timely fashion in order to carry out development and evolution tasks. As a consequence, they often spend as much time reading, understanding and assessing code as they do writing new code [18]. Unfortunately, mainstream integrated development environments (IDEs) focus on low-level programming tasks rather than on supporting program comprehension and decision making during development. Furthermore, the analysis tools that are available have a narrow scope of applicability.

Decision making in any context entails the ability to gather and process relevant information. In the context of software development, this means that we need to model not only the source code of the system under development, but also any other information that can be relevant to the development tasks at hand. In particular, this may include documentation, domain models, bug reports, architectural specifications, runtime traces, and version histories. Furthermore, not only information about the system under development may be relevant, but also that of the ecosystem of related systems. Even analysis of other systems that have been developed may be useful to the decision making process.

In this paper we call for the discipline of *agile software assessment* — the study of the tools and techniques that will allow developers to integrate analysis tools into the daily workflow of software development.

We identify three aspects that must be addressed to support agile software assessment.

1) Customization is needed to allow flexibility in building analyses that match the particular and ever changing problems of the development team.
2) Context must be captured and analyzed to enable a broader understanding of the local phenomena.
3) Continuity of the analysis implies its integration in the forward engineering process and is needed to support decision making regarding software evolution.

Let us briefly consider these three aspects.

Customization: A key bottleneck to effective software assessment is the rigidity of analysis tools which impedes their adaptation to the specific context of the team. Developers ask detailed and domain-specific questions about the software systems they are developing and maintaining. Specialized analyses are needed to effectively answer these questions, but the only tool that developers regularly use is the IDE. However, most IDEs do not provide suitable functionality to answer detailed questions about the design and implementation of software, as they focus on programming rather than software modeling. Research into *meta-tooling* promises to enable developers to rapidly produce custom analyses from available building blocks to support custom analyses.

One particular challenge for software analyses is the rapid construction of appropriate software models from program

source code and other associated data sources. Custom solutions are required since (i) the data sources are commonly heterogeneous (even the program source code often contains a mix of different languages [30]), and (ii) the target models need to be extensible to support various kinds of analyses [28]. We refer to this as *agile model extraction*.

Context: Complex software systems generally exist within an even larger software ecosystem consisting of older versions of the systems, variants, and other client applications of the system or its parts. Being able to query and mine this large resource of information can be critical for some analyses and helpful for others. Given the large and growing amounts of software available, there is an increasing need to model and query large software data sets, including historical information and variants. We refer to this as *large-scale software analysis*.

Continuity: Software assessment must be *continuous* as in *continuous integration*: it needs to become part of the forward engineering workflow. Monitoring the evolution of trends and relationships that involve source code and associated artifacts will allow early detection of problems and opportunities, and will enable small adjustments of the system's evolution.

Two aspects that must be continuously monitored are: (1) the evolution of a system's architecture and (2) the evolution of the ecosystem to which the system belongs.

The *architecture* of a software system consists of the design constraints that guarantee non-functional properties, such as ease of evolution, good run-time performance, and rapid build times. Unfortunately architecture is rarely explicit in code, hence it must be recovered and tracked, sometimes at great cost in developer time and effort. As the system evolves, its architecture also changes and drifts. To support informed decision making, developers need to track these changes to observe trends and to detect potential problems. We therefore need ways to mine, specify and track the evolution of software architecture and other implicit aspects of software systems (such as correlation between components and features, code stability, propensity for errors, *etc.*). We refer to this as *architectural monitoring*.

Structure of the paper: In the following sections, we will explore each of these aspects and identify what we interpret as the key research challenges. In section II we explore customization in the context of agile model extraction and meta-tooling. In section III we consider the challenges of large-scale software analysis to exploit the broader context of software ecosystems and "big software data" to support agile software assessment. In section IV we look at how architectural monitoring can support continuous assessment of evolving systems. We conclude in section V with some remarks about our ongoing and future research.

II. Customization

Goal: "Build the model in the morning, analyze it in the afternoon."

