Agile processes evolution challenges

Ayed, Hajer; Habra, Naji; Vanderose, Benoît

BENEVOL 2013

Software Evolution Research Seminar

16-17 December 2013, Mons, Belgium

Edited by : M. Claes
S. Drobisz
M. Goeminne
T. Mens

Extended Abstracts of the Research Presentations

BENEVOL 2013 Extended Abstracts

CONTENTS

The Impact of Release Engineering on Software Quality

Bram Adams

Abstract

Software release engineering is the discipline of integrating, building, testing, packaging and delivering qualitative software releases to the end user. Whereas software used to be released in shrink-wrapped form once per year, modern companies like Intuit, Google and Mozilla only need a couple of days or weeks in between releases, while lean start-ups like IMVU release up to 50 times per day! Shortening the release cycle of a software project requires considerable process and development changes in order to safeguard product quality, yet the scope and nature of such changes are unknown to many. For example, while migrating towards rapid releases allows faster time-to-market and user feedback, it also implies less time for testing and bug fixing.

To understand these implications, we empirically studied the development process of Mozilla Firefox in 2010 and 2011, a period during which the project transitioned to a shorter release cycle. By comparing crash rates, median uptime, and the proportion of post-release bugs before and after the migration, we found that (1) with shorter release cycles, users do not experience significantly more post-release bugs and that (2) less bugs are being fixed, but those that are fixed are fixed faster.

In order to validate these findings, we interviewed Mozilla QA personnel and empirically investigated the changes in software testing effort after the migration towards rapid releases. Analysis of the results of 312,502 execution runs of Mozilla Firefox from 2006 to 2012 (5 major traditional and 9 major rapid releases) showed that in rapid releases testing has a narrower scope that enables deeper investigation of the features and regressions with the highest risk, while traditional releases run the whole test suite. Furthermore, rapid releases make it more difficult to build a large testing community, forcing Mozilla to increase contractor resources in order to sustain testing for rapid releases. In other words, rapid releases are able to bring improvements in software quality, yet require changes to bug triaging, fixing and testing processes to avoid unforeseen costs.

Replication and Benchmarking: Challenges for Software Evolution Data Analysis

Harald C. Gall

Abstract

The replication of studies in mining software repositories (MSR) has become essential to compare the many mining techniques or assess their findings across many projects. However, it has been shown that very few of these studies can be easily replicated. Their replication is just as fundamental as the studies themselves and is one of the main threats to their validity. In this talk, we discuss the challenges of replication studies and benchmarking of software projects compared to a sample set of systems. We also show how with our SOFAS framework we can help alleviate this problem. SOFAS is a platform that enables a systematic and repeatable analysis of software projects by providing extensible and composable analysis workflows. These workflows can be applied on a multitude of software projects, facilitating the replication and scaling of mining studies. We show how and to which degree replication of studies can be achieved. We also characterize how studies can be easily enriched to deliver even more comprehensive answers by extending the analysis workflows provided by the platform.

I. Short Bio

Harald C. Gall is a professor of software engineering in the Department of Informatics at the University of Zurich, Switzerland. His research interests include software engineering and software analysis, focusing on software evolution, software quality, empirical studies, and collaborative software engineering. He is probably best known for his work on software evolution analysis and mining software archives. Since 1997 he has worked on devising ways in which mining these repositories can help to better understand software development, to devise predictions about quality attributes, and to exploit this knowledge in software analysis tools such as Evolizer, ChangeDistiller, or SOFAS. In 2005, he was the program chair of ESEC-FSE, the joint meeting of the European Software Engineering Conference (ESEC), and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). In 2006 and 2007 he co-chaired MSR, the International Workshop and now Working Conference on Mining Software Repositories, the major forum for software evolution analysis. He was program co-chair of ICSE 2011, the International Conference on Software Engineering, held in Hawaii. Since 2010 he is an Associate Editor of IEEE Transactions on Software Engineering. He received the ICSM 2013 Most Influential Paper Award for his work on populating a release history database.

http://seal.ifi.uzh.ch/gall

Agile processes evolution challenges

Hajer Ayed
PReCISE Research Center,
University of Namur,
Belgium
hay@info.fundp.ac.be

Naji Habra
PReCISE Research Center,
University of Namur,
Belgium
nha@info.fundp.ac.be

Benoît Vanderose
PReCISE Research Center,
University of Namur,
Belgium
bva@info.fundp.ac.be

I. Introduction

In the last few decades, agile processes have grown increasingly popular among the software engineering community. The rationale for using agile in many environments is to allow change. Indeed, the traditional methodologies have the implicit assumption that requirements can be final and that only minor variations can be accommodated later. Conversely, the agile methodologies assume that change is inevitable all along the software development life-cycle and thus, encourage rapid and flexible response to it. The core value of the agile paradigm is therefore to enable change management and not to prevent it.

Paradoxically, the agile practitioners promote the evolution and adaptation of the software but do not focus enough on the changes that may affect the process itself. A common misuse of agile is to adopt a ready-made method without much discernment and that, just because it worked well elsewhere. To avoid failure, most experts and researchers recommend process adaptation and evolution in order to make it suitable to the specific circumstances of the organisation and project. There's a large spectrum of techniques that can be used to support agile process evolution. This paper investigates a model-driven approach for agile processes evolution and assessment. This approach promotes the co-evolution between the software products and their process.

II. Key challenges

A. Agile process customisation and assessment

Practitioners very often experience the challenge of distinguishing convenient agile techniques and practices based on their culture, their values, and the types of systems that they develop. This procedure results in a context-specific process that combines two or more ready-made agile practices and/or blends agile and non-agile practices. This kind of evolution is called agile process customisation. Another important challenge is agile process assessment : the extent to which the process meets the needs of the project should be confirmed.

Most studies that have been undertaken on agile methods customisation are specific to a particular situation and concentrate on reporting the organisation way of customising[1][2]. The problem with such studies is that they are hardly reusable and/or generalisable, since no automation techniques are provided. Few studies such as [3] advocate formal methods for initial agile processes adaptation but do not provide support for later evolution of the agile process model.

Most of the existent agile assessment approaches (such as [4][5]) focus on the agile or plan-driven practices selection based on a comparative analysis. Many of them are also limited to the working software scope (e.g., assessment of the iteration velocity, assessment of the product quality, etc.) [6] and do not provide any support for the enacted process assessment. Such approaches provide a good starting point but cannot be used for assessing the suitability of the enacted agile processes either they are customised or not.

B. Model-driven Process Evolution

Researchers from both industry and academia have pointed out that software processes, including agile processes, need to be rigorously defined through relevant models, in order to support and facilitate their understanding, assessment and automation. We should also be able to analyse them through metrics and well defined quality assessment based analysers, in order to be able to improve them iteratively.

Moreover, in order to model the evolution of the process overtime and the co-evolution with the software, we need to be able to capture the interactions between the modelled process and the enacted process. We therefore need to raise the abstraction level and design an agile processes metamodel.

In fact, despite the differences in fine-grained details, all agile processes follow a common paradigm and can be described by a generic agile processes metamodel. We have found few studies about agile process metamodelling in the literature [3][7] and none of them is targeted for process evolution or assessment.

For example, [3] proposed a metamodel for partial agile method adaptation. This research aims at the description of a formal roadmap of how to configure a method for a partial adaptation, i.e., how agile methods can be broken down into a set of elements and how they can be combined using techniques like merging and generalizing similar elements. The proposed metamodel focuses primarily on partial agile method composition and do not address any evolution issue.

In order to effectively support the goals of modelling analysis and automation, an ideal agile process metamodel must exhibit several characteristics. To give but a few examples :

- It must describe the activities, practices, stakeholders and the expected resulting artefacts
- It must describe evolution and assessment metrics that will be used as an input for process analysers
- It must provide means to capture the dynamic behaviour, for example, by clearly defining the interaction between stakeholders and the operations they perform.

III. AM-QuICK: A MODEL-DRIVEN APPROACH FOR AGILE PROCESSES EVOLUTION

In order to address the challenges discussed in II, we investigated in [8] and [9] a model-driven approach for agile methods evolution that we called AM-QuICK : Agile Methods Quality-Integrated Customisation Framework. AM-QuICK aims to continuously assist agile methodologists, i.e., during the design of the process in the organisation level and throughout its enactment in the process level (Fig.1). The process design is performed thanks to an agile meta-model adapted from the more generic process metamodels SPEM , SMSDM and OPF [10]. More details about this metamodel can be found in [9].

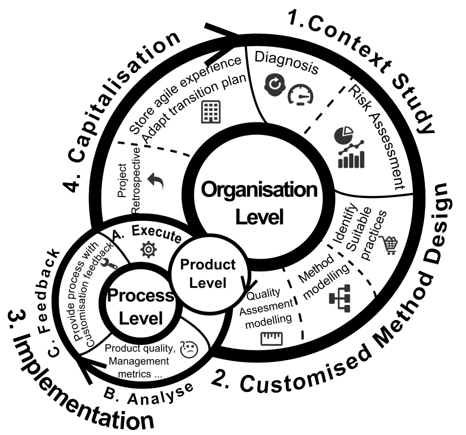


Figure 1. AM-QuICk overview

The AM-QuICk lifecycle, depicted in Fig. 1, consists of three cycles, each corresponding to one level : the first cycle handles the organisation strategy high-level (i.e., the agile transition strategy, the agile values in the business level, the agile culture adoption as an organisation shift of thinking, etc.) and occurs once during one project; the second is for process refinement and takes place continuously during the process execution and the third concerns the working product evolution.

This third cycle addresses the co-evolution between various software products. Any effort provided to ensure this form of co-evolution is therefore reflected at this level. In the meantime, any change monitored on the products is also taken into account in the second lifecyle and may result in a revision of the process for the next development or maintenance iteration. This way, the framework we propose extends the notion of co-evolution to the process itself, allowing to review the way software is developed or maintained, based on the last evolution of its constitutive products.

IV. Conclusion

In order to ensure the agile processes evolution and to assess their suitability, researchers form both academia and industry highlights the need to model them rigorously and to assess their suitability through metric-based analysers.

In this paper, we introduced a model-driven approach to support agile processes design and evolution according to environment changes. The approach implements a generic agile processes metamodel that will serve as a basis for specific processes composition.

The metamodel should evolve in the future, in order to include dynamic behaviour between its elements, so it is able to represent the interaction between various software products and their related processes (and therefore their co-evolution). This will also allow to support working product evolution as data are gathered during process enactment and vice versa.

References

[1] B. Fitzgerald, N. Russo, and T. O'Kane, "Software development method tailoring at motorola," *Communications of the ACM*, vol. 46, no. 4, pp. 64–70, 2003.

[2] L. Layman, L. Williams, and L. Cunningham, "Exploring extreme programming in context: An industrial case study," in *Agile Development Conference, 2004*. IEEE, 2004, pp. 32–41.

[3] G. Mikulėnas, R. Butleris, and L. Nemuraitė, "An appraoch for the metamodel of the framework for a partial agile method adaptation," *Information Technology And Control*, vol. 40, no. 1, pp. 71–82, 2011.

[4] B. Boehm and R. Turner, *Balancing agility and discipline: A guide for the perplexed*. Addison-Wesley Professional, 2003.

[5] P. Lappo and H. C. Andrew, "Assessing agility," in *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004, pp. 331–338.

[6] S. Soundararajan and J. D. Arthur, "A structured framework for assessing the "goodness" of agile methods," in *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on*. IEEE, 2011, pp. 14–23.

[7] E. Damiani, A. Colombo, F. Frati, and C. Bellettini, "A metamodel for modeling and measuring scrum development process," *Agile Processes in Software Engineering and Extreme Programming*, pp. 74–83, 2007.

[8] H. Ayed, B. Vanderose, and N. Habra, "Am-quick : a measurement-based framework for agile methods customisation," in *IWSM-MENSURA*, 2013.

[9] H. Ayed., B. Vanderose., and N. Habra, "A metamodel-based approach for customizing and assessing agile methods," in *The International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2012.

[10] B. Henderson-Sellers, C. Gonzalez-Perez, and J. Ralyté, "Comparison of method chunks and method fragments for situational method engineering," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE, 2008, pp. 479–488.

Exploring Characteristics of Code Churn

Jos Kraaijeveld
Technische Universiteit Delft,
Software Improvement Group
mail@kaidence.org

Eric Bouwers
Technische Universiteit Delft,
Software Improvement Group
e.bouwers@sig.eu

Software is a centerpiece in todays society. Because of that, much effort is spent measuring various aspects of software. This is done using software metrics. Code churn is one of these metrics. Code churn is a metric measuring change volume be- tween two versions of a system, defined as sum of added, modified and deleted lines. We use code churn to gain more insight into the evolution of software systems. With that in mind, we describe four experiments that we conducted on open source as well as proprietary systems [1].

First, we show how code churn can be calculated on different time intervals and the effect this can have on studies. This can differ up to 20% between commit-based and week-based intervals. Secondly, we use code churn and related metrics to auto- matically determine what the primary focus of a development team was during a period of time. We show how we built such a classifier with a precision of 74%. Thirdly, we attempted to find generalizable patterns in the code churn progression of systems. We did not find such patterns, and we think this is inherent to software evolution. Finally we study the effect of change volume on the surroundings and user base of a system. We show there is a correlation between change volume and the amount of activity on issue trackers and Q&A websites.

References

[1] J. M. Kraaijeveld, "Exploring Characteristics of Code Churn," Master's thesis, Delft Technical University, The Netherlands, 2013.

Towards Base Rates in Software Analytics

Extended Abstract for BENEVOL 2013

Magiel Bruntink
System and Network Engineering research group,
Faculty of Science, University of Amsterdam
M.Bruntink@uva.nl

I. Introduction

Nowadays a vast and growing body of open source software (OSS) project data is publicly available on the internet. Despite this public body of project data, the field of software analytics has not yet settled on a solid quantitative base for basic properties such as code size, growth, team size, activity, and project failure. What is missing is a quantification of the base rates of such properties, where other fields (such as medicine) commonly rely on base rates for decision making and the evaluation of experimental results.

Software engineering also needs bases rates to judge experimental results. An example within software engineering could consist of a new methodology that indicates (with 100% recall and 70% precision) whether a project will fail. Without knowing the base rate of project failure in the population one would likely suffer a base rate fallacy by thinking the failure chance is 70% given just a positive test result. Given a project failure base rate of 20%, the chance of a project failing given a positive test would be only 45%.

This talk contributes results of our ongoing research towards obtaining base rates using the data available at Ohloh (a large-scale index of OSS projects). The Ohloh software evolution data set consists of approximately 600,000 projects, from which we obtained a sample of 12,360 projects in July 2013. In this talk we will highlight results of analysing the quality of this data set, and further discuss ongoing work and future challenges.

II. Related work

The (quantitative) study of (open-source) software repositories has been going on for quite some time, leading to a rich body of literature. In the interest of brevity, not all approaches can be mentioned here. Surveys of the field have been provided by Kagdi et. al. in 2007 [2] and Hassan in 2008 [3]. A recent overview (2011) of the OSS ecosystem is being provided by Androutsellis-Theotokis et. al. in [4]. Examples of large software engineering data sets that are publicly available are GHTorrent [5], PROMISE [6], FLOSSmetrics [7], but more are available. A recent overview of data sets is given by Rodriguez et. al. in [8].

Software analytics is a term recently introduced by Zhang et. al. [9] to label research aimed at supporting decision making in software. Our work can be seen as supporting software analytics. Work that is closely related to ours has been done by Herraiz. He studied the statistical properties of data available on SourceForge and the FreeBSD package base [10]. We see our work as an follow-up in terms of scope and diversity, as by studying Ohloh, we use a larger and more diverse data source (which does not primarily focus on C).

Other researchers are also studying the data provided by Ohloh. Recently, Nagappan et. al. [11] used the Ohloh data set for defining a method to quantify the representativeness of empirical studies (on OSS projects). In 2008, Deshpande and Riehle reported on an investigation into the total size and growth of the OSS ecosystem using Ohloh [12]. Also using Ohloh, Arafat and Riehle reported on the distribution of commit sizes [13].

This extended abstract is based on a paper that has been accepted for publication in Science of Computer Programming. A preprint is available at CoRR [1].

III. Ohloh Software Evolution Data

In July 2013, we collected a large data sample from the open-source software index Ohloh [14]. Generally these data consist of a monthly aggregate of data resulting from analysis (done by Ohloh) of the source code and the version control system(s) of an OSS project. In total, data were collected for 12,360 OSS projects, resulting in a grand total of 785,760 project-months. Per month, data are provided on software evolution metrics such as code size, churn, version control activity and team size. The projects have been selected according to Ohloh's measure of project popularity (number of declared users on Ohloh), essentially starting with the most popular project (i.e. 'Mozilla Firefox'). All collected data and tools developed for processing are available publicly [15].

This talk discusses the research we are currently doing on the Ohloh data set, including:

- Results of analysing quality of the Ohloh data,
- Ongoing work in aggregating and summarising monthly data into yearly data,
- Ongoing work in deriving initial base rate results (e.g., for project failure),
- A look ahead to further exploration of the Ohloh data and future challenges.

References

[1] M. Bruntink, "Towards Base Rates in Software Analytics," *Science of Computer Programming, to appear, preprint available*, Oct. 2013. [Online]. Available: http://arxiv.org/abs/1310.0242

[2] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.

[3] A. E. Hassan, "The road ahead for mining software repositories," in *Proceedings of the 24th IEEE International Conference on Software Maintenance, Frontiers track*, 2008, pp. 48–57.

[4] S. Androutsellis-Theotokis, D. Spinellis, M. Kechagia, and G. Gousios, "Open source software: A survey from 10,000 feet," *Foundations and Trends in Technology, Information and Operations Management*, vol. 4, no. 3-4, pp. 187–347, 2011.

[5] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a firehose," in *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR 2012)*, 2012, pp. 12–21.

[6] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The PROMISE Repository of empirical software engineering data," Jun. 2012. [Online]. Available: http://promisedata.googlecode.com

[7] J. M. González Barahona, G. Robles, and S. Dueñas, "Collecting data about FLOSS development," in *Proceedings of the 3rd International Workshop on Emerging Trends in FLOSS Research and Development*. ACM Press, 2010, pp. 29–34.

[8] D. Rodriguez, I. Herraiz, and R. Harrison, "On software engineering repositories and their open problems," in *Proceedings of the First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. IEEE, 2012, pp. 52–56.

[9] D. Zhang, Y. Dang, J.-G. Lou, S. Han, H. Zhang, and T. Xie, "Software analytics as a learning case in practice: Approaches and experiences," in *International Workshop on Machine Learning Technologies in Software Engineering*. ACM, 2011, pp. 55–58.

[10] I. Herraiz Tabernero, "A statistical examination of the properties and evolution of libre software," Ph.D. dissertation, Madrid, Oct. 2008.

[11] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in Software Engineering Research," in *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, Aug. 2013.

[12] A. Deshpande and D. Riehle, "The total growth of open source," *Open Source Development, Communities and Quality*, pp. 197–209, 2008.

[13] O. Arafat and D. Riehle, "The commit size distribution of open source software," in *Proceedings of the 42nd Hawaii International Conference on System Sciences*. IEEE, 2009, pp. 1–8.

[14] Black Duck Software, Inc., "Ohloh," 2013. [Online]. Available: http://www.ohloh.net

[15] M. Bruntink, "OhlohAnalytics data set and analysis tools," 2013. [Online]. Available: http://github.com/MagielBruntink/OhlohAnalytics

When Behavior Disappears

Nicolás Cardozo
Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
ncardozo@vub.ac.be

THeo D'Hondt
Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium

Run-time behavior adaptation, or dynamic evolution, is becoming more and more prominent in the programming languages and middleware communities with the advent of adaptation programming models, such as Context-Oriented Programming (COP) [1] and Self-adaptive systems [2]. Software systems designed using these programming models are expected to be under continuous adaptation due to given situations from acquired information about their surrounding execution environment.

Programming models geared up towards adaptation, and in particular dynamic language-based approaches as COP, offer a new modularization perspective of the system. In such models, an adaptation is defined by means of a *context* (i.e., specific situation of the surrounding execution environment) and its associated *behavioral adaptations* (i.e., partial method definitions). Whenever a context becomes active, in response to changes in the surrounding execution environment of the system, its associated behavioral adaptations are made available to the run-time environment, (possibly) overwriting the behavior originally defined for such functionality (or base behavior). Similarly, if the context is no longer sensed in the surrounding execution environment of the system, then its associated behavior becomes unavailable to the run-time environment, consequently restoring the originally defined behavior.

Currently, programming models that support the dynamic adaptation of the system behavior do not provide any guarantee about the completeness of the system behavior. This means that it is not yet possible to ensure that for every possible situation in the surrounding execution environment of the system, the execution will correctly terminate. In other words, it is not possible to ensure (or verify) that the system execution will never yield a "*message not understood*" message [3]. We recognize, the problem of behavior completeness as a special kind of fragility problem [4] for dynamic systems, which we call the *method fragility problem.* Behavior completeness is indeed a fragility problem as, in this case, the behavior of the system may break due to its evolution (or adaptation) as response to context activation and deactivations. As methods are made available and unavailable in the run-time environment, the assumptions made by the system about the availability of other defined behavior may not hold any longer.

To show the existence of method fragility in dynamic systems let us take for example a situation where a context `A` is associated with a behavioral adaptation `foo` providing a specialized implementation of it, and reusing the behavior provided for a second behavioral adaptation of `foo` associated to a second context `B`. Suppose further that the behavioral adaptation of `foo` associated to `A` calls a second behavioral adaptation `bar` uniquely provided by context `B`. Let us take a situation, starting from the execution of the behavioral adaptation of `foo` associated with context `A`, in which after requesting the `foo` behavioral adaptation associated with context `B` and before calling of the behavioral adaptation `bar`, its associated context `B` becomes inactive. In such a case the execution of the `foo` behavioral adaptation associated with context `A` will not terminate correctly —that is, it will show a *message not understood* as a result of the execution.

To render such adaptive systems more usable, we must ensure that their behavior is absent of method fragilities. For this purpose, a verification approach could be used during the development phase of the system. As with other fragility problems, a set of rules of contracts that manage the completeness of the behavior at run time could be defined. For example, by defining interactions between contexts such that their expectations about the activation state of other contexts, and hence their associated behavioral adaptations, are represented. We identify two techniques that could be used to tackle the method fragility problem:

Context activation management: We have investigated different methodologies to manage the behavior of contexts activation and deactivation. These ideas implicitly express a contract defining the interaction between different contexts, for example by means of context dependency relations [5], or advance context selection mechanisms [6]. We believe these ideas could be formalize and complemented in a similar fashion as it is done with usage contracts, to define specific contracts for the interaction of contexts in dynamically adaptive programming models.

Static analysis: We could borrow ideas from the static analysis world to statically investigate the dependencies between all defined behavioral adaptations and the base behavior of the system. This approach would consist of marrying ideas from source code analyzers, such as JIPDA [7], and explicit representation of contexts and their associated behavioral adaptations, such as context Petri nets (CoPN) [8]. The combination of the two approaches would be able to identify possible points in which method fragility could take place by: Firstly, identify all dependencies between all defined behavioral adaptations and the base behavior of the system. Such information would be provided by an initial source code analysis phase. A second phase would taking into account the associated contexts to such behavioral adaptations, using a CoPN like approach, it could be possible to generate a second call graph, in which expected context dependencies are represented for each behavioral adaptation. Finally, developing a dedicated verification mechanism, we could identify the situations in which the expected context dependencies could be violated, and hence have an incomplete implementation of the system behavior.

REFERENCES

[1] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-oriented programming: A software engineering perspective," vol. 85, no. 8, pp. 1801–1817, Aug. 2012.

[2] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[3] M. D. Ernst, C. S. Kaplan, and C. Chambers, "Predicate dispatching: A unified theory of dispatch," in *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, July 1998, pp. 186 – 211.

[4] K. Mens, "Fragility of evolving software," 12th BElgian Nederland EVOLution seminar, 2013.

[5] N. Cardozo, J. Vallejos, S. Gonzlez, K. Mens, and T. D'Hondt, "Context Petri nets: Enabling consistent composition of context-dependent behavior," ser. CEUR Workshop Proceedings, L. Cabac, M. Duvigneau, and D. Moldt, Eds., vol. 851. CEUR-WS.org, 2012, pp. 156–170, 25–26 June 2012. Co-located with the International Conference on Application and Theory of Petri Nets and Concurrency.

[6] N. Cardozo, S. Gonzlez, K. Mens, and T. D'Hondt, "Safer context (de)activation through the prompt-loyal strategy," ser. COP'11. ACM Press, 2011, pp. 2:1–2:6, 25 July 2011. Co-located with ECOOP.

[7] J. Nicolay, C. Noguera, C. De Roover, and W. De Meuter, "Determining coupling in javascript using object type inference," in *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM'13, 2013.

[8] N. Cardozo, "Identification and management of inconsistencies in dynamicaly adaptive software systems," Ph.D. dissertation, Université catholique de Louvain - Vrije Universiteit Brussel, Louvain-la-Neuve, Belgium, September 2013.

Study on the Practices and Evolutions of Selenium Test Scripts

Laurent Christophe Coen De Roover Wolfgang De Meuter

I. Introduction

Nowadays, testing is the primary means to safeguard the quality of a software system. Of particular interest is functional GUI testing in which an application's user interface is exercised along usage scenarios. An exclusively manual activity in the past, testing has recently seen the arrival of test automation tools like HP Quick Test Pro, Robotium and Selenium. These execute so-called test scripts which are executable implementations of test scenarios that are traditionally written in natural language. Test scripts consist of commands that simulate the user's interactions with the GUI (e.g., button clicks and key presses) and of assertions that compare the observed state of the GUI (e.g., the contents of its text fields) with the expected one.

In this study we focus on Selenium[1], a popular test automation framework for web applications. In contrast to headless testing, where a browser is simulated by a JavaScript environment like Phantom JS, Selenium uses an actual browser to exercise the web application under test. Our choice to work with Selenium is motivated by the quality of its protocol to communicate with the running browser: WebDriver. Indeed, WebDriver is an elegant API which addresses much of the problems related to programmatically interacting with a GUI. Also, WebDriver has been proposed as a W3C standard[2]. The small Selenium test script depicted below performs a Google search and inspects the title of the resulting page.

```
WebDriver driver = new FirefoxDriver();
driver.get("http://www.google.com");
WebElement element = driver.findElement(By.name("q"));
element.sendKeys("cheese!");
element.submit();
assert(driver.getTitle().startsWith("cheese!"));
```

II. Motivation and Related Work

Although test automation allows repeating tests more frequently, it also brings about the problem of maintaining test scripts: as the system under test evolves, its test scripts are bound to break. Assertions may start flagging correct behavior and commands may start timing out thus precluding the test from being executed at all. For diagnosing and repairing these defects, test engineers have little choice but to step through the script using a debugger [1] —an activity found to cost Accenture $120 million per year [2].

At the seminar, we will present the results of a study on the usage and evolution of test scripts. Our motivation for studying maintenance operations on test scripts is to evaluate the feasibility of repairing broken test scripts automatically. However, we expect that our results will also help the testing community to build more robust test scripts. Some studies have been conducted to evaluate quantitative aspects of test script repairs [3], [2], [4] ; for instance the frequency at which test scripts are repaired and the associated costs. In this study we focus on qualitative aspects of test script repairs instead.

[1]http://www.seleniumhq.org/
[2]http://www.w3.org/TR/webdriver/

III. Corpus and Methodology

The dataset that we work on is the result of querying two large code repositories: SourceForge and GitHub. We are using the new BOA [5] query system to select relevant projects of the SourceForge repository. For instance, we found 105 SourceForge projects written in Java that uses Selenium. To query GitHub we use its built-in search API ; a simple search for `import org.openqa.selenium` returns 102.274 matches. The in-depth exploration of each project has been carried out using our own tools (i.e., Exapus [6] and QwalKeko [7]).

IV. Research Questions

To better understand Selenium testing practices, we have investigated the following research questions:

1) *Project layout* Where are calls to the Selenium API located? Are they directly located inside test cases or do test engineers prefer to use a layered architecture where test cases call helpers instead? For instance, those helpers could set up a uniform policy for handling timeout exceptions. Also, are test cases always situated in dedicated files? In the affirmative, are those files grouped together inside a separated folder or are they rather nearby the code they exercise?
2) *Recurrent test patterns* Can we identify any recurrent patterns in test scripts? For instance, we expect some sequences of commands to reccur such as fetching a web page and then filling a login form. Other examples include instances of test-specific design patterns (e.g., PageObject[3] and PageFactory[4]).
3) *Recurrent repair patterns* Can we identify any recurrent patterns of repairs in the history of test scripts? For instance, we expect hard-coded data like DOM locators to be brittle and often updated.

At the seminar, we will present the initial results of our study. We will investigate how these aspects are correlated in future work.

References

[1] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, 2011.

[2] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *Proceedings of the 31st International Conference onSoftware Engineering (ICSE09)*, 2009.

[3] A. M. Memon and M. L. Soffa, "Regression testing of guis," in *ACM SIGSOFT Software Engineering Notes*, vol. 28. ACM, 2003, pp. 118–127.

[4] F. R. Maurizio Leotta, Diego Clerissi and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE13)*, 2013.

[5] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE13)*, 2013.

[6] C. De Roover, R. Lmmel, and E. Pek, "Multi-dimensional exploration of api usage," in *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC13)*, 2013.

[7] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13)*, 2013.

[3]https://code.google.com/p/selenium/wiki/PageObjects
[4]https://code.google.com/p/selenium/wiki/PageFactory

Characterizing, Verifying and Improving Software Resilience with Exception Contracts and Test Suites

Benoit Corny, Lionel Seinturier, and Martin Monperrus
INRIA - University of Lille

I. Introduction

At the Fukushima nuclear power plant, the anticipated maximum height of waves in a tsunami was 5.6m [1]. However, on March 11, 2011, the highest waves struck at 15m. Similarly, in software development, there are the errors anticipated at specification and design time, those encountered at development and testing time, and those that happen in production mode yet never anticipated, as Fukushima's tsunami. In this presentation, we aim at reasoning on the ability of software to correctly handle unanticipated exceptions. We call this ability "software resilience". It is complementary to the concepts of robustness and fault tolerance [2]. Software robustness emphasizes that the system under study resists to incorrect input data (whether malicious or buggy). Fault tolerance can have a wide acceptation [3], but is mostly associated with hardware faults. "Software resilience" conveys the notion of risks from unanticipated errors (whether environmental or internal) at the software level.

II. Approach

We focus on the resilience against exceptions. Exceptions are programming language constructs for handling errors [4]. Exceptions are widely used in practice [5]. In our work, the resilience against exceptions is the ability to correctly handle exceptions that were never foreseen at specification time neither encountered during development. Our motivation is to help developers to understand and improve the resilience of their applications.

This sets a three-point research agenda: (RQ#1) What does it mean to specify anticipated exceptions? (RQ#2) How can one characterize and measure resilience against unanticipated exceptions? (RQ#3) How can one put this knowledge in action to improve the resilience?

Our approach helps the developers to be aware of what part of their code is resilient, and to automatically recommend modifications of catch blocks to improve the resilience of applications.

A. *What does it mean to specify anticipated exceptions?*

A test suite is a collection of test cases, each of which contains a set of assertions [6]. The assertions specify what the software is meant to do. Hence, we consider that a test suite is a specification since they are available in many existing programs and are pragmatic approximations of idealized specifications[7].

For instance, "assert(3, division(15,5))" specifies that the result of the division of 15 by 5 should be 3. But when software is in the wild, it may be used with incorrect input or encounter internal errors [8]. For instance, what if one calls "division(15,0)"? Consequently, a test suite may also encode what a software package does besides standard usage. For instance, one may specify that "division(15,0)" should throw an exception "Division by zero not possible". We will present a characterization and empirical study of how exception-handling is specified in test suites.

The classical way of analyzing the execution of test suites is to separate passing "green test cases" and failing "red test cases" [1]. This distinction does not consider the specification of exception handling. Beyond green and red test cases, we characterize the test cases in three categories: the pink, blue and white test cases. Those three new types of test cases are a partition of green test cases.

a) Pink Test Cases: Specification of Nominal Usage: The "pink test cases" are those test cases where no exceptions at all are thrown or caught. The pink test cases specify the nominal usage of the software under test (SUT), i.e. the functioning of the SUT according to plan under standard input and environment. Note that a pink test case can still execute a try-block (but never a catch block by definition).

[1]those colors refers to the graphical display of Junit, where passing tests are *green* and failing tests are *red*

b) Blue Test Cases: Specification of State Incorrectness Detection: The “blue test cases” are those test cases which assert the presence of exception under incorrect input (such as for instance “division(15,0)”). The number of blue test cases B estimates the amount of specification of the state correctness detection (by amount of specification, we mean the number of specified failure scenarii). B is obtained by intercepting all bubbling exceptions, i.e. exceptions that quit the application code and arrive in the test case code.

c) White Test Cases: Specification of Resilience: The “**white test cases**” are those test cases that do not expect an exception (they are standard green functional test cases) but use throw and catch at least once in application code. Contrary to blue tests, they are not expecting thrown exceptions but they use them only internally.

In our terminology, white test cases specify the “resilience” of the system under test. In our context, resilience means being able to recover from exceptions. This is achieved by 1) simulating the occurrence of an exception, 2) asserting that the exception is caught in application code and the system is in a correct state afterwards. If a test case remains green after the execution of a catch block in the application under test, it means that the recovery code in the catch block has successfully repaired the state of the program.

The number and proportion of white test cases gives a further indication of the specification of the resilience.

B. How to characterize and measure resilience against unanticipated exceptions?

We now present two novel contracts for exception-handling programming constructs. We use the term “contract” in its generic acceptation: a property of a piece of code that contributes to reuse, maintainability, correctness or another quality attribute. For instance, the “hashCode/equals” contract[2] is a property on a pair of methods. This definition is broader in scope than that in Meyer’s "contracts" [9] which refer to preconditions, postconditions and invariants contracts.

We focus on contracts on the programming language construct try/catch, which we refer to as “try-catch”. A try-catch is composed of one try block and one catch block.

d) Source Independence Contract: When a harmful exception occurs during testing or production, a developer has two possibilities. One way is to avoid the exception to be thrown by fixing its root cause (e.g. by inserting a not null check to avoid a null pointer exception). The other way is to write a try block surrounding the code that throws the exception. The catch block ending the try block defines the recovery mechanism to be applied when this exception occurs. The catch block recovers the particular encountered exception. By construction, the same recovery would be applied if another exception of the same type occurs within the scope of the try block.

This motivates the source-independence contract: the normal recovery behavior of the catch block must work for the foreseen exceptions; but beyond that, it should also work for exceptions that have not been encountered but may arise in a near future.

We define a novel exception contract, that we called “source-independence” as follows:

Definition A try-catch is source-independent if the catch block proceeds equivalently, whatever the source of the caught exception is in the try block.

e) Pure Resilience Contract: In general, when an error occurs, it is more desirable to recover from this error than to stop or crash. A good recovery consists in returning the expected result despite the error and in continuing the program execution.

One way to obtain the expected result under error is to be able to do the same task in a way that, for the same input, does not lead to an error but to the expected result. Such an alternative is sometimes called “plan B”. In terms of exception, recovering from an exception with a plan B means that the corresponding catch contains the code of this plan B. The “plan B" performed by the catch is an alternative to the “plan A" which is implemented in the try block. Hence, the contract of the try-catch block (and not only the catch or only the try) is to correctly perform a task T under consideration whether or not an exception occurs. We refer to this contract as the “pure resilience" contract.

A “pure resilience” contract applies to try-catch blocks. We define it as follows:

Definition A try-catch is purely resilient if the system state is equivalent at the end of the try-catch execution whether or not an exception occurs in the try block.

[2]http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

III. Empirical Evaluation

We perform an empirical evaluation on 9 well-tested open-source software applications. We made an evaluation on the 9 corresponding test suites, with 78% line coverage in average. The line coverage of the test suites under study is a median of 81%, a minimum of 50% and a maximum of 94%.

All the experiments are based on source-code transformation, using Spoon[3].

For the test colors, we analyzes 9679 test cases. It shows that between 5 and 19% of test cases expect exceptions (blue tests) and that between 4 and 26% of test cases uses exceptions without bubbling ones (white tests).

For the catch-contrats,we analyzes the 241 executed catch blocks, shows that 101 of them expose resilience properties (source-independence or pure-resilience).

IV. Conclusion

To sum up, our contributions are:

- A characterization of specification of software resilience in test suites,
- A definition and formalization of two contracts on try-catch blocks,
- A source code transformation to improve resilience against exceptions.

References

[1] "CNSC Fukushima Task Force Report," Canadian Nuclear Safety Commission, Tech. Rep. INFO-0824, 2011.
[2] J.-C. Laprie, "From dependability to resilience," in *Proceedings of DSN 2008*, 2018.
[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.
[4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*, 3rd ed. Addison-Wesley, 2005.
[5] B. Cabral and P. Marques, "Exception handling: A field study in java and. net," in *ECOOP 2007–Object-Oriented Programming*. Springer, 2007, pp. 151–175.
[6] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.
[7] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs, tests, and oracles: the foundations of testing revisited," in *2011 International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 391–400.
[8] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 595–605.
[9] B. Meyer, "Applying design by contract," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[3] http://spoon.gforge.inria.fr/

Benchmarking Reverse Engineering Tools and Using Tool Output for Further Analysis

David Cutting
University of East Anglia,
United Kingdom

Joost Noppen
University of East Anglia,
United Kingdom

OVERVIEW

With the growing amount of complex legacy code in use by organisations the ability to comprehend codebases through the use of Reverse Engineering tools is increasingly important. Although a wide number of tools exist to perform such reverse engineering there is no objective standard to determine their relative strengths, weaknesses and overall performance or to validate new techniques. Furthermore for a discipline to mature objective comparison between tools through a form such as a benchmark is required.

In order to address this gap we created the Reverse Engineering to Design Benchmark (RED-BM). The benchmark offers the facilitation of tool performance comparison from a standardised set of source code artefacts against a number of initial metrics geared towards the use of reverse engineering tools to aid structural comprehension of software. The benchmark provides a mechanism for configuring and combining performance metrics which be used to form a weighted compound measures allowing for extensibility into different domains.

As part of RED-BM we assessed the performance of a number of tools for metrics and to provide their output within the benchmark as a yardstick measurement of new techniques. A wide degree of variety was seen in tool performance and also in the supposed standard for output and exchange of reverse engineering information (the Extensible Metadata Interchange format - XMI). Generally speaking it was found that tools are not capable of producing complete accurate models so it is foreseen that a future requirement could be the ability to amalgamate models generated from multiple tools to form a more complete structural picture.

To aid the creation of the benchmark tools were developed to automate the comparison between source code and reverse engineering output. The basis of these tools, specifically the ability to parse XMI generated from different reverse engineering tools, is being developed for future applications such as structural analysis and custom UML projection within the Eclipse platform.

THE BENCHMARK

RED-BM facilitates the empirical comparison of reverse engineering tools in a number of performance areas against provided software artefacts along with a “gold standard” for comparison and the output from existing industry standard tools. The benchmark consists of four central components:

- A set of sixteen software artefacts covering a range of architectural styles such as design patterns and varying sizes of code as a course-grained range of complexity. For each artefact a “gold standard” of detection is provided containing a full listing of classes and a sample set of relationships for the tools to be measured against.
- An base set of metrics included focus on structural elements of reverse engineering (primarily classes and relationships presented in UML class diagram projections), along with metrics separated based upon complexity (size) of artefact target code. Included is a weighted compound measure producing an overall result for structural component detection by a reverse engineering process. These can be used to rank existing reverse engineering systems and also to validate any new approaches.
- A full set of results including reverse engineering tool output and detailed metric analysis for a number of widely used reverse engineering applications.
- A ranked evaluation based upon the base metrics for all artefacts individually and in combination for each of the reverse engineering tools used.

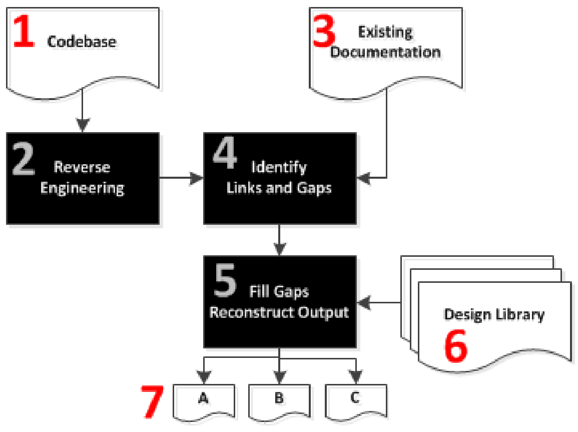


Fig. 1. High-Level Traceability Forensics Project Plan (processes shown in black-background rectangles, documents or artefacts in white background with wavy line)

RED-BM contains base measures for structural information and is extensible through the creation and inclusion of additional metrics, for example design patterns a worked example of which is included. The benchmark includes the facility to introduce additional performance measures and also combine performance measures to form compound performance indicators that can be targeted at specific domains, such design pattern recognition and sequence diagram reconstruction.