Informed decision making presumes the ability to efficiently and effectively construct software models that enable custom analyses. The questions developers ask are many and varied [3], [9], such as: "Where is this piece of code referenced?" "Are there systems that I will break if I remove this function?" "Who worked on this code before, and why is it designed like this?" "What code supports this user feature?" "How will this change impact the architecture of the system?"

Customization is a key requirement to building the needed models and analyses. In this section, we will identify some of the research challenges in these two areas.

A. Challenge: Meta-tooling

Developers devote considerable time to reading, understanding and assessing code to effectively support decision making in the development process. They are often forced to recover implicit knowledge by exploring code and discussing with teammates [18]. Often developers have very particular requirements that can't be addressed with generic tools. If they can't quickly customize the appropriate tool or analysis they will eventually not do it. By *meta-tooling*, we refer to tools to rapidly and effectively build custom software assessment tools.

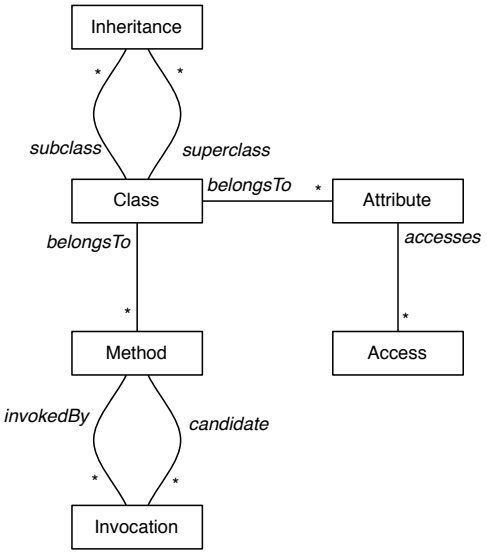

Fig. 2. FAMIX core metamodel (extract)

Direction — Integrated Platform for Custom Software Analyses: Most IDEs do not easily accommodate new functionality. An environment like Eclipse, for example, allows users to rearrange and customize the individual windows of the environment, and new tools can be downloaded and installed as "plug-ins", but developing a new plug-in is highly non-trivial. We envision, by contrast, a "malleable" IDE which can be easily extended with new software assessment tools composed from the more basic mechanisms described above.

One example of a meta-tooling environment is Moose, a platform for analyzing software and data [28]. Several of the components that can be combined easily to build new tools are:

- *FAMIX*, is a meta-model for modeling object-oriented systems (see Figure 2). It enables customization through

extensions that express new concepts. Examples include modeling version histories [10], semantic clustering of classes and packages by lexical content [16], and feature views that track which software artifacts support which user features [11].

- *Mondrian* [24] (Figure 3) is a visualization engine that makes it easy to generate a wide range of visualizations: Figure 3 shows an example. Since its host language, Smalltalk, has a very simple syntax, composing new visualizations resembles writing scripts in a DSL. A lightweight, interactive UI allows developers to immediately see the result of changes to their Mondrian scripts. Mondrian supports agile software assessment by enabling developers to generate new visualizations in minutes instead of days.

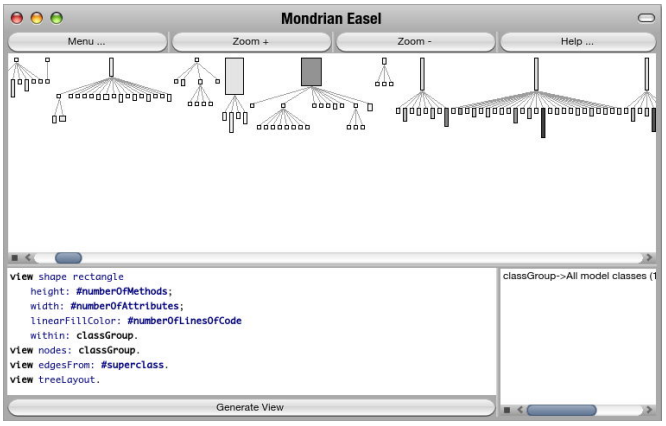

Fig. 3. Mondrian visualization engine displaying a scripted System Complexity view

- *Glamour* [5] offers a generic engine to construct custom browsers for arbitrary models (not just source code). A browser is built up from a number of components, each of which renders some fragment of a model, and *transmits* information about that fragment to another connected component. A classical source code browser with panels displaying packages, classes, methods and source code can be constructed as a compact Glamour script. Dedicated browsers for other kinds of models can similarly be scripted with low effort.