In addition we have created two tools to automate the generation of a “gold standard” in class detection and then measure reverse engineering output against this standard, highlighting inconsistencies. These tools are included with the benchmark release to facilitate replication of the initial results and also allow for quicker analysis with modified or extended metrics.

Current and Future Research Direction

This work is part of the Forensic Software Traceability project, the overall goal of which is to recover traceability information from partially or badly documented systems. We aim to complement areas of existing knowledge through novel techniques and processes. To this end we will work with existing codebases (1 - numbers in brackets refer to captions in figure 1) and reverse engineering output of these codebases (2). In combination with existing documentation (3) links and gaps between these data sources will be identified. These gaps will then be resolved through the use of complimentary additional processes (5) such as the use of a library of existing software designs (6) to identify structural commonalities.

Ultimately a set of reconstructed outputs will be generated (7) along with confidence weightings and rationales providing a comprehensive richer overall picture of a software system giving insight into form and function.

The tools developed for RED-BM are now being developed further to make more general programmatic use of reverse engineering output in XMI format. Working with a common XMI parser and interface two new tools concentrate on structural information recovery and comparison as well as integrating UML projection from multiple reverse engineering results within the Eclipse Platform (using UMLet).

It is intended that our XMI analysis and primarily identification of differences between outputs will be used within a reasoning component to make automated or user-guided decisions for structure leading to a more complete picture to be generated from a range of information. This will then ultimately be combined and cross-referenced with documentation analysis to generate a fuller set of both technical diagrams (UML) and textual description in the form of technical documentation.

Predicting Bug-Fixing Time using Bug Change History

Alessandro Murgia,
Javier Pérez,
Serge Demeyer
Ansymo group
, Universiteit Antwerpen

Coen De Roover,
Christophe Scholliers,
Viviane Jonckers
Software Languages Lab,
Vrije Universiteit Brussel

The constant need for change drives the manner in which modern software systems are conceived. However, version control systems, automated testing approaches, bug repositories and static analyses all start from the fundamental assumption that they act upon or build towards a single complete release of the system. Consequently, there exists a remarkable disparity between the trend towards embracing change and the tools used by today's software engineers. The CHA-Q project (Change-centric Quality Assurance)[1] is aimed at reducing this gap, striking the balance between agility and reliability through change-centric software development.

Recently, some authors have explored the potential of sampling the evolution of software systems at a fine-grained level of detail; for example recording the developer's edits right from the IDE and considering changes as first class objects [1]. One of the fields that can benefit from this approach is bug triaging –*i.e.* the set of activities related to bug management. There is a large body of work on predicting bug-fixing time [2], [3]. However, little work takes leverage of the information stored in the change history.

In this work, we investigate which change-related factors can be used by a classifier to better estimate the bug-fixing time. Some experiments have produced good results using change-history information for coarse-grained predictions [4]. However, are these results "good enough" for industry practitioners? To answer this question, we contacted industrial companies and collected a set of requirements fo making a good and useful estimator of bug-fixing time. Using these requirements, we run several experiments using the Mozilla bug database [5] and an industrial bug repository. We have explored several machine learning algorithms: K-Neighbours, Support Vector Machines, Multinomial Naive Bayes, Decision trees.

At the seminar we will present our results as well as the challenges we encountered. The skewness of the distribution of bug fixing time comprises a key one. Indeed, our predictors encounter difficulties estimating the bug fixing time for longer-living bugs. Although the initial results are promising, more work is needed to fine-tune our predictors and to select the most significant features in bug change history.

ACKNOWLEDGMENTS

This work has been sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled "Change-centric Quality Assurance (CHAQ)".

REFERENCES

[1] R. Robbes, "Of change and software," Ph.D. dissertation, University of Lugano, 2008.
[2] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Working Conference on Mining Software Repositories*, 2007.
[3] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Working Conference on Mining Software Repositories*, 2011.
[4] A. Lamkanfi and S. Demeyer, "Predicting reassignments of bug reports - an exploratory investigation," in *Conference on Software Maintenance and Reengineering*, 2013.
[5] A. Lamkanfi, J. Pérez, and S. Demeyer, "The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information," in *Working Conference on Mining Software Repositories*, 2013.

[1]http://soft.vub.ac.be/chaq/

The CHA-Q Meta-Model: A Comprehensive, Change-Centric Software Representation

Coen De Roover,
Christophe Scholliers,
Viviane Jonckers
Software Languages Lab,
Vrije Universiteit Brussel

Javier Pérez,
Alessandro Murgia,
Serge Demeyer
Ansymo group,
Universiteit Antwerpen

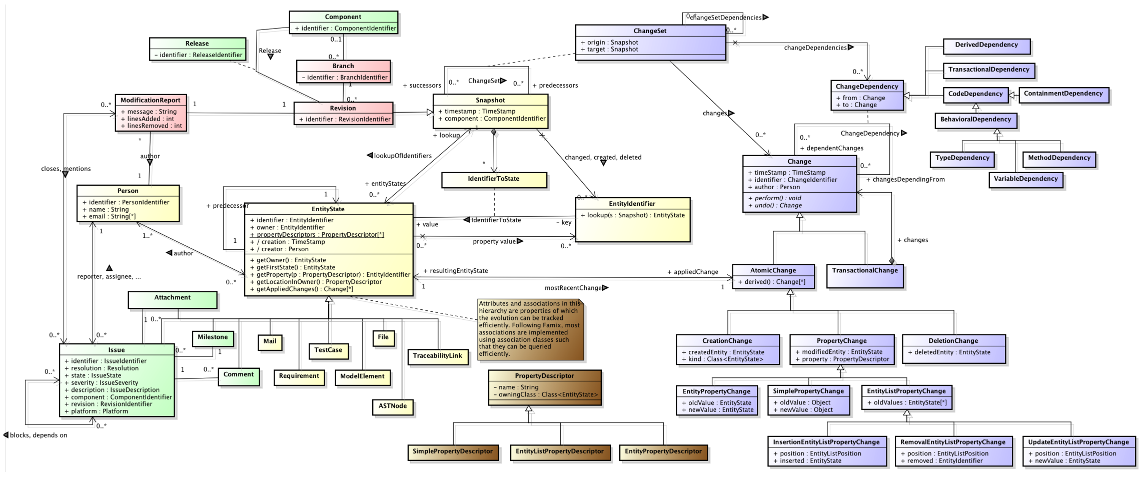
Although contemporary software development processes have embraced the need for continuous change, most development tools still assume that they act upon a single complete release of the system. The CHA-Q project (Change-centric Quality Assurance)[1] aims to strike a balance between agility and reliability through change-centric quality assurance tools. These tools are to share a first-class representation of changes to software artefacts. At the seminar, we will present the CHA-Q meta-model that defines this representation and highlight important characteristics of its implementation: an object-oriented API, persistency through a graph database, and a strategy for tracking the history of artefacts in a memory-efficient manner.

I. OVERVIEW OF THE CHA-Q META-MODEL

The CHA-Q meta-model defines a representation of the various artefacts that comprise a software system, as well as the complete history of all individual changes to these artefacts. Based on our experiences with the FAMIX [1], ChEOPS [2] and Ring [3] meta-models, we have opted for an object-oriented representation. Figure 1 depicts its high-level UML class diagram.

Changes are modeled as *first-class* objects that can be analyzed, repeated and reverted (cf. `Change`). To this end, we provide a representation of the dependencies between two changes (cf. `ChangeDependency`). These imply a partial ordering within a given set of changes (cf. `ChangeSet`). The corresponding elements are depicted in **blue**. Similar meta-models have already proven themselves for representing changes to code (e.g., SpyWare [4], ChEOPS [2] and Syde [5]) and to EMF models (e.g., UniCase [6]). Our meta-model goes beyond the state of the art by representing changes to the properties of any system artefact (i.e., source code, files, commits, bugs, e-mails, ...) in a uniform manner. This uniform treatment of an artefact's properties is inspired by the reflective API of the Eclipse JDT. The corresponding elements (cf. `PropertyDescriptor`) are depicted in **brown**.

[1]http://soft.vub.ac.be/chaq/



Attributes and associations in this hierarchy are properties of which the evolution can be tracked efficiently. Following Famix, most associations are implemented using association classes such that they can be queried efficiently.

Fig. 1. Overview of key CHA-Q meta-model elements.

Applying a change results in a new state for its subject (cf. `EntityState`). Figure 1 depicts the corresponding elements in yellow. Examples include abstract syntax trees (cf. `ASTNode`) and issues managed by an issue tracker (cf. `Issue`). The meta-model elements related to issue tracking and e-mail communication are inspired by the meta-model used by the Evolizer [7] and STNACockpit [8] tools respectively. Figure 1 depicts them in **green**.

Snapshots correspond to the state of all of a system's artefacts at a particular point in time as seen by a particular developer (cf. `Snapshot`). The delta between two snapshots is a set of changes (cf. `ChangeSet`). Snapshots of the entire system can be inspected and compared. This connection is similar to the one between Ring's history and change meta-model [3]. Revisions (cf. `Revision`) are snapshots placed under control of a version control system. Figure 1 depicts the corresponding elements, such as a modification reports and branches, in pink. These are inspired by the revision meta-model used by Evolizer [7].

II. Implementation Highlights

The CHA-Q meta-model associates a unique identifier (cf. `EntityIdentifier`) with each change subject (cf. `EntityState`). This enables tracking the evolution of a single subject throughout the history of a system. For each subject, a history of previous states is kept in a memory-efficient manner; successive states share the values of properties that do not change. We deem this necessary as copying of entity states has been observed to consume 3GB of memory for the Syde change-centric representation of a version repository of 78MB [5]. However, a selective cloning approach would be impractical to implement as all entities are interconnected transitively. We therefore follow the approach advocated by the Orion [9] and Ring [3] history meta-models. Property values are identifiers (cf. `EntityIdentifier`) that are looked up with respect to a particular snapshot.

To ensure that this additional level of indirection does not endanger type safety, our implementation relies on Java generics and property annotations. The property `initializer` of a `VariableDeclaration`, for instance, can only have `Expression` identifiers as its value:

```
public class VariableDeclaration extends ASTNode {
        @EntityProperty(value = Expression.class)
        protected EntityIdentifier<Expression> initializer;
        //....
}
```

Despite this memory-efficient representation, the working memory of a typical development terminal is unlikely to suffice for the entire history of the industry-sized projects that we aim to support. Our implementation therefore persists instances of meta-model elements to a Neo4J[2] graph database and retrieves them on a strict as-needed basis. Use of weak references ensures that instances that are no longer needed can be reclaimed by the garbage collector. Our two-way mapping is driven by run-time reflection about the aforementioned property annotations. This renders our implementation extensible. Extensive caching ensures that reflection does not come at the cost of a performance penalty.

Acknowledgments

This work has been sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled "Change-centric Quality Assurance (CHAQ)".

References

[1] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings of International Symposium on Principles of Software Evolution (ISPSE00)*, 2000.
[2] P. Ebraert, "A bottom-up approach to program variation," Ph.D. dissertation, Vrije Universiteit Brussel, March 2009.
[3] V. U. Gómez, "Supporting integration activities in object-oriented applications," Ph.D. dissertation, Vrije Universiteit Brussel - Université des Sciences et Technologies de Lille, October 2012.
[4] R. Robbes and M. Lanza, "Spyware: a change-aware development toolset," in *Proceedings of the 30th international conference on Software engineering (ICSE '08)*. ACM, 2008, pp. 847–850.
[5] L. P. Hattori, "Change-centric improvement of team collaboration," Ph.D. dissertation, Università della Svizzera Italiana, 2012.
[6] M. Koegel, J. Helming, and S. Seyboth, "Operation-based conflict detection and resolution," in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM09)*, 2009.
[7] H. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Softw.*, vol. 26, no. 1, pp. 26–33, 2009.
[8] M. Pinzger and H. C. Gall, "Dynamic analysis of communication and collaboration in oss projects," in *Collaborative Software Engineering*. Springer Berlin Heidelberg, 2010, pp. 265–284.
[9] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri, "Supporting simultaneous versions for software evolution assessment," *Science of Computer Programming*, vol. 76, no. 12, pp. 1177–1193, Dec. 2011.

[2]http://www.neo4j.org

Technical debt KPI for driving software quality evolution Feedback from an industrial survey

Nicolas DEVOS
29, rue des frres Wright
6041 Gosselies, Belgium
Tel: +32 71 490 700
Fax: +32 71 49 07 99
e-mail: nd@cetic.be

Dimitri DURIEUX
29, rue des frres Wright
6041 Gosselies, Belgium
Tel: +32 71 490 700
Fax: +32 71 49 07 99
e-mail: ddu@cetic.be

Christophe PONSARD
29, rue des frres Wright
6041 Gosselies, Belgium
Tel: +32 71 490 700
Fax: +32 71 49 07 99
e-mail: cp@cetic.be

Abstract

IT start-ups often emerge from academic research grounds where software development is primarily driven by innovative research. Software quality dimensions such as maintainability, portability and evolvability are not a priority and are often dependent on the personal background of the involved researchers. This deficit of quality (or technical debt) generally reveals itself when transitioning from research to company exploitation. In this paper, we report about our work of assessing and advising IT-startups hosted in a pool of Belgian incubators. In addition to the innovative nature of each project, the incubator strategic board is also taking software quality into account as part of the global evaluation of the future company potential. We report here the use of high level manager oriented indicators like technical debt on top of product and process metrics. We draw some observations and discuss them from a risk management and recommendation point of view.

I. Introduction

Reaching perfect quality in software development is never achieved and is also generally not required in most application domains. However, living with imperfect software has a cost - known as technical debt [1]. It increases the effort required to maintain and evolve the software. In addition, aspects such as performance, security or scalability will be more difficult to manage at a long term, with the increasing numbers of data or customers. Ignoring importance of this debt and not having a strategy to keep it under control will in the long run have damageable impact on the business potential.

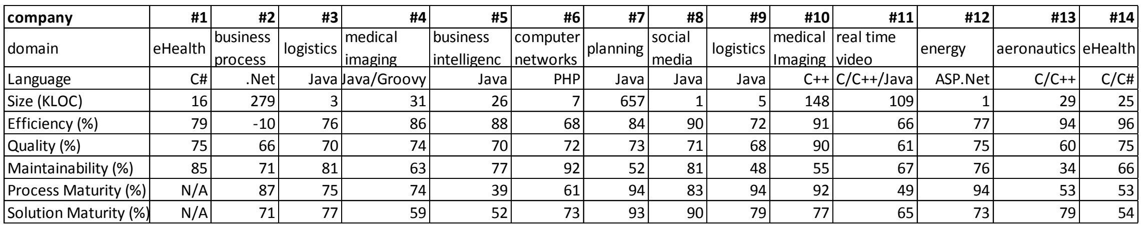
The focus of this survey is on IT start-up companies which have a high risk of accumulating technical debt. Such companies may lack of maturity in software development and evolution. In addition, it has been shown that economic factors can also come into play such as the need to shorten the time to market, preserve the startup capital or delay the development expense [2].

IT start-ups are generally hosted in technology incubators. In addition to ensure the follow-up on innovation and business plan validation, incubators are also increasingly considering the quality related to the solution and its development (both on product and process sides). This helps start-up to manage their investments and take better decisions for a longer perspective. On one hand, investigating the technical quality of a solution is a way to investigate the technical skills of the project team and take actions in terms of training, coaching or engagement. On the other hand, the internal quality of a solution determines how it will support evolution. Without measuring quality, it is actually impossible for investors to manage this aspect and to anticipate future development efforts. By nature, investors do not have the skills necessary to investigate technical quality. So they need to rely on a trusted IT partner that has the capabilities to perform a quality assessment and express its results in terms of risks.

In this paper, we have carried out a specific survey of management level indicator such as the technical debt in a sample of 14 IT start-ups. We have developed together with incubators a specific assessment approach - part of a larger follow-up program - with the goal to provide adequate means to control this risk. The approach combines both product and process quality aspects.

- **At process level**, high level indicators such as technical debt can be calculated as the effort required to correct quality flaws that remain in the code. It is based on static source code analysis [3][4]. A set of quality rules is defined based on development good practices and standards. This set covers several aspects of the internal quality (ISO9126) from maintainability (e.g. documentation, comments and naming convention, architectural design) to performance (e.g. loop and resource management) via reliability and security (e.g. error management or unsecure construction).

- **At process level**, the new standard ISO/IEC 29110 Lifecycle profiles for Very Small Entities was used at the entry profile which is a sufficient basis both because of the expected maturity level but also to keep it lightweight [5].



company	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**	**#13**	**#14**
domain	eHealth	business process	logistics	medical imaging	business intelligenc	computer networks	planning	social media	logistics	medical Imaging	real time video	energy	aeronautics	eHealth
Language	C#	.Net	Java	Java/Groovy	Java	PHP	Java	Java	Java	C++	C/C++/Java	ASP.Net	C/C++	C/C#
Size (KLOC)	16	279	3	31	26	7	657	1	5	148	109	1	29	25
Efficiency (%)	79	-10	76	86	88	68	84	90	72	91	66	77	94	96
Quality (%)	75	66	70	74	70	72	73	71	68	90	61	75	60	75
Maintainability (%)	85	71	81	63	77	92	52	81	48	55	67	76	34	66
Process Maturity (%)	N/A	87	75	74	39	61	94	83	94	92	49	94	53	53
Solution Maturity (%)	N/A	71	77	59	52	73	93	90	79	77	65	73	79	54

Figure 1. Key quality indicators of IT start-up projects.

The typical duration of a survey is 5 days, including contact, process assessment interview, code source gathering and analysis, report writing (including recommendations) and final debriefing with action plan.

II. Overview of survey results and discussion

A. *Some observations*

- The efficiency indicator reflects the importance of the technical debt. Not surprisingly about the half of start-ups have let their debt grow above 25%, a level requiring to pay attention especially on large code base. In a single case, it was advised to throw away the existing code and restart from scratch as the estimated correction effort was exceeding the debt.
- When looking at the details of the debt composition, each project has a specific balance of flaws (design problems, standard programming practices violations, test coverage, complexity and lack of documentation). Tests and documentation are largely present in the top ranking of recommended enhancements.
- The observed quality indicator is quite homogeneous (between 65 and 75%). This is satisfactory. Only one start-up could achieve a very good quality on a consequent code base. This can be related to a higher level of maturity and to the criticality of their sector (need to comply with safety standards in a near future).
- The maturity of the development process is higher than expected from teams which are generally not composed of people having a degree in computer science. However they generally have a high degree of education in another scientific discipline. A number of development good practices (such as code versioning) are becoming largely observed. In several case, they were learned in the contact of Open Source projects.
- Globally, we can see that the quality results are largely dependent on the developers experiences. It is also clear that the general effort related to quality is highly dependent to the solution characteristics. In particular, the innovation field of the solution could clearly affect some quality results. For example, a solution which resolves an issue already addressed by other competitors but whose innovation resides in the fact that it is more efficient or adaptable has resulted in better quality results on performance or modularity.

B. *Risk analysis and mitigation for managing software evolution*

- From the incubator point of view, the main objective of the assessment is to offer a way to investigate quality in order to identify risks and define a mitigation strategy. Based on the assessment results, actions can be defined to address issues and anticipate risks.
- In the context of start-ups with limited resource to raise quality, the action plan proposed to managers is phased over time starting with short-term actions requiring low correction effort and high return on investment, followed by mid- or long-term actions with a specific quality target.

- The mitigation does not restrict to process improvement but also provides feedback on the technical skills of the project team. This can highlight needs to train or hire of a new team member.

Our experience with start-ups showed they were very receptive about the feedback and keen to quickly integrate the feedback to secure the future of their project. Some feedback may seem of little value to IT professional but are extremely useful for companies which may not have an IT profile within their team. In addition start-ups are often composed of highly motivated people forming a team with a strong cohesion with is an ideal ground to grow a quality approach.

REFERENCES

[1] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, 1993. [Online]. Available: http://c2.com/doc/oopsla92.html
[2] J. C.-H. M. Denne, *Software by Numbers*. Prentice Hall, 2003.
[3] Cast softare. [Online]. Available: http://www.castsoftware.com
[4] Sonarqube. [Online]. Available: http://www.sonarqube.org/
[5] "Software engineering - lifecycle profiles for very small entities (vses)." Geneva: International Organization for Standardization (ISO), 2011, iSO/IEC 29110:2011.