These examples illustrate the benefits of meta-tooling, but they only constitute some small steps towards an expressive meta-tooling environment to support agile software assessment.

Instead of integrating a fixed set of software analysis and assessment tools into the IDE, we propose to focus instead on integrating a set of key software assessment mechanisms, and offering a meta-tooling framework that allows these mechanisms to be easily configured and combined to address a particular software assessment task, such as identifying error-prone code, locating components that support user features, or identifying the impact of a proposed change.

B. Challenge: Agile Modeling

The key prerequisite to analysis is fact extraction. When faced with a new language, a variation of a language, or a domain specific language, building a fact extractor can be very time consuming [4].

With "agile modeling" we refer to the challenge of automatically constructing software models from source code and other data sources. Since building a full parser by hand for a general-purpose programming language is time-consuming and expensive, there is a need for techniques to streamline this process [17].

What is missing is a unifying framework to enable rapid and iterative construction of software models suitable for analysis from multiple program and data sources.

The following characteristics of the problem lead us to believe this goal is reachable:

- Most programming languages can be classified into groups with similar features. Using appropriate techniques, one can decompose parsers into parts that resemble one another.
- Complete parsers are not needed for software modeling. In an initial phase, it may be enough to capture coarse structure. A grammar can be refined as more details are needed.
- Typically, a large base of existing code samples are available for analysis. This fact can be exploited to automatically generate and test parsers.

Direction — Guided, Iterative and Incremental Fact Extraction: We believe the following techniques will be useful to support agile modeling:

- Guided parser refinement. In software modeling it is typically not necessary to completely parse all parts of the source code in order to perform many kinds of analyses. Complete parsers are not needed for initial software analysis. This suggests an approach in which an initial grammar is defined as a coarse island grammar [27] that can be iteratively refined by the software modeler. The refinement process will be guided by the user, for example, by highlighting sample code fragments and specifying their mapping to model elements (*i.e.*, classes, methods, statements *etc.*).
- Reusable parser fragments. PEGs (Parsing Expression Grammars) support the definition of parsers from composable parts [8]. Scannerless parsers furthermore avoid the need for a separate lexical pass, and avoid the need to commit to lexemes of a given language [35]. By composing scannerless parsers for different languages, one can cope with heterogeneous code bases using multiple embedded languages.
- Structural inference. While user intervention will be needed to guide the parser refinement process, automation can help to provide the user with hints on how to proceed. A few promising avenues are:
  - Exploiting indentation to identify structural elements to model [12].

- Analyzing recurring names to distinguish potential keywords from identifiers.
- Analyzing source code text to identify comment conventions.
- Exploiting language similarities by generating island grammars for common language fragment parsers.
- Mutating parser fragments [37] or instantiating them based on source code analysis.

Finally, the exploration of these techniques can be extended to structured data related to the development lifecycle, such as profiling data and bug reports [23].

III. Context

Goal: "Exploit massive amounts of source code to build better tools and improve software quality in the organization."

There exists a broad corpus of research in mining software repositories, but it is usually concerned with mining versioning repositories of individual systems to detect problems with the source code [34], [36], mining a single developer's interactions with his IDE to improve the IDE [13], [31], or to predict bugs [6]. However, recently there has been a surge of interest in pushing software analysis beyond the level of individual systems, one of the reasons being the new availability of data. Indeed software is entering the age of big data, which is characterized by increasing volume (amount of software), velocity (speed of software generation), and variety (range of data sources).

There are two interesting directions to pursue in the context of analyzing large numbers of projects: ecosystem analysis and big software data. Ecosystem analysis focuses on solving the inherent problems existent when multiple inter-related software systems co-evolve independently. Big software data treats information at lower abstraction levels such as files, API calls, and lines of source code, and on this data it applies data analysis and mining techniques.

A. Challenge: Ecosystem Analysis

Information needed to support effective decision making during development can be found outside of the source code of the system. The software architecture, domain knowledge, user features, developer knowledge, and so on, are classic examples.

One special type of context that we forecast will gain preeminence in the analysis is the *software ecosystem* — the collection of software systems that are developed with common technology, co-evolve in the same environment [20] and are inter-dependent. The environment can be an organization (*e.g.*, large banks are known to host hundreds of inter-dependent systems), an open source community (*e.g.*, the Apache software foundation), or a technology (*e.g.*, Android). A number of problems that are relevant for individual system analysis remain relevant at the ecosystem level. The importance of some of these problems is even augmented. And some of these problems can be better solved if the ecosystem is taken into account.

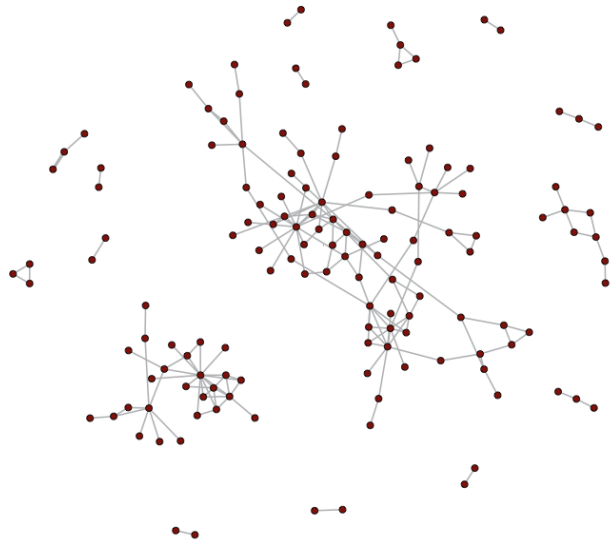

Fig. 4. A subset of systems in the SqueakSource ecosystem show a tight network of compile-time dependencies

A large number of analyses can benefit from taking the ecosystem context of a system into account: computing thresholds for metrics, detecting code duplication, understanding the impact of a change in a system on the other systems that depend on it. We believe it is critical to devise techniques and infrastructures to model an evolving ecosystem to allow the fast access and querying of the data at the ecosystem level.

Moreover, studying and understanding the evolution of the ecosystem as a whole can benefit the analysis of the individual systems. Figure 4 shows a subset of the systems in the SqueakSource ecosystem that we have studied [22]: the tight interconnection indicates a need for ecosystem analysis.

Direction — Modeling Ecosystem Evolution: The evolving source code for all the systems in an ecosystem can be massive. If analysis tools are to be augmented with the capacity to take into account the ecosystem, they must also scale up to model its evolution. At the moment, most of the analyses that take into account large numbers of systems consider only snapshots without modeling their evolution [2]. A few solutions that do take into account the evolution either do not scale well enough [20], or they capture a very lightweight evolutionary model [32]. The infrastructure should be tailored for the particularities of evolving software:

- Source code evolves *forward* in time. Once a version is committed to the versioning repository, it does not change anymore. Every structural artifact in the software (*e.g.*, method, class, package) once published, is frozen. This suggests that data warehousing approaches may be applied.
- Source code duplication is common, from micro-duplication patterns, and boilerplate code, to product families and branches in the versioning system. We can

use this information to minimize the analysis effort, and optimize storage (*e.g.*, hashing files by their content eliminates the need for storing duplicates)
- Software data is a combination of highly structured and unstructured data. The highly structured data are the source files and the unstructured are the associated artifacts (*e.g.*, documentation, emails from the mailing lists). One can extract a large number of relationships between the different artifacts of the source code (*e.g.*, containment, calling, *etc.*).

B. Challenge: Big Software Data Analysis

The scope of big software data analysis can be broader than the ecosystem and include all the source code that is available online in a given hosting site (*e.g.*, GitHub) or even all the software that is available online written in a given programming language. Based on such analysis one can achieve a broad variety of goals from improving the IDE to improving the programming languages themselves.