OSCAR: an empirical analysis of co-evolution in a data-intensive software system

Mathieu Goeminne
Software Engineering Lab,
University of Mons,
Belgium

Tom Mens
Software Engineering Lab,
University of Mons,
Belgium

Extended Abstract

Current empirical studies on the evolution of software systems are primarily analyzing source code. Very few studies, however, focus on *data-intensive software systems* (DISS), in which a significant part of the total development effort is devoted to maintaining and evolving the database schema, typically through the use of a specific database management system and database technology (such as Hibernate). Adding new functionality to the system may affect the database structure and, conversely, modifying the database structure may impact the source code associated to it. Because of this, evolution of the DISS requires the source code and the database schema to co-evolve. Very little empirical studies have been carried out to study the co-evolution between source code changes and database schema changes in a DISS.

In this presentation, we report on our ongoing research to empirically analyse the co-evolution for OSCAR, a non-trivial DISS. OSCAR is an open source Electronic Medial Record system used for healthcare and currently supporting over 1.5 million patients across Canada. The analyzed historical data of its source code, obtained from a Git repository, covers a period of more than 10 years, from November 2002 till August 2013. Over this period, 100 distinct persons have contributed more than 20,000 files to OSCAR, with over 18,000 commits and over 90,000 file touches. OSCAR has been implemented primarily in Java and JSP.

Using OSCAR as a case study, we wish to gain insight in the co-evolution between code-related and database-related activities. To achieve this, we have formulated a whole range of questions, some of which are mentioned below:

- Does the co-evolution between source code changes and database-related changes happen synchronously or is it phased?
- Are database schema changes less frequent than source code changes?
- Does everyone in the development team modify both the source code and database structure, or is there a clear separation of concerns, with a group of developers working primarily on the source code functionality, and another group being primarily in charge of the database changes?
- What is the impact on co-evolution of introducing a particular database technology? How do developers divide their effort between the activity types involved in evolving a DISS?

To answer these questions, we extracted all commits from OSCAR's git repository using CVSAnaly2 and stored all relevant information about file changes for in a FLOSSMetrics-compliant database. This database was iteratively enriched with extra views and tables to facilitate answering our research questions. We also relied on identity merging to cope with the fact that the same contributor may use a different account for different commits. Then, we statistically analysed the information stored in our database using R.

We observed that OSCAR is steadily growing over time, both in number of pure, database-unrelated code files and in number of database-related files. A few contributors are very active, while the majority of contributors are only occasionally contributing commits. We also observed that the proportion of Java files is increasing over time at the expense of the proportion of JSP files. The proportion of pure (database-unrelated) code files stays relatively stable over time, and always exceeds 60%. For the evolution database-related files, we observed some changes in database technology. In the beginning, raw SQL queries were embedded in the source code. Starting from July 2006, the ORM framework Hibernate was introduced. XML files were used to map Java classes to the database tables. Starting from July 2008, JPA technology was introduced as an alternative to these XML files by adding particular annotations directly in the Java classes that represent the database tables. These JPA annotations start to replace the XML files gradually over time. Using statistical techniques like correlation analysis we studied the relation between database-related and database-unrelated code file changes. With the exception of Hibernate mapping files, we did not find a clear separation of concerns between both types of activities.

Further analysis is needed to study these preliminary results in depth, and to explore the reasons behind this.

Acknowledgment: This research is carried out in the context of FRFC PDR research project T.0022.13 ”Data-Intensive Software System Evolution”, in collaboration with the University of Namur, and financed by the F.R.S.-FNRS Fonds de la Recherche Scientifique.

M^3: An Open Model for Measuring Code Artifacts

Anastasia Izmaylova Paul Klint Ashim Shahi Jurgen Vinju

I. Motivation

In the context of the EU FP7 project “OSSMETER” we are developing an infra-structure for measuring source code. The goal of OSSMETER is to obtain insight in the quality of open-source projects from all possible perspectives, including product, process and community.

The main challenge that our part of the design, which focuses on code, faces is variability: the different languages we support as well as the different metrics we will compute. The standard solution is to put an explicit model (database, graph) in between such that model producers (parsers & extractors) can be de-coupled from model consumers (metrics & visuals). This abstract is a “white paper” on M^3, a set of code models, which should be easy to construct, easy to extend to include language specifics and easy to consume to produce metrics and other analyses. We solicit feedback on its usability.

The context of M^3 is the Rascal meta-programming language[1]. This is a domain specific language specifically designed to include primitives we need to model any programming language syntax and semantics, and to analyze and manipulate these models. Three essential design elements for the purpose of this paper are that Rascal has value semantics for all in-memory data, including sets and relations, it has support for URI literals, called “source locations”, and it has term rewriting and relational calculus primitives to deal with hierarchical and relational data, respectively. This includes generic traversal and pattern matching primitives as well as relational operators such as transitive closure and comprehensions.

Caveat emptor. The reader should be aware that we do *not* intend to create a unified model for programming language semantics. Such a language independent model would be inaccurate (wrong), and deliver meaningless metrics. Instead we opt for a unified *form* for storing facts about programs. This means that all models will have a predictable shape, but we do not assume any reusability of metrics or visuals producers between models produced by different parsers.

M^3 is inspired by models such as FAMIX, RSF, GXL, ATerms and S-Expressions. The differences are that M^3 deals with purely immutable, typed, data and can be directly produced, manipulated and analyzed using Rascal primitives. Two unique elements are the introduction of URI literals to identify source code artifacts in a language agnostic manner and support for fully structured type symbols. Otherwise M^3 is very similar in intent and solution patterns to the aforementioned existing models.

II. Design aspects

A. Textual models

M3 is, like all Rascal data, fully typed and fully serializable as readable text with a standard notation that is equal to the expression syntax for literals. This means that any intermediate step can be visualized as plain text and not only searched and edited using standard text editing facilities, but also stored and retrieved persistently. One particular aspect of the Rascal IDE is that all printed source location literals (see below) in editors and consoles are treated as *hyperlinks*. M3 models are therefore “programmer friendly”: easy to explore both inter-actively and programmatically using low-brow techniques.

B. Locations

To verify the correctness of metrics or for explaining them we want to trace back measurements to code. For example, when we present the largest class in a project, we need the size as well as a link to the source code of this class. In other words, want to link information back to source code for all derived facts we produce. From the semantic web we take the idea of using URI (Uniform Resource Identifiers) to model the identity of any artifact. Each URI takes the following shape: `|<scheme>://<auth>/<path>?<qry>|(<off>,<len>)`.

We distinguish between two kinds of code locations: physical and logical. A *physical* location identifies a storage location. Physical locations may be absolute or relative. Examples of absolute physical locations are

[1]http://www.rascal-mpl.org

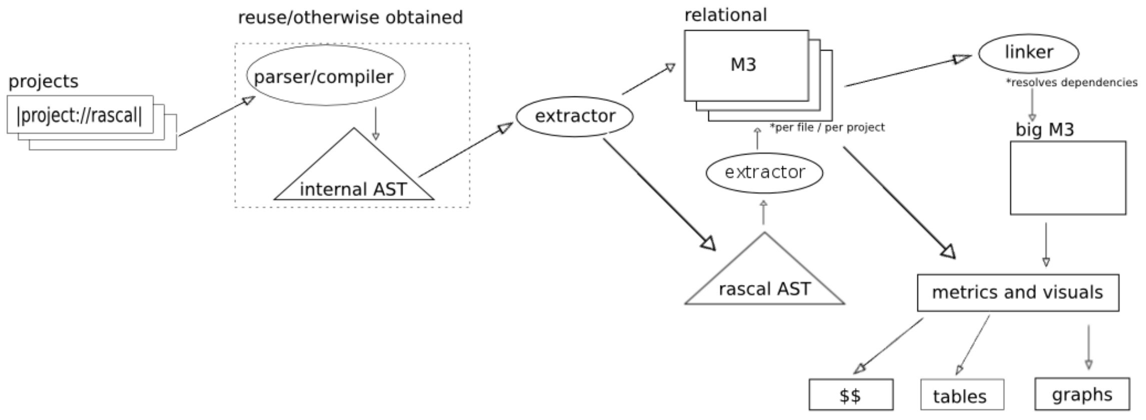


Fig. 1. An overview: from code to metrics and visuals via M3 models.

`|file:///tmp/Hello.java|` and `|http://foo.com/index.html|`, and `|project://MyPrj/Hello.java|` is a relative location. It is always the *scheme* of a URI that defines to which root a URI is relative. In the case of `project`, it is an Eclipse project in the current workspace, in the case of `cwd` it is the current working directory. The set of physical schemes is open and extensible. We have schemes for Eclipse projects, Java class resources, OSGI bundle resources, JDBC data sources, jar files, etc.

A *logical* code location is akin to a fully qualified name. For each specific language we design a naming scheme for each source code element that is, in some sense, declared. An example of a logical location is `java+class://myProject/java/util/List`. The scheme represents both the language and the kind of artifact that is identified. The authority declares the scope from which the name is resolved, in this case from `myProject` which depends on a particular version of the Java run-time. Finally, the path identifies the qualified name of the artifact in this scope. One goal of logical locations is to link uniquely to physical locations, at a certain moment in time, and at the same time be more or less stable under irrelevant code movement (such as moving the root source directory within a project). Another goal for such links is to be readable, writeable, recognizable and memorizable by human beings when developing new extractors, metrics or visuals. I.e. we might explore an M^3 model by projecting the information for an arbitrary class: the Rascal command `m@inheritance[|class://myPrj/java/util/List|]` would produce all interfaces that inherit from java.util.List.

The query part of a URI is used to *modify* identities, for example to scope them for a version of a system: `class://myPrj/java/util/List?svn=4242`. The offset and length fields are used to identify consecutive slice of characters of the identified artifact.

M^3 models are build on this concept of logical and physical source locations. It uses binary relations between locations, it annotates AST nodes with these locations and it embeds these locations into symbolic facts (such as types) to link back to source code whenever possible.

C. Relations.

The M^3 model is both layered and compositional. This means that M^3 models can be combined ("linked") and that they can be extended ("annotated"). The core relations are all between code locations: *containment* defines which artifact is (logically) contained in which other artifact, *declarations* define which logical locations are located at which physical locations, *uses* defines which logical locations are used by which other logical locations. An example containment tuple would be `<|class:///foo/Bar|,|pkg:///foo|>`.

This core model is language independent, facilitating not only, volume metrics, browsing visuals (drill-down) and generic aggregation over containment relations, but also dependence between artifacts and thus impact and coupling/cohesion analyses. Also note that this core model is not restricted to handling programming languages. It can be used without doubt to model other kinds of formal languages like grammars, schema languages or even pictorial languages.

For modeling language specific information we annotate the above core model with extra relations. Again these are binary relations between logical locations. Examples for Java are *inheritance*, *overrides*, *invocations*. These relations model key aspects of the static semantics of a programming language. Note that we never refer to instantiated or dynamic objects here, not even parametric type instantiations. All relations refer to source locations literally. For the accuracy of source code metrics, it is essential that M^3 separates what is written in the source code from what the code means dynamically. For example, if an abstract method from an interface is called we should not infer immediately all the call sites and add those to the invocations relation. Some metrics may want to count the fan-out to abstract methods, while other metrics want to know the impact on concrete implementations. You can compute this kind of information by composing basic facts, e.g. "invocation $\circ$ overrides" gives all the concrete callees for calls to abstract methods, and then compute a metric over the resulting relation instead.

D. Trees.

For abstract syntax trees we use a general concept of algebraic data-types in Rascal. Every language comes with its own definitions. Algebraic data-types are easy to extend with new constructors (new programming language constructs). For M^3 we standardize some of the names used in defining AST types. In the core we standardize on five algebraic sorts to use when defining an abstract syntax: `Expression`, `Declaration`, `Statement`, `Type`, `Modifier`. The goal is to add as few as extra sorts as possible when adding a new language. This leads to models which *over-approximate* the possible programs, but also increases the chance of reuse and extending existing fact extractors. For example, if all statements are in the same sort, then a basic function computing the cyclomatic complexity can be extended to cover a new language by just adding cases for the new types of statements (e.g. a `foreach` statement). We also provide annotations types for specific nodes, i.e. all nodes have a `src` annotation to point to the physical source location, all declarations may have a `decl` annotation to their logical location identifier and all Expressions may have a `type` annotation (see below).

Trees are useful mostly for the computation of metrics over code units that contain statements, such as cyclomatic complexity, but also to infer data and control flow information for use in the more advanced analyses. Trees are also expensive to keep in memory, so in M3 models they are always computed *on-demand* for a particular logical location.

E. Types

For types we introduce a single sort called `TypeSymbol`. We use this to represent any kind of abstract value that variables and expressions in a language may produce. For Java we have a default set of type symbols to represent (parametrized) class and interface types method signatures and its primitive types. These symbols can be used to compute with raw and parametrized types, either instantiated or uninstantiated. An example of a type symbol is: `class(|class:///java/util/List, [class(|class:///java/lang/String|,[])])`, meaning the instantiated parametrized List type generated by the List class definition, and its type parameter is instantiated by the String class. We extended the core M^3 model with initial types: a relation from declarations to the types they generate and we annotate the trees of expressions with the types they produce. Using type symbols we may compute with and reason about dynamic artifacts that are never declared yet may exist at run-time. For example, a metrics for the number of possible instantiations of a parametrized type can be computed based on such information.

F. Model composition

When we extract M^3 models we do this incrementally, i.e. per file, per project, per composition of a project with its dependencies. Each file (in a given programming language) produces one M^3 model. Then the models for all files in a project are fused into one single M^3 model by applying set union to all the relations of the model. Finally, if there are project dependencies, we may fuse the M^3 models for different projects.

Some analyses are best done before fusion. We compute the volume of a project before we fuse in the declarations of the jars we depend on. Other analyses are done only after fusing: Depth of inheritance can only be computed if the models of classes we depend on our fully available. Since M^3 models are immutable values, like all Rascal values, it can never happen that we accidentally mix such models up. The `compose` function is called explicitly by the programmer to union the relation between two M^3 models and the `link` function does the same but updates the authority fields of all logical locations such that uses from one project may point to the declarations of another.

Currently we have extractors of M^3models for jar files (i.e. from bytecode) from the JRE and Eclipse plugins, and from the source code of Eclipse project separately. We then link these independently acquired M3 models to form complete models for further analysis.

III. Conclusion

We have shown you a taste of M^3, an extensible and composable model for source code artifacts based on relations and trees, with immutable value semantics, source location literals and extensible with annotations. It has support for basic language independent analyses and we have a detailed model for Java. Extensions to be expected soon are C# and PHP support, and control flow and program dependence relations. We use M^3 in our course on Software Evolution at UvA and OU, and in the context of two research projects at CWI. At BENEVOL we hope to have discussion on its usability in a larger context of software analysis and software analytics.

Software Patterns for Runtime Variability in Online Business Software

Jaap Kabbedijk
Utrecht University

Slinger Jansen
Utrecht University

Extended Abstract

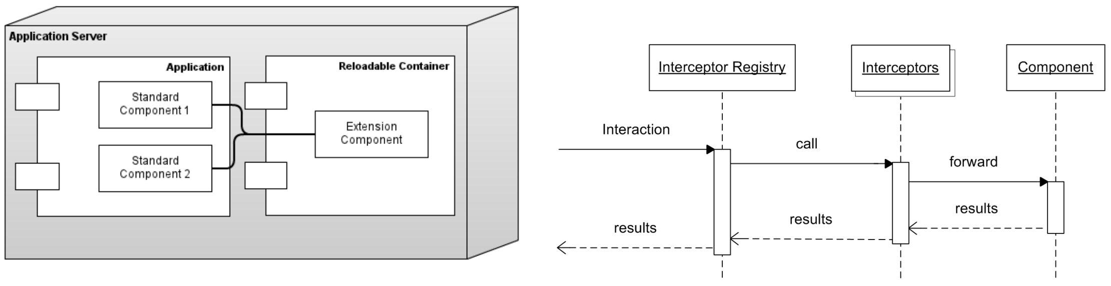
Business software is increasingly moving towards the cloud. As business software has to be offered as a multi-tenant solution, variability of software in order to fit requirements of specific customers becomes more complex. This can no longer be done by directly modifying the application for each client, because of the fact that a single application serves multiple customers in the Software-as-a-Service paradigm. A new set of software patterns and approaches are required to design software that supports runtime variability. We identify two frequently occurring problems in multi-tenant software and present four patterns solving these problems. Identifying these problems and representing the solutions in a structured way (i.e. patterns) helps software architects redesigning software products. Products have to evolve from on-premises single-user solutions to online multi-tenant solutions. This often causes large parts of the software to be redesigned and explicit descriptions of the problems and possible solutions faced while undertaking this evolution help in structuring the process.

The first problem related to runtime variability is the dynamical adaption of functionality in an online software product. We discuss the Component Interceptor Pattern and the Event Distribution Pattern as solutions for this problem. the second problem identified is the extension of the datamodel of software product. We discuss both the Datasource Router Pattern and the Custom Property Objects Pattern as solutions to this problem. The patterns are based on case studies of current software systems and are reviewed by domain experts. An evaluation of the patterns is performed in terms of security, performance, scalability, maintainability and implementation effort. To allow for the addition of extra functionality in the application, a solution is needed that allows configuration of such functionality. The solution can be found in implementing either the Component Interceptor Pattern or the Event Distribution Pattern. The appropriate pattern to implement depends on the specific situation and type of software product. For the sake of brevity, only one pattern and one problem are presented here.

The Component Interceptor Pattern as depicted in Figure 1a consists of only a single application server. Interceptors are tightly integrated with the application, because they run in-line with normal application code. Before the *StandardComponent* is called the interceptors are allowed to inspect and possibly modify the set of arguments and data passed to the standard component. To do this the interceptor has to be able to access all arguments, modify them or pass them along in the original form. Running interceptors outside of the application requires marshalling of the arguments and data to a format suitable for transport, then unmarshalling by the interceptor component and again marshalling the arguments to be passed on to the standard component that was being intercepted.

Figure 1b depicts the interaction between the interceptors involved. Interaction with standard components that can be extended goes through the interceptor registry. This registry is needed to keep track of all interceptors that are interested in each interaction. Without the registry the calling code would have to be aware of all possible interceptors. As depicted, multiple interceptors can be active per component. It is up to the interceptor registry to determine the order in which interceptors will be called. An example strategy would be to call the first registered interceptor first or to register an explicit order when registering the interceptors.

The complete research covers two of such problems and shows four patterns to solve the problems, giving a complete analysis of their attributes and showing a comparison between all patterns. All patterns are observed in currently active software products and all evaluations of quality attributes are performed by



(a) System Model (b) Sequence Diagram

Figure 1: Component Interceptor Pattern Models

experts. We conclude with matching different scenarios to different patterns and give software architects some rules of thumb when evolving their software product to a multi-tenant solution.

A conference paper on two of the patterns is presented at PATTERNS2013 and was awarded ‘best paper’. An extended version of the research containing four patterns will be submitted as journal article.
Corresponding author: Jaap Kabbedijk (J.Kabbedijk@uu.nl)

Who's Who on GNOME Mailing Lists: Identity Merging on a Large Data Set

Erik Kouters, Bogdan Vasilescu, Alexander Serebrenik
Technische Universiteit Eindhoven,
Den Dolech 2, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
e.t.m.kouters@student.tue.nl, {b.n.vasilescu, a.serebrenik}@tue.nl

INTRODUCTION

An open-source software project often uses multiple channels of communication (e.g., software repository, mailing list, bug tracker) in each of which an individual has to create a new identity. People might switch between different email addresses (e.g., private and corporate email addresses), causing a mailing list to contain multiple identities for a single individual. Identity merging attempts to solve the challenge of aggregating data by individuals instead of identities (e.g., by merging identities throughout multiple conferences [1]).

Identity merging is the process of identifying which identities belong to the same individual. Aliases are values identifying an individual, commonly found in the form of different $\langle name, emailAddress\rangle$ tuples in mailing lists and source code repositories. By using an identity merging algorithm, two aliases will be matched based on the similarity determined by the algorithm. When two aliases are matched positive, they are considered as belonging to the same individual.

Existing identity merging algorithms are not very robust to noise (e.g., misspelling, diacritics, nicknames, punctuation). We introduce an algorithm which is inspired by information retrieval [2]. We have evaluated this algorithm's performance, and compared it to three existing identity merging algorithms. We present preliminary results of evaluating the performance on a large data set. A priori it is unknown how the algorithms will perform on a data set with this order of magnitude.

The identity merging was performed on GNOME's mailing list archives, which were extracted on April 11, 2012. At that time, the mailing list archives contained $2,202,746$ emails which were sent by $73,920$ distinct email addresses. A previously studied data set, GNOME's software repository logs, was smaller in an order of magnitude, and is expected to contain less noise [2]. The algorithm we introduced is designed to be more robust to the types of noise found in the mailing list archives than the existing identity merging algorithms, and therefore performs better than these existing algorithms. As the data set grows, more people with the same name will occur in the data; existing algorithms do not take this into account (e.g., aliases containing only a common first name will be matched, generating false positives).

EXISTING ALGORITHMS

We consider three existing identity merging algorithms: Simple algorithm by Goeminne and Mens [3], an algorithm by Bird et al. [4], and our interpretation of Bird's algorithm [2]. After implementing and evaluating performance of our interpretation of Bird's algorithm [2], we acquired Bird et al.'s original code. After evaluating this original code, which we will refer to as *Bird Original*, the results compared to our interpretation of Bird's algorithm were very different. For the sake of completeness, we decided to keep both versions for evaluation.

The simple algorithm bases its merging on the *name* and *email address prefix* using simple heuristics. If two aliases share any of the elements, name or prefix, they are considered as a positive match. Additionally, the algorithm has a threshold *minLen* that filters out short words that would easily match everything together.

Bird's algorithms use more complex heuristics such as splitting the *name* into the first and last names, and comparing names by using the first letter of the first name, concatenated with the last name (e.g., Erik Kouters ⇒ ekouters). These rules are used on both the name and the email address prefix. Additionally, the algorithms use the Levenshtein distance similarity threshold, *levThr*, to allow for differences in names (e.g., as a result of misspelling).

Algorithm	Precision	Recall	F-measure
Simple Algorithm	0.35	0.90	0.50
Bird's Algorithm	0.18	0.92	0.30
Bird's Original Algorithm	0.41	0.90	0.56
Kouters' Algorithm	0.67	0.98	0.80

TABLE I
THE RESULTS OF EACH ALGORITHM ON GNOME'S MAILING LIST ARCHIVES.

THE ALGORITHM

We introduce an algorithm inspired by information retrieval that is more robust to noise and larger data sets. The bigger a data set becomes, the more likely two people with the same first/last name occurs. Because of the heuristics used by the existing algorithms, it is expected they scale badly when the data set grows by an order of magnitude.

The data set, consisting of a list of aliases, is transformed to Vector Space Model, essentially creating a term-document matrix. This matrix is then augmented by the Levenshtein distance similarity used by Bird et al., due to name differences as a result misspelling. To add robustness with respect to frequent names, we apply the term frequency – inverse document frequency (tf-idf) model which scales to the most frequent name. Finally, the similarity between the aliases are computed using the cosine similarity.

The algorithm accepts three different parameters: *minLen* filters out short words, similar to the simple algorithm; *levThr* adds robustness with respect to misspelling, similar to Bird et al.'s algorithms; *cosThr* defines the threshold of similarity when two aliases are considered as positive or negative matches.

EVALUATION

The algorithms described in this article have been evaluated on GNOME's mailing list archives. The preliminary results are shown in Table I. The parameters used for the algorithms are $minLen = 2$ for the simple algorithm; $levThr = 0.8$ for Bird et al.'s algorithms; $minLen = 2$, $levThr = 0.75$ and $cosThr = 0.75$ for Kouters' algorithm. The choice of these parameters was based on earlier research [2]. Table I refers to Kouters' algorithm which is the algorithm described in the previous section.

All algorithms have a high recall on GNOME's mailing list archives. Even the simple algorithm is able to achieve a high recall with its simple heuristics, showing that the people using the mailing lists are consistently using names when sending emails. Furthermore, Bird et al.'s algorithms do not have a much higher recall than the simple algorithm, despite the more complex heuristics. Kouters' algorithm has the highest recall of all algorithms, even higher than Bird et al.'s algorithms. Bird et al.'s algorithms base their heuristics on the first and last names. However, in some cultures it is common to have a name with more than two words (e.g., a Spanish name typically has a first name and two surnames). Better handling of these type of names might improve recall for Bird et al.'s algorithms.

The precision is what the algorithms differ in the most. Low precision is caused by a high number of *false positive* matches. As a result of complex heuristics, our interpretation of Bird's algorithm is very sensitive to false positives: People with the same last name and first letter of the first name are matched (e.g., Aaron Smith, Alex Smith). Kouters' algorithm prevents matching on common names by applying the tf–idf model; names that occur often are decreased in value. This characteristic scales well with a larger data set.

FUTURE WORK

The parameters that were used for the preliminary results presented in Table I were chosen based on earlier research that included a large-scale experiment that tested all combinations of parameters. Doing a similar large-scale experiment on the large data set is considered future work; we do not know how the parameters will affect the results on a large data set. Ideally, an identity merging algorithm has one optimal parameter combination that performs best on all data sets.

REFERENCES

[1] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload – A case study of the Gnome ecosystem community," *Empirical Software Engineering*, pp. 1–54, 2013.

[2] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "Who's who in GNOME: Using LSA to merge software repository identities," in *Proc. ICSM*, 2012, pp. 592–595.

[3] M. Goeminne and T. Mens, "A comparison of identity merge algorithms for software repositories," *Science of Computer Programming*, 2011.

[4] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proc. MSR*. ACM, 2006, pp. 137–143. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138016

Control Flow in the Wild
A first look at 13K Java projects

Davy Landman*, Alexander Serebrenik*†, Jurgen Vinju*
* Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{Davy.Landman, Jurgen.Vinju}@cwi.nl
† Eindhoven University of Technology, Eindhoven, The Netherlands
a.serebrenik@tue.nl

I. Introduction

We are interested in the understandability of software. Maintainability models such as the SIG model use cyclomatic complexity to measure understandability. However, doubts have been raised about the relation between cyclomatic complexity and understanding of the code. In a grounded theory approach we first observe control flow in a large corpus. Which in the long term will enable us to find categories and create well-founded metrics or indicators for understandability.

We present our early observations of Control Flow Patterns (CFPs) [1] in the Sourcerer Corpus [2], a set of 13 thousand Java projects. We observe saturations when CFPs belonging to two or more systems are considered, but no saturation when all patterns are considered. Most observed patterns are unique, only present in one system, moreover they are small, less than 20 statements. We conclude with questions for future research.

II. Experiment

We took the Sourcerer Corpus which contains 18K (13K non empty) Java projects. Using a Java grammar and Rascal [3] we parsed all Java files. All methods were transformed [1] into CFPs.

A CFP is an AST created by removing all statements not related to control flow. Table I contains a list of Java's language constructs kept. The last step is to change all expressions inside the arguments of the constructs into an empty string.

TABLE I
Java language constructs used in a CFP.

if	if else	switch	case	labeled	continue	break
for	while	do while	return	throw	synchronized	try

Table II describes how large the Sourcerer corpus is, and how many CFPs we extracted and how many of those CFPs were unique to one system.

TABLE II
Size of Sourcerer Corpus and extracted CFPs

Size	Files	LOC†	Methods	CFPs	CFPs$^{‡}_{\text{unique}}$
19GB	2M	477M	23M	678K	516K

† measured using `wc -l`
‡ CFPs only observed in one system.

III. Observations

Figure 1 shows the amount of CFPs observed, where we see that almost every time when we add a new systems, we observe new patterns. Narrowing our definition of a pattern, only considering patterns present in 2 or more systems, we observe a saturation. Even more so for patterns shared by 3 or more and 4 or more. Figure 2 shows these narrowed definitions in more detail.

Unique CFPs are patterns only occurring in exactly one system. The almost linear growth in Figure 1 raises the question what causes it. Figure 3 shows that this is not primarily caused by large patterns, that most unique patterns are actually smaller then 20 control flow statements.

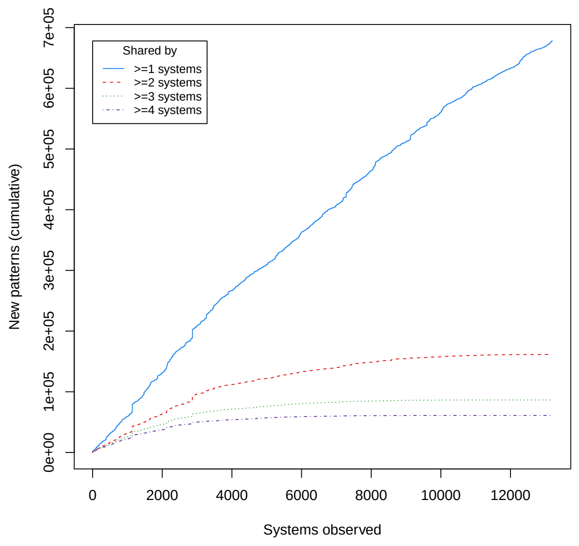


Fig. 1. Saturation of the patterns in the Sourcerer corpus. The four lines represent the saturation of patterns appearing in x or more systems.

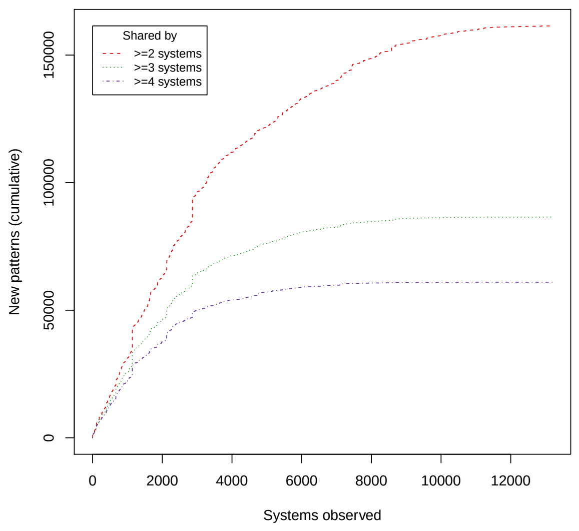


Fig. 2. Zoomed in on the patterns shared in more than one system.

The theoretical reason for so many small *unique* CFPs, is the exponential growth in possible patterns. For size 4 there are already 2.474.634 possible CFPs. Figure 4 shows how many different CFPs per size were observed and it shows the theoretical maximum.

Figure 5 shows the distribution of the size of a CFP and in how many systems it occurs. Here we can see that even the larger CFPs are shared. Eye-balling these larger shared CFPs revealed code clones and code generated by the same generator. We also observed code clones where the full library was embedded.

IV. Open Questions

As future work we have the following questions:

1) Are systems with a lot of CFPs not using OO constructs?
2) Can we find categories of CFPs?
3) Are CFPs abstract enough?
4) Can we find a relation between the naming of a method and it's CFP?
5) If we observe more systems, would the saturation change?
6) If we analyse non Java systems, would we find similar patterns and saturations?
7) What would be the impact of removing clones on the amount of shared patterns?

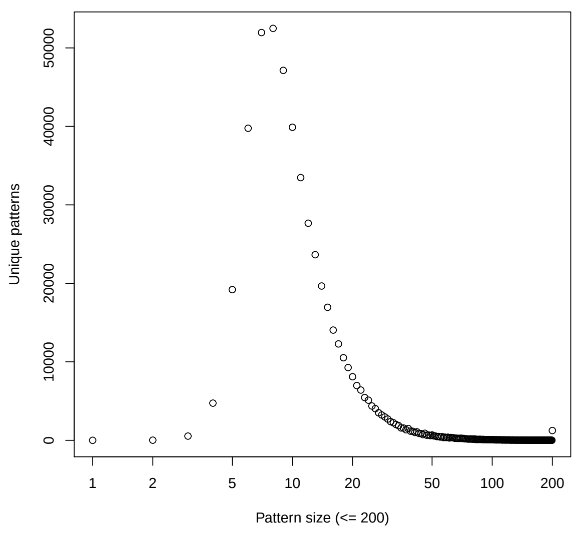


Fig. 3. How many unique patterns of a certain size are found. Patterns equal or larger than 200 are grouped to show that the long-tail does not contribute that much to the continuous growth of patterns observed.

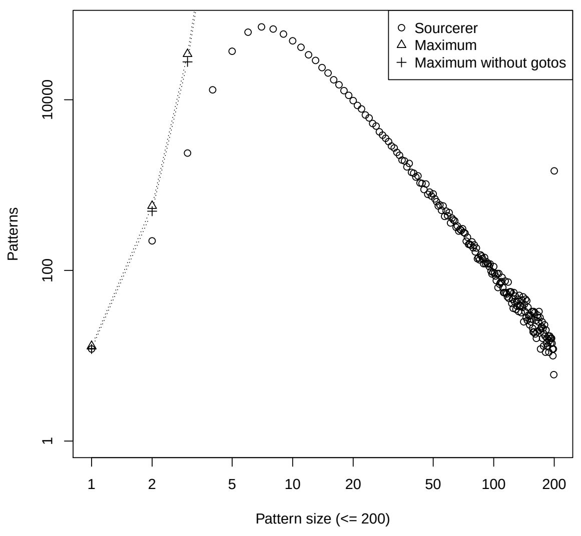


Fig. 4. How many different patterns of a certain size were observed and the maximum possible (only shown for $1 - 3$ due to the exponential growth). The "maximum without GOTOs" does not take into account the "structured" GOTOs in Java.

8) Could CFPs be used to fingerprint systems?
9) Why is there so much control flow in an OO language?

References

[1] J. J. Vinju and M. W. Godfrey, "What does control flow really look like? eyeballing the cyclomatic complexity metric," in *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2012.

[2] E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Min. Knowl. Discov.*, vol. 18, no. 2, pp. 300–336, 2009.

[3] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proc. 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, September 2009, pp. 168–177.

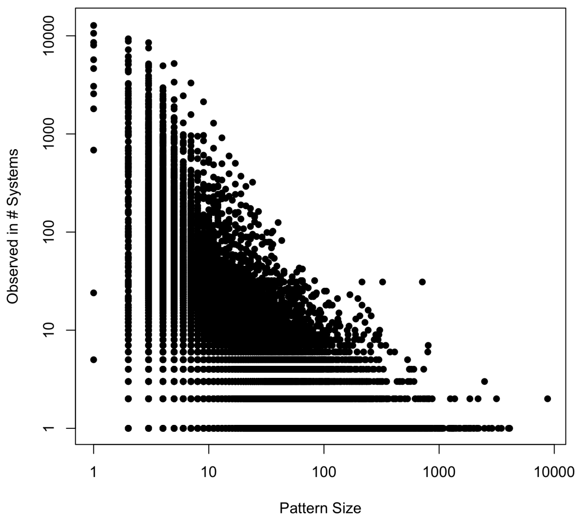


Fig. 5. A scatterplot of the pattern's sizes and in how many projects they occurred.

Analyzing Database Schema Evolution

Loup Meurice
PReCISE Research Center,
University of Namur
Email:loup.meurice@unamur.be

Anthony Cleve
PReCISE Research Center,
University of Namur
Email:anthony.cleve@unamur.be

Abstract

Nowadays, software evolution is an important and sensitive activity becoming ubiquitous and indispensable. Program comprehension is a typical initial phase of software evolution. Over last decades, the research community has largely analyzed software programs evolution. But little research focuses on the analysis of database evolution. Yet software systems are more and more data-intensive and the database often occupies a central place. The goal of our work is to contribute to narrowing this gap and exploring the use of the database evolution history as an additional information source to aid software evolution. We present a systematic tool-supported approach for studying the evolution history of databases and we apply this approach on a complex case study.

I. Setting the context

Understanding the evolution history of a complex software system can significantly aid and inform current and future development initiatives of that system. Software repositories such as version management systems and issue trackers provide excellent opportunities for historical analyses of system evolution. Most research work in this area has concentrated on program code, design and architecture. Fewer studies have focussed on database systems and schemas. This is an unfortunate gap as databases are often at the heart of many of today's information systems. Understanding the database schema often constitutes a prerequisite to understanding the evolution of such systems. Our main objective is recovering a precise knowledge of the evolution history of the database schema because it constitutes an important prerequisite for gaining an understanding of the database. We propose a fully generic tool-supported approach allowing to extract such a historical knowledge from a project's repository.

II. Approach

While our main objective aims at extracting, representing and exploiting the history of a database schema, our approach firstly consists in extracting and comparing the successive versions of the database schema in order to produce the so-called *historical schema*. This historical schema is a visual and browsable representation of the database schema evolution over time. The global process followed by our approach can be divided into steps:

1) SQL code extraction: we first extract all the SQL files corresponding to each system version, by exploiting the versioning system (e.g. Git/SVN repository).
2) Schema extraction: we extract the logical schema corresponding to each SQL file.
3) Historical schema extraction: we build the corresponding historical schema by comparing the successive logical schemas.
4) Visualization & Exploitation: the historical schema can then be visualized and queried to obtain historical information about the database evolution over time.

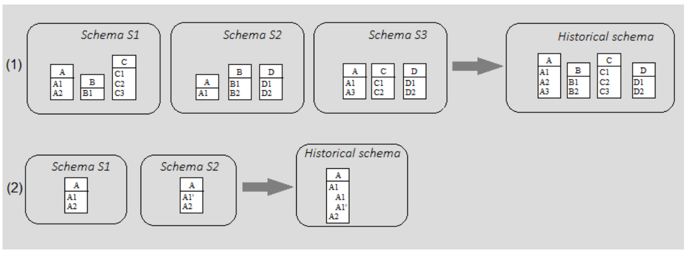


Fig. 1. Two schema evolution examples and their corresponding historical schema.

Figure 1 shows two examples of database schema evolution. The left-hand size of the first example (1) illustrates three successive schema versions. Schema S_1 is the oldest one and schema S_3 is the most recent. We can see that between S_1 and S_2 column A_2 has been deleted, column B_2 has been created as well as table D and its columns. Moreover the entire table C has been dropped. In S_3, table B has disappeared, table D is unchanged, and table C has re-appeared. The historical schema derived from the first example is depicted at the right-hand side. The historical schema is a global representation of all previous versions of a database schema, since it contains all objects that have ever existed in the whole schema history. Furthermore, each object of the historical schema is annotated with the following meta-attributes:

- $listOfPresence$: the list of schema version dates where the object is present.
- $listOfDeletion$: the list of schema version dates where the object has been deleted (an object may have *several lives*.)

The example (2) shows a simpler schema evolution. In S_1, table A has columns A_1 and A_2. In S_2, the two columns are still present but the datatype of A_1 has changed (A_1'). The corresponding historical schema must contain this historical information, and therefore we introduce a new kind of objects called *sub-column*. Each *sub-column* represents a datatype change of the *parent-column*. Moreover, a sub-column is annotated with the meta-attribute $creationDate$ corresponding to the schema version date of the change.
The tool we implemented provides the user with a visual and browsable representation of the database schema evolution over time. It takes the historical schema as input and allows, among others, to (1) compare two arbitrary schema versions, (2) extract the database schema at a given date, (3) extract the complete history of a particular schema object (column/table), (4) extract various statistics about the evolution of the database schema, (5) analyze the involvement of each developer in that evolution. Up to now, our tool allows to detect different low-level categories of atomic database change types occurred between successive schema versions: adding/dropping a table; adding/dropping a column; changing the column datatype; adding/dropping an identifier; adding/dropping an index; adding/dropping a foreign key; renaming a column or table. Concerning the last category, detecting a table/column renaming is an easy process when the SQL migration scripts between successive schema versions are available (SQL parsing). However, in case of absence of those scripts, the task becomes more complex. Indeed, if table A is renamed as B, there is no direct way to detect it and the historical schema will consider that table A has been dropped while table B has been created. In such a case, we see a more refined approach is required. This is why we propose a semi-automatic approach supporting the identification of implicit (column/table) renamings. The name similarity as well as the column type similarity are the main criteria to determine if a table (column) has been renamed. Indeed, we can never be totally sure there is a renming or not. Thus, we have defined a function using those criteria to calculate the similarity probability between two tables/columns. We can formally define the problem of column renaming detection as follows:
Let n successive schema versions $S_1,\ldots,S_n$,
$\forall k \in \{2,\ldots,n\}$,$\forall$ table $T \in (S_{k-1} \wedge S_k)$, we have to find the column renamings that occurred at time k according to a minimal acceptance probability. More formally:
Let $sim : (column \times column) \rightarrow [0,1]$, a function returning the similarity probability between 2 columns, the minimal acceptance probability p,
$c_k = \{c_{k_0},\ldots,c_{k_s}\}$, the set of columns of table T that have been created at time k,
$d_k = \{d_{k_0},\ldots,d_{k_t}\}$, the set of columns of table T that have been dropped at time k,

$$C = \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0t} \\ c_{10} & c_{11} & \cdots & c_{1t} \\ \vdots & \vdots & \ddots & \vdots \\ c_{s0} & c_{s1} & \cdots & c_{st} \end{pmatrix}, \; c_{ij} = \delta_p(c_{k_i}, d_{k_j}) \times sim(c_{k_i}, d_{k_j}) \text{ with } \delta_p(a,b) = \begin{cases} 1, \text{ if } sim(a,b) \geq p, \\ 0, \text{ otherwise} \end{cases},$$

we can express it as an optimization problem (1) with a set of constraints (2):

(1) $max \sum_{i=0}^{s} \sum_{j=0}^{t} c_{ij} x_{ji}$

$$(2) \begin{cases} \forall i \in \{0,\ldots,t\}, \sum_{j=0}^{s} x_{ji} = 1 \\ \forall j \in \{0,\ldots,s\}, \sum_{i=0}^{t} x_{ji} = 1 \\ x_{ij} \in \{0,1\} \end{cases}$$

This a well-known problem solvable with linear programming such as, if $x_{ji} = 1$ and $c_{ij} \neq 0$, then (c_{k_i}, d_{k_j}) is retained as a column renaming by our algorithm. We can define a similar problem for the table renaming detection.