The challenge when analyzing such data is realizing the infrastructure that would balance the trade-offs between providing a simple representation of the data and optimizing for a particular task. Mockus reported on his experience in amassing a very large index of version control systems and the associated challenges: the data from the SourceForge CVS occupies more than 1TB; extracting large amounts of files from the Mercurial database took more than one month; using standard database techniques that would work for the code in a large corporation would lead to processing times of many months, *etc.* [26]. Therefore, in the realm of big software data, an intelligent infrastructure can bring large savings in processing and retrieval time.

Direction — Embedding Intelligence and Adaptation in the IDE: Modern IDEs are rigid pieces of software: with very few exceptions, they do not learn from their usage, and when they learn, they learn from the local interactions. An IDE that would learn from the interactions of numerous users could be improved in various ways:

- Suggesting code completion and code navigation based on recording the IDE interaction data of large numbers of developers
- Suggesting code snippets, and documentation that might be useful for the code that the user is writing at the moment. Given the large amounts of code and the complexity of current libraries (*e.g.*, Sun's Java SDK contains more than 3,500 classes organized in more than 200 packages) this can be very useful.
- Suggesting tests based on existing tests for similar code

Direction — Large-scale Dynamic Analysis: Run-time data of a software system can be a rich source of information that can improve the quality of the system as well as the quality of the development process. Dynamic analysis tools and techniques are conventionally limited to a single run of a single project. There is nothing inherent in these techniques that requires this to be so, aside from a question of scale. By creating a central repository of run time information, we can enable new analyses and improve the precision of existing ones.

The key challenges in this research direction are discovering how to efficiently generate, transmit, and store the dynamic information in a central repository without impacting the performance of the running applications.

IV. Continuous Assessment

Goal: "Continuously monitor the code base to inform development decisions."

A large proportion of analysis tools are designed as use-once approaches. When such a tool reports problems with the system, usually the list is so long that it can paralyze action. Agile Software Assessment implies the integration of analysis tools in the daily development workflow and the continuous reporting of the opportunities for evolution and improvement. This will enable understanding of trends and evolution of the system as it unfolds.

The challenge of continuously monitoring the code base can be considered based on the abstraction level of interest: the architecture or the ecosystem.

A. Challenge: Monitoring Evolution

As the environment of a successful software system evolves, the system too has to evolve, and typically, so must its architecture. This holds also for the other systems that the system interacts with. Monitoring a system's evolution in its context promises to improve system stability, quality and robustness, and to ease the development process.

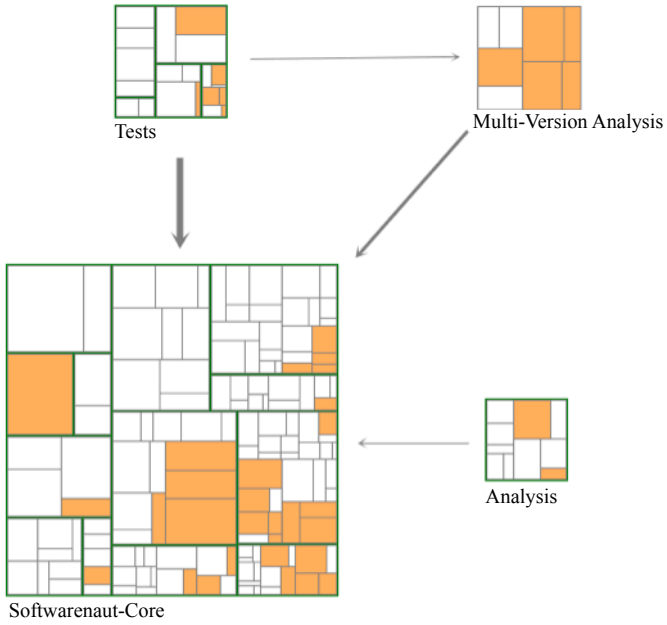

Fig. 5. The Softwarenaut prototype showing an evolutionary view of itself in which the most frequently changed classes are highlighted