III. A case study: The OSCAR system

OSCAR (*Open Source Clinical Application Resource*) is full-featured Electronic Medical Record (EMR) software system for primary care clinics. It has been under development since 2001 and is widely used in hundreds of clinics across Canada. As an open source project, OSCAR has a broad and active community of users and developers. The OSCAR database schema has over 440 tables and many thousands of attributes. We analysed the history of the OSCAR database schema during a period of almost ten years (22/07/2003 - 27/06/2013). During this period, a total of 670 different schema versions can be found in the project's repository. The earliest schema version analyzed (22/07/2003) includes 88 tables, while the latest schema version considered (27/06/2013) comprises 445 tables. After having generated the corresponding historical schema, we extracted some interesting statistics regarding the evolution of the schema: the evolution of the number of tables/columns over time, the stability of the tables (a table that has been created a long time ago, and that was not subject to frequent modifications can be considered stable), the detection of the database specialists (we observe than the few most active schema committers have hardly touched 20% of the OSCAR tables), ... Thanks to those statistics, we can answer to some primordial evolution questions such as "Which tables seem to be the most sensitive to evolve?", "Who are the specialists of that part of the database?", "How do database schemas evolve? What are the most common evolution patterns that can be observed".

IV. Conclusion

We present a tool-supported process that allows us to analyze the evolution history of a database over its lifetime. The method is based on the automated derivation of a global historical schema, that includes all the schema objects involved in the entire lifetime of the database, each annotated with historical and temporal information. While this work only makes a first humble step towards the understanding of large database evolution histories, it also opens several important new research and collaboration perspectives for the entire software evolution research community. Indeed, considering the link between the evolution of the database and the evolution of all the other software artefacts remains a largely unexplored yet important research domain.

Acknowledgements. This work extends our joint research results with Jens Weber, Maxime Gobert and Jerome Maes[1] [2]. Loup Meurice is supported by the F.R.S.-FNRS in the context of the DISSE project, in collaboration with the University of Mons.

[1]Understanding Schema Evolution as a Basis for Database Reengineering. Gobert, M., Maes, J., Cleve, A. & Weber, J. 2013 Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013). IEEE Computer society

[2]Understanding Database Schema Evolution: A Case Study
Cleve, A. , Gobert, M., Meurice, L. , Maes, J. & Weber, J. 2014: Science of Computer Programming.

Towards an empirical analysis of the maintainability of CRAN packages

Maëlick Claes
COMPLEXYS Research Institute,
University of Mons
maelick.claes@umons.ac.be

Tom Mens
COMPLEXYS Research Institute,
University of Mons
tom.mens@umons.ac.be

Philippe Grosjean
COMPLEXYS Research Institute,
University of Mons
philippe.grosjean@umons.ac.be

Abstract

The R ecosystem has been the subject of study by several researchers. In this presentation, we focus on the CRAN archive, containing all packages contributed by R developers. This software ecosystem contains over 5000 packages being actively maintained by over 2500 maintainers. Through an empirical analysis of this ecosystem, we aim to understand the factors that affect the reliability and maintainability of CRAN packages. To do so, we use the built-in and automated CRAN check mechanism that daily checks for different sources of errors in contributed packages. According to the imposed policy, package developers have the obligation to fix these errors, if not their package will get automatically archived after a certain amount of time. Based on an analysis of the package dependencies, characteristics, of the package maintainer and the target platform, we present preliminary results on the sources of errors in each package, and the time needed to fix these errors. With such a study, we aim to get a better insight in the main factors that cause packages to become more reliable, which is not only beneficial to the package maintainer itself, but to the ecosystem as a whole.

I. Introduction

R is a GNU project providing a complete free statistical environment. It provides a package system where packages can contain code (written in R, C or any other programming language), documentation, scripts, tests, data sets, and so on. Those packages can also depend on other R packages which can be retrieved automatically and recursively.

CRAN, the Comprehensive R Archive Network, stores such R packages using different HTTP and FTP mirrors. Its history goes back to 1997. It is managed by the R core team and currently contains over 5000 packages and libraries. While other R package repositories exist (e.g., Bioconductor, R-forge), CRAN is the official, oldest and biggest repository for R packages.

The evolution of the R ecosystem has been analyzed in [1]. The problem has been raised that there are too many R packages, and that the number of contributed packages is growing exponentially [2]. There are also many challenges with the way package dependencies are managed in CRAN, especially in the light of the maintainability of packages [3].

This problem is becoming a major issue inside the R community where the CRAN package repository has grown to a size of more than 5000 packages and where the community is regularly complaining on difficulties of package maintenance because of dependencies complexity. In this talk we will present our early results on analysing the dependencies of the CRAN R packages repository. We aim both at showing empirical evidence of the impact of errors spreading through dependencies and at providing R package maintainers objective information giving more insight in which packages are safe to reuse and why.

This could be for a variety of reasons: because the package they want to reuse (i.e., depend on) has been quite stable over time, implying perhaps that it will continue to remain stable in the future; because the package has been maintained by a reliable developer (e.g., someone that is also maintaining many other packages in the CRAN, or a developer that is very (re)active), or because the package has encountered little or no bugs in the (recent) past.

II. On the evolution of CRAN packages

CRAN has a strict policy[1] and packages have to pass a check before being accepted in the repository. Packages are checked for different configurations called *flavors*[2]. A flavor is a combination of operating system, hardware, R version and compiler. The checking process is regularly reapplied on the whole

[1] http://cran.r-project.org/web/packages/policies.html
[2] http://cran.r-project.org/web/checks/check_flavors.html

set of packages in order to test if they still pass it. Indeed, if one package is updated, other packages depending on it may not work anymore if the functions they were relying on have changed.

Even if CRAN provides on its packages web page the list of reverse dependencies and even if the CRAN policy recommends to take care of not breaking those reverse dependencies when updating a package, such dependency breaks are unavoidable. Packages can also fail the check if a new R release introduces, changes or removes features.

Packages are never removed from CRAN but archived instead. Upon release of a new R version, maintainers have to take care that their packages still pass the checking process. If this is not the case, and if maintainers don't promise to fix the problems quickly, they will be archived when the next non-minor R version will be released unless maintainers promise and do solve errors before a stated deadline.

One major issue with R and CRAN is that two versions of the same package cannot be installed at the same time and that archived packages cannot be installed automatically. Thus, if a package maintainer decides for various reasons to stop maintaining its packages, dependency breaks will appear in other packages relying on it, creating a ripple effect in the ecosystem. This is a major issue for replicability of statistical research. It is a known problem in the R community and solutions have already proposed to solve the problem [3]. These proposed solutions are the use of distributions like it is done in the Linux world or a version packaged management like *Node.js* package manager *NPM*.

III. Research Questions

In order to improve our comprehension of the aforementioned problems we have started to analyse the CRAN packages over time, with the aim to answer the following research questions:

- What is the source of errors in CRAN packages?
- Are package dependency errors more likely to occur if the maintainer is different for both packages?
- How are errors fixed in CRAN packages and how long does it take to fix them?
- What are the characteristics of a maintainer that makes him more likely to introduce package dependency errors?
- What are the characteristics of a package that makes it more vulnerable to package dependency errors?
- How does the use of a specific flavor impact errors in CRAN packages?

During our presentation, we will explain how we are trying to address each of these questions.

References

[1] D. M. Germán, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *European Conf. Software Maintenance and Reengineering*, 2013, pp. 243–252.

[2] K. Hornik, "Are there too many R packages?" *Austrian Journal of Statistics*, vol. 41, no. 1, pp. 59–66, 2012.

[3] J. Ooms, "Possible directions for improving dependency versioning in r," *R Journal*, vol. 5, no. 1, pp. 197–206, Jun. 2013.

Fragility of Evolving Software

Kim Mens
ICTEAM,
Université catholique de Louvain,
Louvain-la-Neuve,
Belgium

Software systems are fragile with respect to evolution. They consist of many software artefacts that make implicit assumptions about one another. When such artefacts get replaced by newer versions, some of these assumptions may get invalidated, thus causing subtle evolution conflicts. We refer to this phenomenon as *fragility of evolving software*.

A particular instance of fragility in class-based object-oriented programming languages is the *fragile base class problem* [1], [2]. It occurs when a base class (a class from which other classes are derived through inheritance) gets replaced by a newer version. When the derived classes make certain assumptions about the base class that the evolved base class no longer provides, this can cause subtle conflicts. E.g., suppose that a base class *B* implements a method *m* in terms of an auxiliary method *n*. Now suppose that a derived class *D* of *B* overrides the implementation of *n*, with the intention not only of adapting the behaviour of *n*, but also that of *m* which is defined in terms of *n*. Independently, however, the base class *B* evolves into a newer version *B'* where *m* no longer depends on *n* for performance reasons. After this evolution, *D*'s assumption that *m* depends on *n* is no longer valid, and *D*'s overridden implementation of *n* no longer affects *m*, thus causing an unexpected behavioural conflict.

Another instance of fragility is the *fragile pointcut problem* in aspect-oriented programming [3], [4]. It may occur upon evolution of an aspect-oriented program, which contains pointcuts expressed over a base program. Pointcut expressions express execution points, called join points, in the base program where the aspects need to be applied. Pointcuts are fragile, because the base program is oblivious of their existence. When the base program evolves, this may have unexpected effects on the pointcuts expressed over that base program (such as accidental captures or misses of join points). *Advice fragility* is a related problem caused by the obliviousness of aspect code with respect to base code [5]. It arises when the advice code (i.e., the aspect code that will be woven with the base code at the join points) is too tightly coupled to the base code. This may cause problems when the base code evolves in such a way that the advice code no longer fits with that base code.

In general, fragility problems arise when the assumptions made by a software artefact about another artefact get invalidated upon evolution of that other artefact. In a sense, those assumptions, which are often not documented explicitly, constitute a kind of implicit contract to be respected upon software evolution. Solutions to the fragility problem therefore typically involve providing a means to define such an *evolution contract* explicitly, detecting possible breaches of that contract upon evolution, classifying the possible conflicts that may arise when the contract gets breached, and proposing appropriate solution strategies for each of those possible kinds of conflicts.

For example, as a solution to the fragile base class problem exemplified above, *reuse contracts* [2] document the so-called *specialisation interface* [6] of the base class, i.e. its internal calling dependencies, as well as how the derived class depends on the base class. In our previous example, the reuse contract would document, amongst others, that in base class *B* method *m* calls *n*, and that the derived class *D* 'refines' *B* by overriding the implementation of *n*. Furthermore, an evolution operator would describe that *B'* 'coarsens' *B* by removing an internal calling dependency (method *m* in *B'* no longer calls *n*). A two-dimensional classification of possible *reuse conflicts* can then be made in terms of the evolution operators and derivation dependencies. E.g., the above conflict where the base class coarsens a method *m* by removing its dependency on *n*, whereas the dependent class overrides *n*, would be flagged as an 'inconsistent method' [2]. Based on this classification, for each type of conflict[1] a corresponding solution strategy can be proposed. E.g., for the inconsistent method above either the method *m* in *B'* should keep its dependency on *n* or, alternatively, *D* should be replaced by a *D'* that overrides not only *n* but also *m*.

More recently, *usage contracts* [7] were proposed as an alternative mechanism to solve some of the problems caused by base class fragility. Usage contracts explicitly document the expected structural

[1]Other possible conflicts are conflicting method interfaces, unimplemented methods, or accidental method captures. [2]

regularities that a base class wants its dependent classes to conform to. These regularities can be verified automatically when implementating or modifying the dependent classes, flagging potential conflicts immediately so that they can be corrected as soon as they arise.

In the case of the fragile pointcut problem, pointcuts are fragile w.r.t. evolution of the base code. A possible solution [3] consists in making the pointcuts more robust by declaring them in terms of an intermediate pointcut model, rather than directly in terms of the base code. This model forms a kind of evolution contract between the base code and the pointcut expressions, by making explicit some of the assumptions the pointcuts make about the base code. Furthermore, it enables detecting, after evolving the base code, whether the base code still satisfies the pointcut model. If it doesn't, the mismatch between the base code and the model gives an indication of potential mismatches (accidental or missed captures) the pointcuts have with respect to the base code, and appropriate solutions can be proposed (by adapting either the base code or the pointcuts that refer to it).

The examples above illustrate only some instances of fragility problems and their solutions. Our claim is that the problem of fragility applies in general to any kind of adaptable systems, and so do the proposed solution approaches. The problem always amounts to a lack of documentation on the implicit assumptions between a base entity and its dependent entities, and the solution always involves documenting these assumptions more explicitly in a kind of evolution contract, and verifying upon evolution whether the contract remains respected. By classifying the types of possible conflicts (depending on the kind of evolution and the kind of dependency) and their corresponding solution strategies, appropriate solutions to these conflicts can be proposed.

In the near future we will investigate the problem of fragility in the area of dynamic software evolution, and context-oriented programming in particular. In fact, this example will be worked out in more detail in a separate contribution submitted to this BENEVOL2013 seminar.

References

[1] L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem," in *Proceedings of ECOOP'98*. Springer-Verlag, 1998, pp. 355–382.

[2] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt, "Reuse contracts: Managing the evolution of reusable assets," in *Proceedings of OOPSLA'96*. ACM Press, 1996, pp. 268–285.

[3] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *Proceedings of ECOOP'06*. Springer-Verlag, 2006, pp. 501–525.

[4] C. Noguera, A. Kellens, D. Deridder, and T. D'Hondt, "Tackling pointcut fragility with dynamic annotations," in *RAM-SE'10 workshop at ECOOP'10*. ACM Press, 2010, pp. 1–6.

[5] A. Cyment, N. Kicillof, R. Altman, and F. Asteasuain, "Improving AOP systems' evolvability by decoupling advices from base code." in *RAM-SE'06 workshop at ECOOP'06*, 2006, pp. 9–21.

[6] J. Lamping, "Typing the specialization interface," in *Proceedings of OOPSLA '93*. ACM Press, 1993, pp. 201–214.

[7] A. Lozano, A. Kellens, and K. Mens, "Usage contracts: Offering immediate feedback on violations of structural source-code regularities," under revision.

User Interface Evolution: Machine learning for GUI at runtime contextualisation

Nesrine Mezhoudi
Universit catholique de Louvain LSM,
Louvain Interaction Lab.
Place des doyens,
1B-1348 Louvain-la-Neuve,
Belgium.
Nesrine.mezhoudi@uclouvain.be

Jean.vanderdonckt
Universit catholique de Louvain LSM,
Louvain Interaction Lab.
Place des doyens,
1B-1348 Louvain-la-Neuve,
Belgium.
Jean.vanderdonckt@uclouvain.be

Abstract

Adapting user interfaces in response to changing user preferences in the executing environment enhance the user experience, mainly by improving their interaction and reducing their errors. According to the Standish Group, the User Involvements was graded in the top 3 list of IT project success/failure/challenged rates since 1994 till 2013 in the CHAOS Report [1]. However, given the significant variability of user needs and expectations, adapting UIs often demands complex inferences and strategies for acquiring and considering up-to-date contextual data. Thus adaptation should have a cross-cutting impact on software patterning and appearance depending on the situation and the ambient-context with an insignificant cost [2], [3]. However, adaptation stills a challenge in the HCI field since there is no arranged technique for greatest adaptation.

To perform user-centred adaptation, interaction should be considered as well as different user backgrounds. In fact several works investigate the user profiles in term of their interests, culture, expertise. In this way, considering the user profiles allows the system to benefit from their characteristics as a supplemental user-related fact, which seems promising to enhance the end users influence in the UI definition.

Within this context, Machine Learning (ML) techniques seem appropriate to give the system the opportunity of adapting the UI according to user preferences and interventions. To aid the adaptation decisions, ML algorithms can be applied to support reasoning, to optimize the adaptation process, to deal smartly with the contextualization [3], [4] and to ensure a high predictability precision among systems. We aim mainly to ensure an UI adaptation based on Machine Learning and user intervention. It is based on taking advantage from users feedbacks during the interaction to reinforce existing adaptation rules, besides the extraction of new supplied acquaintance for the UI personalization. The challenge is to investigate how ML algorithms manifest themselves to ensure an adaptive learning during user interaction. According to this perspective, performing the adaptation learning among the software requires an enhancement to deal smartly with adaptation according to user preference by mixed initiative (User and System).

Machine learning has as main advantages the ability to sense the context in a dynamic manner to evolve the adaptation engines, providing users more suitable adaptations. Then, mainly the learning infrastructure fulfills two general requirements. First gathering the changing preferences of the user (with explicit and implicit feedback and the interaction history for instance). We assume that perceived events increase the system predictability to perform adaptations fitting the end-user expectations at runtime. And then adapting the engine by adjusting parameters related with priorities or preferences. Such controllability allows the user to keep the improved graphical interface in the tolerated space of interfaces but moreover it provides control tools to keep under control an iterative adaptation process.

Due to the large availability of technological devices, it is each day more important to implement and provide users applications able to evolve and effectively adapt themselves to the user needs. The complexity of implementation of such task is basically due to the large amount of involved context information. Therefore, ML algorithms can efficiently support this task. ML has as main advantages the ability to sense the context in a dynamic manner to evolve the adaptation engines, providing users more suitable adaptations.

The user experience recapitulates the end user preferences, which need to be explicitly transmitted to systems. The system evolution is ensured via a runtime GUI adaptation based on Machine Learning and user intervention.

As showed in Figure 1, The UI evolution capitalize on ML techniques to handle a training phase, which involves an upgrading the pre-existing set of adaptation rules according to the user experience

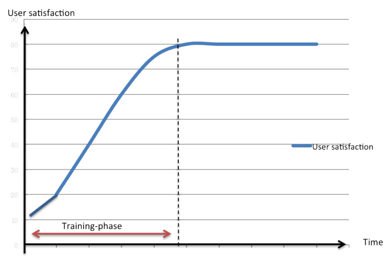


Figure 1. Estimation of user-satisfaction degrees by time

with the system. It is based on taking advantage from users feedbacks during the interaction to reinforce existing adaptation rules, besides the extraction of new supplied acquaintance for the UI personalization. Moreover we believe that there is an opportunity to achieve significant improvement by mixing adaptive and adaptable behaviors during interaction with individuals, and we were thus motivated to pursue users interfaces that generate personalized interfaces instead of treating all users the same. The mentioned training phase is intended to outline a monotonic function drawing up the increase of user satisfaction degrees by time. The variance should show a system jump from the adaptive mode to proactive one by the end of the training phase.

REFERENCES

[1] C. F. Caroll, “It success and failure the standish group chaos report success factors,” 2013.
[2] D. Barber, *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2010.
[3] V. G. Motti, N. Mezhoudi, and J. Vanderdonckt, “Machine learning in the support of context-aware adaptation. proceedings of the workshop on context-aware adaptation of service front-ends,” 2012.
[4] Mitrovic, J. A., N. Royo, and E. Mena, “Adaptive graphical user interface: An approach based on mobile agents,” 2009, pp. 1–36, networks.

Generic Pushdown Analysis and Program Querying of Higher-Order Languages

Jens Nicolay Coen De Roover

Pushdown analysis is an improvement over classic finite-state analysis in terms of precision and performance, especially when analyzing recursive programs. However, implementing a pushdown analysis from scratch is time-consuming and technically challenging. JIPDA is a framework that takes an abstract machine representing program semantics, and produces a pushdown analyzer for the set of programs that that abstract machine can interpret. Besides the implementation of the actual pushdown system and summarization algorithm, JIPDA also provides several other extendable and reusable components for quickly creating pushdown analyses. The result of a pushdown analysis is a graph in which the nodes are machine states and the edges represent stack changes. Program properties can then be expressed as queries over this graph.

The starting point for defining a pushdown analysis is the definition of an abstract state machine that expresses the program semantics of interest. The only link between the abstract machine and JIPDA is through an abstract stack provided by the framework when exploring successor states. Valid stack actions for reaching a successor state are either pushing or popping frame, or leaving the stack unchanged. In order to have a finite pushdown system (possibly expressing an infinite state space), the abstract machine may only generate a finite number of states and frames. This can be accomplished by using abstract values and operations instead of concrete ones, organized in a lattice. JIPDA provides several abstract machines for interesting subsets of Scheme and JavaScript. These machines are expressed as CESK machines, where the "K" component is the abstract program stack provided by the framework. JIPDA also provides several lattices that can be used by abstract machines. The framework also defines a store with abstract counting, and an abstract garbage collector for abstract machines that wish to implement garbage collection by removing unreachable addresses from a store.

The result of pushdown analysis is a Dyck state graph (DSG), which is a state graph with only reachable machine states and transitions. Machine states are the nodes of the graph, while the edges are transitions corresponding to the possible stack actions (push, pop, or unchanged). Every path from the initial state to another state will be a legal path consisting of balanced pushes and pops of frames, possibly intermixed with unchanged or "epsilon" transitions. The DSG also contains summary edges, which are epsilon edges that connect states between which there exists a path in which all pushes and pops cancel each other out (i.e. there is no net stack change).

A DSG is in fact a finite representation of the runtime behavior of a program from which it was constructed. As such it is possible to derive program properties by performing graph queries on a DSG. For example, by starting from a certain state and repeatedly following incoming push-edges or epsilon edges, one obtains an NFA that describes all possible stacks at that state. By collecting all the pushed frames from that NFA in a set, one obtains all possible frames on the stack, necessary for example for computing the root set when performing garbage collection. Other examples include obtaining the values produced by an evaluation state (value flow), especially when it concerns the value of an expression in operator position (control flow), jumping to declaration sites of identifiers (skipping nested scopes using epsilon transitions), etc. On these low-level, reusable graph queries, more abstract graph queries can be defined.

JIPDA is already being used in an academic setting. It is used for obtaining program depence graphs for JavaScript programs, as the basis for performing JavaScript model checking, for detecting pure functions in JavaScript, and for implementing and assessing the effect of (abstract) memoization as an optimization of a Scheme CESK. In the future we want to express DSG queries using logic meta-programming, so we can for example declaratively express preconditions of refactorings.

Effort estimation based on Mining Software Repositories - A Preliminary Approach

Carlos Cervigóni
Universidad Rey Juan Carlos
Madrid (Spain)
c.cervigon@alumnos.urjc.es

Gregorio Robles
Universidad Rey Juan Carlos
Madrid (Spain)
grex@gsyc.urjc.es

Andrea Capiluppi
Brunel University
London (United Kingdom)
andrea.capiluppi@brunel.ac.uk

Abstract

A commercial company participating in an free/libre/open source project has a clear view of the effort it applies to the projects it chose to contribute to. The effort by other companies contributing to the same project, or by the volunteers participating in it, should be quantified, so that any company could evaluate a return-on-investment on their devoted effort.

We propose a new way of estimating effort based on the activity of the developers by mining software repositories. This approach is focused on the development of free/open source projects, where there may be companies and volunteers collaborating and information of effort allocated is difficult to obtain by other means.

Keywords

effort estimation; mining software repositories; software evolution; free software; open source;

I. Introduction

In this extended abstract we propose a different approach to effort estimation from the one that has been developed so far in the related literature [1]. Instead of predicting the future effort of a software development, our approach focuses on the estimation of the how much effort has been produced for a given project. If feasible and accurate, this method could be adequate for some scenarios, such as in free/libre/open source software (FLOSS) projects: in particular, it could be used to estimate the return-on-investment of the effort devoted by a company, when contributing to a FLOSS project.

It should be noted that in traditional settings, the effort devoted (to an activity or a project) is known to the company from its records: in these cases, the company has an interest in predicting the future effort based on the historical records. However, and particularly in FLOSS development scenarios, the effort utilised in the activities is not known a-priori, since the stakeholders participating in the project may be volunteer developers or, for sponsored projects, developers affiliated to a company [2]. Companies participating in a FLOSS project are in general aware of the effort they put into the projects in both monetary and human terms. On the other hand, the same companies cannot clearly quantify the effort applied by the rest of the community, including the other companies involved in their same FLOSS project.

Our approach is based on estimating the effort by characterizing the developer activity [3], in particular using data obtained from the *git* repositories. When mining a repository, it is straightforward to obtain "when" developers performed their commits: an open question that the current repositories cannot answer is "how long" the developers devoted to these commits. This missing information could be used to formulate a first draft of the effort devoted by an individual to each commit.

In our approach, we assume that we can identify the developers affiliated to a specific company, and that those developers are full-time committed to the project. These developers are assigned an effort equivalent to 40 hours per week, although we are trying other timespans to include days off, vacations and other circumstances where developers cannot be tracked in the repository, but are still working on the project.

Assuming that we can identify these developers and that we can estimate the effort from the rest of developers from their activity in the versioning system, we expect to have a good estimation of the effort that has been devoted to the project.

II. Benefits

Our approach could benefit companies different scenarios. In the following list we are proposing a few:

- *Return-on-investment*: a company knows well how much effort it has invested in the project, but it is likely that they are not completely aware of the total effort that all participants have devoted in the project (or in the part of the project where the company is involved).
- *Open process visibility*: FLOSS foundations, such as the ones for Mozilla, GNOME or OpenStack, are very interested in these figures and quantitative analyses, since one of their goals is to promote the visibility and openness of their processes while attracting new developers and companies [4].
- *Fairness of contributions*: the involvement of commercial companies in FLOSS projects has been always been a very sensitive issue, as volunteers and other participating companies demand a "fair" collaboration environment [5], [6].
- *Effort-saving synergies*: when a patch is accepted by a FLOSS project, the responsibility of its maintenance and future evolution is shifted from the original authors to the overall project. For a commercial company, getting a contribution accepted in "upstream" may be considered as an added value, that reduces the future effort in maintaining the same patch. Therefore, a company could benefit from such analysis, by measuring the effort saved when any of its development becomes "upstream".
- It offers valuable strategic information for the project.

III. Methodology

The methodology that we are following is composed of following steps that are briefly described:

1) Extract log information from the versioning system.
2) Identify authors and committers from the activity. We can use in this step, ideas and methods used for the classification of the Linux kernel developers presented by Capiluppi et al. [7].
3) Merge author information [8], as authors may use several identities.
4) Assign authors to companies.
5) Identify professional authors (and assign them 40 hours per week or any reasonable measure for a full-time developer).
6) Obtain an estimation of the effort for the rest of developers. This could be done by clustering the rest of contributors and transforming their activity to effort, but we are currently trying several approaches.
7) Obtain set of metrics: total sum of effort, effort by companies, ratio affiliated versus volunteer developers, RoI, etc.
8) Evaluate the activity to effort transformation by means of a survey of developers.
9) Compare the results obtained with traditional models, such as COCOMO and COCOMO II [9], or previous studies that have researched similar issues, such as "What does it take to develop a million lines of open source code?" [10] or "Indirectly predicting the maintenance effort of open-source software" [11]. .

Acknowledgments

The work of Gregorio Robles has been funded in part by the Spanish Government under project SobreSale (TIN2011-28110).

References

[1] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 33–53, 2007.
[2] B. Fitzgerald, "The transformation of Open Source Software," *Mis Quarterly*, pp. 587–598, 2006.
[3] J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona, "Effort estimation by characterizing developer activity," in *Proceedings of the 2006 international workshop on Economics driven software engineering research.* ACM, 2006, pp. 3–6.
[4] D. Riehle, "The economic case for Open Source foundations," *Computer*, vol. 43, no. 1, pp. 86–90, 2010.
[5] L. Dahlander and M. G. Magnusson, "Relationships between open source software companies and communities: Observations from Nordic firms," *Research Policy*, vol. 34, no. 4, pp. 481–493, 2005.
[6] E. Capra, C. Francalanci, and F. Merlo, "An empirical study on the relationship between software design quality, development effort and governance in Open Source Projects," *Software Engineering, IEEE Transactions on*, vol. 34, no. 6, pp. 765–782, 2008.
[7] A. Capiluppi and D. Izquierdo-Cortázar, "Effort estimation of FLOSS projects: a study of the Linux kernel," *Empirical Software Engineering*, vol. 18, no. 1, pp. 60–88, 2013.

[8] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. van den Brand, “Who’s who in GNOME: Using LSA to merge software repository identities,” in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 592–595.

[9] B. W. Boehm, R. Madachy, B. Steece *et al.*, *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, 2000.

[10] J. Fernandez-Ramil, D. Izquierdo-Cortazar, and T. Mens, “What does it take to develop a million lines of Open Source code?” in *Open Source Ecosystems: Diverse Communities Interacting*. Springer, 2009, pp. 170–184.

[11] L. Yu, “Indirectly predicting the maintenance effort of Open-Source Software,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 5, pp. 311–332, 2006.

Software Forks: A new Challenge for Software Evolution Research

Gregorio Robles
Universidad Rey Juan Carlos
Madrid (Spain)
grex@gsyc.urjc.es

Jesús M. González-Barahona
Universidad Rey Juan Carlos
Madrid (Spain)
jgb@gsyc.urjc.es

Abstract

Software evolution usually happens within the authoring organization. Even in most free/libre/open source software (FLOSS) projects, developers (and companies) organize themselves around the project, being responsible for the evolution of the project. However, FLOSS licenses allow to create your own branch of the project and make it evolve independently of *main* one; a *fork* is created. Although forking has not been seen positively by the FLOSS community, many forks can be found in the last twenty years. A fork has as a consequence the parallel development of the same source code base and raises some questions that have not attracted yet much interest from the software engineering research community. In this extended abstract we want to point out the relevance of forking for the software industry, and enumerate some research questions that we think would be interesting to address.

Keywords

software forks; software evolution; free software; open source;

I. Introduction

Research on software evolution has always assumed that a software project is performed under the umbrella of an organization or a group of developers that drive the project. Nonetheless, the fourth of Lehman's *laws* is labelled "Conservation of Organisational Stability" and states that there is an invariant work rate: "The average effective global activity rate in an evolving E-type system is invariant over product lifetime" [1].

However, in the free/libre/open source (FLOSS) world there exist cases where a software project evolves in an unusual manner: two different developer groups take a version of the source code and make it evolve independently. This is known as "forking", and is embedded into the freedoms that FLOSS licenses grant the user, as modification of the software and redistribution of these modifications (without permission needed from the original author(s)) is possible.

Although traditionally the FLOSS community has been very unfavorable to forking due to various reasons (many argue that a fork splits the community introducing *bad* mood, that effort is duplicated or that incompatibilities among the software and its formats arise), historically forking has been not infrequent as previous research by the authors has shown [2]. We have counted up to 220 significant forks in the last 20 years, including some very well-known ones: OpenOffice.org and LibreOffice, the BSD family (FreeBSD, NetBSD and OpenBSD), Emacs and XEmacs or X.org and the X server, among others. Almost any major FLOSS project has suffered in its history an episode of forking risk. Recently, Google forked the Webkit project, where it collaborated with Apple, Samsung and other large IT companies, to create Blink, its own web browser engine. Our research has shown that most of the forks evolve in parallel to the original project and that, in contrast to our initial assumption, the chance of discontinuation of a fork is almost the same as the original, even if they have a disadvantageous starting situation. Another interesting result is that software projects that are forked ensure sustainability for the users, because in very few cases neither the fork project nor the original one get both discontinued; at least one of them survives in the long term. Several re-merges have been identified, but their number is very low. When a re-merge happens, integration of both source code bases is a problem, and often one of them is dismissed.

Due to the main role that FLOSS projects have achieved in today's society and in the software industry, it is the opinion of the authors that forking is going to be a relevant and possibly frequent situation in the future. However, almost no research has been performed on this issue.

Noteworthy is the fact that the technological advance is making the creation of forks easier. The ample use of distributed versioning systems such as `git` and `Mercurial` facilitate the task for doing a fork. Platforms such as GitHub promote forking, and see in it a way of boosting innovation. In many cases, these forks result in a collaboration with the source project, but many times the fork results in a software evolving in parallel.

II. Research questions

There are several research questions related to software forks that we would like to address in the near future:

- What type of forks exist? Can we find forks that collaborate more with the source project, even if not directly?
- Is forking a zero-sum game? Forking has historically been seen by the FLOSS community as a situation to avoid, because all players lose. Is this true? To what extent do some win and others lose?
- Are merges of forks possible? If yes, in how far are they possible? What are the difficulties?
- What lessons can be learnt from the study of historic forks?
- How much do original and forking projects collaborate, even in an involuntary way, by exchanging code, bug reports and fixes?
- Parallel software development should be devoted a closer look; how should technology and processes facilitate integration and co-evolution of two parallel evolving projects?
- How does the community move when a fork occurs? This would include among others answering questions such as where the key developers go, how many of the developers are active both in the original and forking projects, or if there is any correlation between a *positive resolution* of the fork and who pushed for it.
- How does the socioeconomic and technical context influence the probability and the type of a fork? So, for instance, if project maturity boosts forking or not.
- Are projects led by software companies more prone to have software forks? How can companies avoid such forks?

Acknowledgments

The work of Gregorio Robles and Jesús M. González-Barahona has been funded in part by the Spanish Government under project SobreSale (TIN2011-28110).

References

[1] M. M. Lehman and L. A. Belady, Eds., *Program evolution: Processes of software change*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[2] G. Robles and J. M. González-Barahona, "A comprehensive study of software forks: Dates, reasons and outcomes," in *Open Source Systems: Long-Term Sustainability*. Springer, 2012, pp. 1–14.

Visualizing the Complexity of Software Module Upgrades

Bram Schoenmakers,
Niels van den Broek,
Istvan Nagy
ASML Netherlands B.V.,
The Netherlands
{bram.schoenmakers, niels.van.den.broek, istvan.nagy}@asml.com

Bogdan Vasilescu,
Alexander Serebrenik
Technische Universiteit Eindhoven,
The Netherlands
{b.n.vasilescu, a.serebrenik}@tue.nl

I. Introduction

Modern software development frequently involves multiple codelines (branches), being canonical sets of source files required to produce a specific software instance. Codelines correspond, *e.g.,* to maintenance, release and development versions of a system, or to variants targeting different user groups or platforms. A typical scenario involves one mainline, containing the latest features and bug fixes and being continuously updated, and a number of customer codelines, containing different configurations being used by different customers. Such customer codelines need to be updated frequently, *e.g.,* to provide new features or bug fixes. The updates then translate to patches or entirely new releases, which are shipped to customers and have to be integrated into their environments.

However, integrating the patches on the customer-side (referred to as upgrading) can become particularly costly, especially for safety-critical or real-time embedded software, that require extensive integration testing and complex initialisation routines. Consequently, large upgrades (*e.g.,* upgrading the entire codeline to a new release) are typically undesirable, and performing upgrades in a module-based fashion is preferred.This way, customers receive only the features they requested. Moreover, as less changes are introduced modular upgrades reduce testing and integration effort, as well as limit the risk of the system downtime.

Nonetheless, modules are often interdependent, hence upgrading a particular one may introduce the need to upgrade additional others, until all dependencies have been satisfied.

In [1] we have proposed a framework to assess the difficulty of upgrading a software module. In this extended abstract we discuss the visualization part of the framework and the framework evaluation at ASML Netherlands B.V., a large manufacturer of photolithography systems.

II. Minimal Upgrades

Dependencies arise as a result of some modules providing interfaces to be used by other modules. Each interface can disclose various program elements, which we call *symbols*, such as constants, enumerations, data types and functions. Each symbol can be provided by only one module.

Consider the three system states in Figure 1. Suppose that the system is initially at $t = 1$ and we would like to upgrade module A to the version at $t = 3$. Between these two versions, A has started to depend on symbol S of module B, which is not available at $t = 1$. Therefore, module B has to be upgraded as well. Here we have a choice: upgrade module B to the version at $t = 2$ or $t = 3$. Both versions provide the symbol S required by the latest version of A. However, B at $t = 3$ has a dependency on a new module C. So there are two valid ways to satisfy the dependencies: $\mathfrak{C}_1 = \{(A \to 3), (B \to 2), (C \to \bot)\}$, where $\bot$ represents absence of a module in the configuration, and $\mathfrak{C}_2 = \{(A \to 3), (B \to 3), (C \to 3)\}$. However, recall that the system is at $t = 1$. Hence, to obtain $\mathfrak{C}_1$ we need to upgrade two modules (A and B) and to obtain $\mathfrak{C}_2$—three modules (A, B and C). $\mathfrak{C}_1$ is preferred to $\mathfrak{C}_2$.

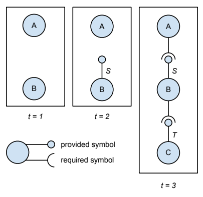


Fig. 1: Choosing module B at $t = 2$ yields the cheapest solution.

In [1], [2] we presented an algorithm finding versions of all other modules such that all dependencies are satisfied, and the number of modules that have to be upgraded is minimal.

III. Is the software easy to upgrade?

To answer this high-level question we construct a heatmap of the system. The color of a cell (*Module*, *Version*) corresponds to the number of modules that—as determined by the algorithm—have to be upgraded when *Module* is being upgraded from *Version* to the most recent version.

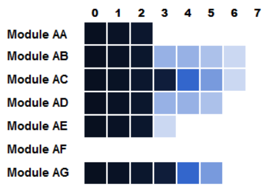


Fig. 2: The more upgrade dependencies, the darker the cell.

Case study. To evaluate the approach proposed we have applied it to software of a large photolithography system, developed by ASML. At the time of the case study, the software contains almost 400 modules, 7000 interfaces and more than 40 million lines of code. To identify dependencies between the modules we have applied CScout (for C) as well as a number of proprietary tools for Python, proprietary data definition files and configuration files. We have successfully extracted information from 327 modules[1]; together with nine monthly versions of the system we've obtained 2616 = 327 * 8 scenarios involving upgrade of each one of the modules from each one of the versions 0–7 to the most recent version.

A partial heatmap of the system is shown in Figure 2. We observe that the colors in a row become lighter from left to right, *i.e.,* the older the version, the more upgrade dependencies are involved. This can be expected because the time span to the latest version is longer, suggesting that more changes could have occurred. Moreover, most modules show dark cells in columns 0–2, and much lighter cells in columns 3–7. This means that most modules are difficult to upgrade to the most recent version 8 if they are of version 2 or older. The heat map does not reveal the cause of this "cliff" from version 2 to version 3: we reconsider this issue in Section IV. Finally, module AF is easy to upgrade: its row is completely blank, indicating that there are no upgrade dependencies. This is typical for modules which see little to no development. It could still be the case that this module has changed, but that these changes were internal to the module.

IV. Why Does Upgrading One Module Require Upgrading Many Additional Modules?

When a module is being upgraded to a more recent version, modules it uses or modules using it might require an upgrade as well. Consider the following scenarios:

Scenario 1 Let module A be upgraded from version i to version j. If version j of A requires a symbol S from B, that was not required by version i, then upgrading A necessitates the upgrade of B if the current version of B does not already provide S. We say that there is an *upgrade dependency from A to B (caused by adding S)*. In Figure 1 there are two upgrade dependencies: from A to B caused by adding S and from B to C caused by adding T.

Scenario 2 Let module B be upgraded from version i to version j. If version j of B no longer provides a symbol S, provided in version i, and module A requires S, then upgrading B necessitates the upgrade of A. We call this an *upgrade dependency from B to A (caused by removing S)*.

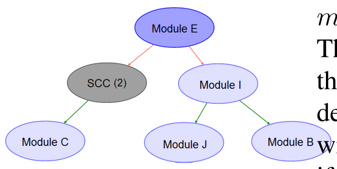


Fig. 3: Upgrade dependency graph of Module E (dark blue) contains an SCC-vertex (dark gray).

The *upgrade dependency graph* represents a single upgrade of a module m, based on the configuration $\mathfrak{C}$ which was determined by the algorithm. The upgrade dependency graph is a directed graph with vertices representing the modules that have to be upgraded, and edges representing upgrade dependencies. Each edge in the upgrade dependency graph is associated with a set of symbols that caused the upgrade dependency. Edges are green if all symbols in the set have been added, red if all symbols in the set have been removed and black if some symbols in the set have been added and while some others have been removed. Cardinality of the set of symbols is represented by thickness of the edge. In order to ease comprehension of the graph, the strongly connected components (SCC) are collapsed to a single vertex, which can be further inspected.

Case study. One of the first observations was the presence of suspicious dependencies, whose existence cannot be immediately explained given the functionality of the modules involved. In many cases these upgrade dependencies involved the test code: while 264 out of 327 modules had more than 150 upgrade dependencies, after the test code has been excluded only 4 out of 327 modules had more than 150

[1]The remaining modules did not contain source code or contain code in languages not supported by the dependency identification tools.

dependencies. Based on this observation we stress the importance of separating the test code from the production code.