Direction — Monitoring Architectural Evolution: One of the difficulties in reasoning about software architecture is that architecture is largely implicit in the source code. Aside from coarse-grain package structure and naming conventions, very few indicators exist in source code that reveal the architecture. This fact has led to two separate research activities, namely work on *specifying* software architectures, and work on *recovering* architecture. A third activity, *monitoring* the evolution of software architecture is in its infancy. One such example is the work of Knodel [15] who proposes *constructive compliance checking*, an approach to provide live feedback to developers when they violate architectural specifications which, authors show in a controlled experiment with students, improves the quality of the resulting system [14].

Shaw and Garlan [33] provide a detailed survey of so-called "architectural description languages" (ADLs) which are intended to augment the source code of a software system with a specification of its architecture. Classical ADLs such as Rapide and Wright defined architecture in terms of components and connectors [19], [1]. More recent work continues in the same tradition: Dashofy *et al.* [7] focused on developing an XML-based ADL that is modular and composable, but at the same time remark that ADLs are not having a significant impact on software engineering practice.

We identify the following research opportunities in monitoring architectural evolution:

- *Agile specification of a system's architecture.* Since ADLs are not having a significant impact on software engineering practice [7], we believe that there is an opportunity for developing techniques to specify architecture in ways that reflect the actual practices and needs of developers.
- *Enforcing the correct evolution of a system.* To prevent the architecture from drifting away from the code, the constraints encoded in the architecture must be enforced during system evolution. This suggests that feedback mechanisms would be useful to inform developers when proposed changes conflict with the system's architecture. When violations to the architecture are inevitable for any reason, mechanisms must be in place to allow the exceptions to be made explicit, so that long term maintenance is facilitated.
- *Presenting the evolving architecture of a system.* Figure 5 presents Softwarenaut — our architecture recovery prototype — visualizing its own macro structure with modules and dependencies and highlighting inside the modules the classes that are the most changed [21]. Similar simple and lightweight visualizations could be augmented to indicate architectural constraints and their violations, and more generally to indicate the co-evolution of the system and its architectural description. An *architectural dashboard* would allow the different stakeholders to monitor the co-evolution of architecture and source code in a system.

Direction — Monitoring Ecosystem Evolution: Developers of a system must be able to query the way their source code is used by other projects and the users of a library should be able to find the usages of that library. Currently there is no open-source infrastructure to support this.

In our discussions with industry we have discovered that this direction is critical: one developer in a large corporation reported that it takes days until he finds out whether his changes to the library that he is maintaining will break other teams' code. Begel *et al.* conducted a study with Microsoft engineers to discover the needs of the developers working in a large corporate ecosystem. They discovered that some of the important needs are related to awareness and impact [3].

In an effort to obtain empirical evidence of the importance of impact analysis at the inter-system level, we recently performed a case study on a large Smalltalk ecosystem. We discovered that changes in libraries and frameworks can have a very large impact in the ecosystem: dozens of projects and developers can be forced to update to new version of a library, but the developers of these libraries and frameworks lack tools which would allow them to predict the impact of a change [32].

Several existing works in this directions are related to automatically extracting dependencies between software systems and increasing the awareness of the developers of the way the other developers in the organization work. Ossher *et al.* started from another developer need that commonly arises in the open-source world, which is to build a newly downloaded artifact. Accordingly, they studied whether they could cross-reference a project's missing types with a repository of candidate artifacts [29]. Codebook is a proprietary approach to modeling the variety of artefacts that are associated with the source code in an ecosystem. Mileva *et al.* [25] studied the evolution of libraries in the apache ecosystem and discovered that the *wisdom of the crowds* can be helpful when deciding which version of a library to use.

V. Conclusion

Developers routinely need to perform custom analyses on software systems under development, but mainstream development tools do not offer the appropriate tools to support them. We accordingly posit the need for *agile software assessment*, that is, tools and techniques to enable the rapid and effective analysis of software systems.