Even after the test code has been excluded, we observe that a module upgrade from the version of October 2011 to the version of July 2012 often includes many additional upgrade dependencies. The heat map in Figure 2 tells us that upgrades are much easier from version 3 (January 2012) and onwards. From there, more than half of the modules are upgradeable with only 10 or less additional modules, where the majority of these modules have no additional upgrade dependencies. By inspecting upgrade dependency graphs we further discover that many edges are red, *i.e.,* many upgrade dependencies are being caused by symbols being removed. If removal of symbols is dismissed, the difficulty of upgrading the modules decreases. As opposed to 204 modules (55%) that require upgrading ten modules or less when upgrading from version 3 to version 8 if symbol removal is taken into account, 91% of the upgrades involve ten modules or less if symbol removal is dismissed.

Therefore, to facilitate the upgrades disallowing symbols removal should be considered, or at least a structured deprecation and removal of symbols, *e.g.,* symbols are removed only when they are no longer used in any supported release.

Finally, we have evaluated application of the tool at ASML. ASML developers reported that the tool provided valuable insights in the upgrade structure of the system.

V. Conclusion

We have reported on an industrial approach to complexity assessment of software modules' upgrades. The approach combines a high-level assessment ("is the software structured such that a module can be upgraded with few additional dependencies?") with a lower-level visual feedback to developers ("if an upgrade causes many additional dependencies, what is the cause of this?"). Based on the feedback the developers can consider restructuring the system to simplify future upgrades. The approach has been applied to a software of a large photolithography system, developed by ASML.

References

[1] B. Schoenmakers, N. van den Broek, I. Nagy, B. Vasilescu, and A. Serebrenik, "Assessing the complexity of upgrading software modules," in *WCRE*, 2013, pp. xx–xx.

[2] B. Schoenmakers, "Assessment of software module upgrade complexity," Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, Aug. 2012.

Patching Patches

Reinout Stevens
Software Languages Lab
Vrije Universiteit Brussels

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussels

Viviane Jonckers
Software Languages Lab
Vrije Universiteit Brussels

I. Introduction

Nowadays, it is an industry best practise to store software systems in a Version Control System (VCS). A VCS allows developers to store, share and undo changes of the stored software system. Branching allows developers to implement experimental features (eg. an experimental garbage collection algorithm), or to develop different flavors of the software project (eg. an iPhone and Android version). Code written in one particular branch does not affect code in another branch. When features developed in one branch need to be integrated in the main branch of the software system both branches are merged. These frequently result in merge conflicts, as both branches made changes to the same code. These need to be solved manually. Another possibility is that only particular changes made in a branch need to be replayed in another branch. A common example would be a fix for a bug that occurs throughout multiple branches of the software system. A fix here would be a patch file, for example created by `diff`, that specifies which lines of code need to be changed. Multiple issues arise here as well:

1) A regular merge conflict may occur when both the other branch and the patch changed the same code.
2) Changes made in the patching branch may break the patch in the target branch (eg. renaming a method that is used by the patch).
3) Changes made in the target branch may break the patch in the target branch.

In the rest of this paper we provide an outline of our approach to solve these issues.

II. Proposed Solution

Our proposed solution works in different steps. First, we distill the changes that the patch made to the program. These are the operations that are applied to the AST when applying the patch. The difference between this process and using the output from `diff` is that these operations work on a higher level, and thus are easier to reason over. The used algorithm for this step is discussed by Chawathe et. al. [1], and has been used by other researchers as well (eg. ChangeDistiller[1] [2]).

Second, we track all the changes made in the patching branch up until the branching point, i.e. the version shared between the two branches. We select all the changes that affect changes in the patch under investigation. Whenever an affecting change is detected we apply this change to the patch as well. The end result is a patch that is no longer dependent on changes made in the patching branch.

Finally, we track all the changes made in the target branch up until the target version. Once again we select all the changes that affect the current patch and apply them to that patch. The end result is a patch that can be applied in the target version.

Detecting whether a change affects a change in the current patch is done using the logic program query language *Ekeko*[2] and *QwalKeko*[3] [3]. At the seminar we will present our current progress and results of this research.

References

[1] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change detection in hierarchically structured information,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD96)*, 1996, pp. 493–504.

[2] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” *Transactions on Software Engineering*, vol. 33, no. 11, 2007.

[3] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, “A history querying tool and its application to detect multi-version refactorings,” in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13)*, 2013.

[1]https://bitbucket.org/sealuzh/tools-changedistiller
[2]https://github.com/cderoove/damp.ekeko
[3]https://github.com/ReinoutStevens/damp.qwalkeko

Towards Services for Agile Organizations: The HYDRA Approach

Damian A. Tamburri, Patricia Lago
VU University Amsterdam,
The Netherlands

Luciano Baresi
Politecnico di Milano,
Italy

Sam Guinea
Politecnico di Milano,
Italy

Modern markets are dominated by unprecedented levels of unpredictability and dynamism [1]. In this scenario, organizations must constantly harmonise their organizational and social structures to welcome the new collaborations necessary to maintain their end-to-end business value chains. Such continuously evolving networked organizations increase the need for flexible, or *agile*, organizational structures [2], that successfully span across multiple heterogeneous, distributed partners, while maintaining quality of service [3]. For example ABN-Amro, outsources its IT maintenance division to India. In doing so, ABN-Amro harmonised its organizational structure with new organisational and social awareness services to welcome its maintenance partner, so that maintenance is reachable 24/7 by all ABN-Amro's distributed sites. In addition, ABN-Amro updated its business processes in concordance to the harmonised organisational structure, e.g. to avoid banking information leaving headquarters in The Netherlands. The problem exemplified in the above scenario is twofold. On the one hand, agile organizations [4] need to quickly adapt their distributed business workflows. On the other hand, they need to support the evolution of their organizational structure, e.g., to include or remove new organizations as partners. While the evolution of distributed workflows has been a "hot" topic in services computing research, little is known about how to support the evolution of networked organizations. Yet, this is essential in fostering truly agile organizations. Organizations are based on continuous learning processes, in which constant flows of organizational and social knowledge need to be collected and analysed[1]. To achieve *organizational agility* we need to understand organisations' structures in a more actionable way. The *service* abstraction can provide significant advantages to model networked organizations, and can help to account for dynamicity. The research question is, therefore, "*How can we model and analyse a networked organization using a service-dominant perspective?*". We propose an approach and accompanying tool, to establish and maintain the organizational knowledge necessary to achieve agile organizations. The methodology and tools go under the name of HYDRA, which stands for "Harmonising sYnergies in networkeD oRganizAtions". HYDRA was designed using a service-dominant perspective. The benefits of using this perspective are manifold: (a) the service abstraction creates a continuum between an organisation's service-offer and its organizational structure, e.g. for easier alignment and co-adaptation; (b) service discovery and dynamic-binding offer a playground to tackle non-trivial governance problems in agile organizations, e.g., tackling employee turnover with dynamic skills-retrieval through the Amazon MTurk[2]; (c) services make it easier to reason on the scalability of networked organizations, e.g., by abstracting all assets of organizations as services; (d) finally, services computing research offers a bounty of approaches for context awareness and service adaptation - we need "only" to reduce networked organizations' evolution to a runtime service evolution problem that we can solve before it turns into a costly and untraceable organizational barrier [5]. Figure 1 gives a high-level overview of how HYDRA models a networked organization as an interacting composition of basic social community types [2]. This powerful service-based abstraction leads to a number of possible analyses. For example, we can conduct service conformance analysis by comparing service networks for networked organizations with "ideal" configurations from literature [2], [6]. Also, simulating the model, we can analyse its flexibility, or *elasticity*, to understand the dimensions along which it can evolve.

References

[1] S. Karnouskos, D. Savio, P. Spiess, D. Guinard, V. Trifa, and O. Baecker, "Real world service interaction with enterprise systems in dynamic manufacturing environments," in *Artificial Intelligence Techniques for Networked Manufacturing Enterprises Management*, L. Benyoucef and B. Grabot, Eds. Springer, no. ISBN 978-1-84996-118-9.

[1] http://www.forbes.com/sites/stevedenning/2012/06/06/making-the-entire-organization-agile/
[2] https://www.mturk.com/mturk/welcome

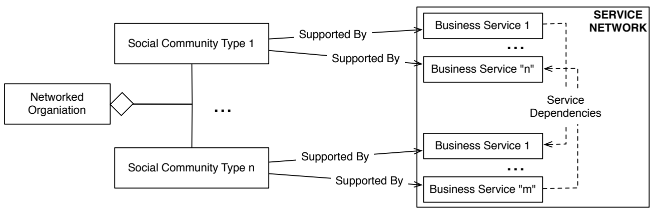


Fig. 1. HYDRA representing networked organizations.

[2] D. A. Tamburri, P. Lago, and H. van Vliet, "Organizational Social Structures for Software Engineering," *ACM Computing Surveys*, pp. 1–34, 2012 – preprint available: http://www.fileden.com/files/2012/3/7/3275239/acm_csur_oss.pdf.

[3] G. Canfora, G. Tretola, and E. Zimeo, "Autonomic workflow and business process modelling for networked enterprises," in *Methodologies and Technologies for Networked Enterprises*, ser. Lecture Notes in Computer Science, G. Anastasi, E. Bellini, E. Nitto, C. Ghezzi, L. Tanca, and E. Zimeo, Eds. Springer Berlin Heidelberg, 2012, vol. 7200, pp. 115–142. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31739-2_7

[4] R. Arell, J. Coldewey, I. Gat, and J. Hesselberg, "Characteristics of agile organizations," in *XP*. Agile Alliance Charter, 2012.

[5] A. M. R. Correia, A. Paulos, and A. Mesquita, "Virtual communities of practice: Investigating motivations and constraints in the processes of knowledge creation and transfer." *Electronic Journal of Knowledge Management*, vol. 8, no. 1, pp. 11–20, jan 2010.

[6] D. A. Tamburri, P. Lago, and H. van Vliet, "Uncovering latent social communities in software development," *IEEE Software*, vol. 30, no. 1, pp. 29 –36, jan.-feb. 2013.

Crowdsourced Knowledge Catalyzes Software Development

Bogdan Vasilescu
Eindhoven University of Technology,
The Netherlands
b.n.vasilescu@tue.nl

Vladimir Filkov
University of California,
Davis, USA
filkov@cs.ucdavis.edu

Alexander Serebrenik
Eindhoven University of Technology,
The Netherlands
a.serebrenik@tue.nl

I. Introduction

Developers create and maintain software by standing on the shoulders of others [1]: they reuse components and libraries, and go foraging on the Web for information that will help them in their tasks [2]. For help with their code, developers often turn to programming question and answer (Q&A) communities, most visible of which is StackOverflow. To engage its participants to contribute more, StackOverflow employs gamification [3]: questions and answers are voted upon by members of the community; the number of votes is reflected in a person's *reputation* and *badges*; in turn, these can be seen as a measure of one's expertise by peers and potential recruiters [4] and are known to motivate users to contribute more [3], [5].

The analogy of StackOverflow as an effective educational institution asserts itself then. The extended effect of education, beyond the immediate edification, is to accelerate or catalyse societal advances. Does StackOverflow have the same effect on software development communities? The connection between developer productivity and their using of StackOverflow is not well-understood. On the one hand, StackOverflow is known to provide good technical solutions [6] and to provide them fast [7], to the extent that closer integration between Q&A websites and modern IDEs is now advocated [8], [9]. On the other hand, as an exponent of social media, using StackOverflow may lead to interruptions impairing the developers' performance [1], especially when gamification is factored in.

In a recent paper [10] we investigated the interplay between asking and answering questions on StackOverflow and committing changes to open-source GitHub repositories. This extended abstract summarises our main findings. GitHub is arguably the largest social coding site, hosting more than three million software projects maintained by over one million registered developers. The two platforms overlap in a knowledge-sharing ecosystem (Figure 1): GitHub developers can ask for help on StackOverflow to solve their own technical challenges; similarly, they can engage in StackOverflow to satisfy a demand for knowledge of others, perhaps less experienced than themselves, or to compete in the "game" to achieve higher reputation. By identifying GitHub users active on StackOverflow and studying their activities on both platforms, we can study if a connection exists between their participation in StackOverflow and their productivity on GitHub. GitHub users are a mix of novice and professional programmers [11]. While it is known that foraging is common for novices and experts alike [2], their *diets* are different [12], with potentially different impact on their performance. Is participation in StackOverflow related to productivity of GitHub developers? Is it more beneficial for some groups of developers than for others? Do the StackOverflow activities impede GitHub commit activities or do they accelerate them?

II. Experimental Setup

We integrated data from two sources: StackOverflow (as part of the Stack Exchange data dump released in August 2012, containing information about 1,295,622 registered users) and GitHub (from GHTorrent [13], a service that gathers event streams and data from GitHub, containing information about 397,348 users and 10,323,714 commits from the July 2011 to April 2012 period).

A key step in our process was merging the GitHub and StackOverflow datasets, i.e., identifying those contributors which were active on both platforms. Merging aliases used by the same person in different software repositories is a well-known problem [14]–[17]. We followed a conservative approach to identity merging and made use of email addresses, present in the GitHub dataset but obscured using an MD5 hash in the StackOverflow one. We decided to merge (i.e., link) a GitHub and a StackOverflow user if the computed MD5 hash of the former's email address was identical to the MD5 email hash of the

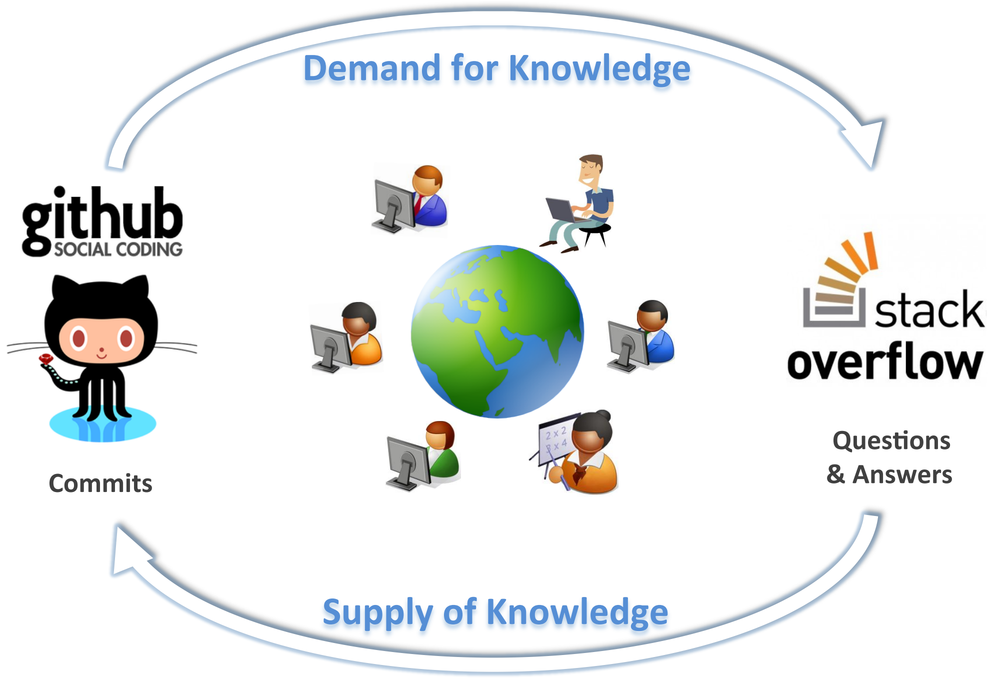


Fig. 1: Demand and supply of knowledge between source code and Q&A.

latter, resulting in approximately one quarter of the GitHub users (23.6%, or 93,771) being linked to StackOverflow. Only 46,967 of these (or 11.8% of the GitHub dataset) asked or answered at least one question on StackOverflow between July 2011 and April 2012.

III. Findings

A. *StackOverflow Experts are Active GitHub Committers*

First, we focussed on differences in StackOverflow involvement of the GitHub developers. We found a *direct relationship between GitHub commit activity and StackOverflow question answering activity*: the more active a committer, the more answers she gives. In other words, highly productive committers tend to take the role of a "teacher" more actively involved in providing answers rather than asking questions. Similarly, the more active an answerer, the more commits she authors. In other words, top users on StackOverflow are "superstars" rather than "slackers": they don't just compete for reputation and badges, but are actually active (open-source) software developers.

In contrast, we found an *inverse relationship between GitHub commit activity and StackOverflow question asking activity*: active GitHub committers ask fewer questions than others; less active question askers produce more commits. Overall, these findings suggest that an activity-based ranking of Stack-Overflow contributors reflects one extracted from their open-source contributions to GitHub, increasing the confidence in the reliability of social signals based on StackOverflow (e.g., answering questions on StackOverflow can be seen as a proxy for one's commit activity on GitHub).

B. *Experts and Novices Have Different Working Rhythms*

Next, we studied whether the working rhythm of the GitHub contributors is related to their StackOverflow activities. We observed that individuals that tend to ask many questions distribute their effort in a less egalitarian way than developers that do not ask questions. No differences were observed between the work distributions for individuals grouped based on the number of answers given. In other words, developers who ask many questions on StackOverflow commit changes to GitHub in bursts of intense activity followed by longer periods of inactivity, i.e., they focus their attention at any given time. Specialization (or focus) of developers has also been noted previously in the context of activity types (e.g., coding versus translating) or files touched as part of a shared project [17], [18]. Therefore, *asking* questions on StackOverflow influences how developers distribute their time over commits on GitHub, while *answering* questions does not seem to have the same effect. We conjecture that this observation is due to developers learning from StackOverflow and committing their experiences to GitHub.

C. Crowdsourced Knowledge Catalyzes Software Development

Finally, we associated GitHub commits and StackOverflow questions and answers over time, in an attempt to understand whether activities in the two platforms show signs of coordination. We found that the rate of asking or answering questions on StackOverflow is related to the rate of commit activities in GitHub. In other words, despite interruptions incurred, for active GitHub developers StackOverflow activities are positively associated with the social coding in GitHub. Similar observations hold for active askers as well as individuals who have been involved in GitHub for sufficiently long time. Finally, StackOverflow activities accelerate GitHub committing also for the most active answerers as well as for developers that do not answer any questions at all.

References

[1] M.-A. D. Storey, C. Treude, A. van Deursen, and L.-T. Cheng, "The impact of social media on software engineering practices and tools," in *FoSER*. ACM, 2010, pp. 359–364.

[2] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *CHI*. ACM, 2009, pp. 1589–1598.

[3] S. Deterding, "Gamification: designing for motivation," *Interactions*, vol. 19, no. 4, pp. 14–17, 2012.

[4] A. Capiluppi, A. Serebrenik, and L. Singer, "Assessing technical candidates on the social web," *IEEE Software*, vol. 30, no. 1, pp. 45–51, 2013.

[5] A. Anderson, D. P. Huttenlocher, J. M. Kleinberg, and J. Leskovec, "Discovering value from community activity on focused question answering sites: a case study of Stack Overflow," in *KDD*. ACM, 2012, pp. 850–858.

[6] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow," Georgia Institute of Technology, Tech. Rep., 2012.

[7] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," in *CHI*. ACM, 2011, pp. 2857–2866.

[8] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing Stack Overflow for the IDE," in *RSSE*. IEEE, 2012, pp. 26–30.

[9] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based search to overcome learning barriers in software development," in *RAISE*, 2012, pp. 47–51.

[10] B. Vasilescu, V. Filkov, and A. Serebrenik, "StackOverflow and GitHub: Associations between software development and crowdsourced knowledge," in *Proceedings of the 2013 ASE/IEEE International Conference on Social Computing*. IEEE, 2013, pp. 188–195.

[11] L. A. Dabbish, H. C. Stuart, J. Tsay, and J. D. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository," in *CSCW*. ACM, 2012, pp. 1277–1286.

[12] B. Evans and S. Card, "Augmented information assimilation: social and algorithmic web aids for the information long tail," in *CHI*. ACM, 2008, pp. 989–998.

[13] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a firehose," in *MSR*. IEEE, 2012, pp. 12–21.

[14] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *MSR*. ACM, 2006, pp. 137–143.

[15] M. Goeminne and T. Mens, "A comparison of identity merge algorithms for software repositories," *Science of Computer Programming*, vol. 78, no. 8, pp. 971–986, 2011.

[16] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "Who's who in Gnome: Using LSA to merge software repository identities," in *ICSM*. IEEE, 2012, pp. 592–595.

[17] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload—a case study of the Gnome ecosystem community," *Empirical Software Engineering*, pp. 1–54, 2013.

[18] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *ICSE*. IEEE, 2013, pp. 452–461.