We have identified three key challenges to agile software assessment: (i) *customization*: how to quickly produce software models from program and data sources, and how to build the custom analyses using these models; (ii) *context*: how to exploit the broader context of software ecosystems and "big software data" to support the develop in decision making tasks; and (iii) *continuity*: how to exploit information about the continuous evolution of a system and its context to support the developer.

We have similarly identified a series of promising research directions to meet these challenges, and we encourage researchers to join us in exploring these directions. Some of the key ideas we plan to explore include: the development of a meta-tooling environment to support custom analyses; guided, incremental fact extraction from program and data sources;

the application of "big data" techniques for large-scale software analysis; and continuous monitoring of architectural and ecosystem evolution to detect trends and to keep developers aware of changes that might impact them.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012). We also thank Fabrizio Perin and Andrea Caracciolo for their review of a draft of this paper.

REFERENCES

[1] Robert Allen and David Garlan. The Wright architectural specification language. CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[2] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, ICSE Workshop on*, 0:1–4, 2009. doi:10.1109/SUITE.2009.5070010.

[3] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM. doi:10.1145/1806799.1806821.

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010. doi:10.1145/1646353.1646374.

[5] Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, April 2009.

[6] Marco D'Ambros, Michele Lanza, Mircea Lungu, and Romain Robbes. On porting software visualization tools to the web. *In Journal on Software Tools for Technology Transfer*, 13:181 – 200, 2011. doi:10.1007/s10009-010-0171-9.

[7] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, April 2005. doi:10.1145/1061254.1061258.

[8] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM. doi:10.1145/964001.964011.

[9] 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. doi:10.1145/1806799.1806828.

[10] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[11] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006. doi:10.1002/smr.340.

[12] 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. doi:10.1109/ICPC.2008.13.

[13] 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. doi:10.1145/1052898.1052912.

[14] Jens Knodel. *Sustainable Structures in Software Implementations by Live Compliance Checking*. PhD thesis, Univ. of Kaiserslautern, Computer Science Department, 2011.

[15] Jens Knodel, Dirk Muthig, and Dominik Rost. Constructive architecture compliance checking – an experiment on support by live feedback. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 287–296, 2008.

[16] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007. doi:10.1016/j.infsof.2006.10.017.

[17] Ralf Lämmel and Chris Verhoef. Semi-automatic grammar recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001. doi:10.1002/spe.423.abs.

[18] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM. doi:10.1145/1134285.1134355.

[19] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[20] Mircea Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, November 2009.

[21] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming (SCP)*, page to appear, 2012.

[22] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press, 2010. doi:10.1145/1858996.1859058.

[23] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140. IEEE, 2009. doi:10.1109/MSR.2009.5069491.

[24] 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. doi:10.1145/1148493.1148513.

[25] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 57–62, New York, NY, USA, 2009. ACM. doi:10.1145/1595808.1595821.

[26] Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proceedings of MSR 2009*, pages 11–20, 2009.

[27] 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. doi:10.1109/WCRE.2001.957806.

[28] 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, 2005. ACM Press. Invited paper. doi:10.1145/1095430.1081707.

[29] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 130 –140, may 2010. doi:10.1109/MSR.2010.5463346.

[30] Fabrizio Perin, Tudor Gîrba, and Oscar Nierstrasz. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *Proceedings of International Conference on Software Maintenance 2010*, September 2010. doi:10.1109/ICSM.2010.5609572.

[31] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/ASE.2008.42.

[32] Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems (nier). In *Proceedings of the 33rd International Conference*

on Software Engineering (ICSE 2011), pages 904–907, May 2011. doi:10.1145/1985793.1985940.

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

[34] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: raising risk awareness. *SIGSOFT Softw. Eng. Notes*, 30:107–110, September 2005. doi:10.1145/1095430.1081725.

[35] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[36] C.C. Williams and J.K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *Software Engineering, IEEE Transactions on*, 31(6):466 – 480, June 2005. doi:10.1109/TSE.2005.63.

[37] Sandro De Zanet. Grammar generation with genetic programming — evolutionary grammar generation. Master's thesis, University of Bern, July 2009.