1 Agile Software Assistance: Summary

As software systems evolve, developers struggle to track and understand the vast amount of software information related to the software source code itself, the application domain, its quality concerns, changes to the underlying infrastructure, and the software ecosystem at large. Mainstream integrated development environments (IDEs) offer only limited support to advise the developer during common development tasks, mainly in the form of so-called “quick fixes” related to purely technical aspects of the programming language. This continuation of our ongoing SNSF project\(^1\) will explore these issues in the following four thematically related PhD tracks:

Speculative software analysis. In this track we tackle the research question: “How can software information be speculatively analysed, and results be automatically presented that are relevant to the developer’s task at hand?” Developers are confronted with large amounts of software data: versions of the software itself, documentation, used libraries and frameworks, contents of the issue tracker, and all related information about the software ecosystem. Although some analysis tools exist, developers are often not aware of what tools or data might be useful to support which tasks, and relevant tools are typically not part of the standard IDE. We envision an automated developer support that proactively offers analysis results tailored to the current development context.

Executable domain models. Here we explore the question: “How can domain models be specified and deployed as executable software artifacts suitable for testing, expressing requirements, and driving design and implementation?” Domain knowledge is at the core of any software development process, and is essential for requirements analysis, object-oriented design, and management of software evolution. However domain models are often manifested only as static documentation that rapidly diverges from reality as the software system evolves. Although model-driven approaches have had some success, their application is largely limited to domains where changes are well-understood so models can be automatically transformed to code. Instead of transforming models to code, we imagine an approach in which executable domain models are developed throughout the software life cycle, and form an integral part of the system under development.

Domain-specific software quality. In this track we address the question: “How can domain-specific quality concerns and their corresponding corrective actions be effectively specified and monitored?” As a software system evolves, there may be important quality aspects of which the developer may only have passing knowledge, such as the security impact of certain implementation choices. We envision a system that actively monitors such domain-specific quality concerns and advises the developer of possible corrective actions. We plan to focus mainly on security issues in Android software, an area where we have achieved some very promising initial results.

API client migration. As software systems evolve, client software that depends on them must be adapted to the evolving Application Programming Interfaces (APIs). Here we plan to study the question: “What is a suitable model for specifying, reasoning about, and automating API client migration?” Although strides have been made in automating certain kinds of adaptations, generally API migration is poorly supported in practice. We imagine a system in which migrations can either be automated, or supported by tools that systematically guide developers in the migration. We will analyze various case studies of past API migrations to better understand the potential for automated migration, and carry out experiments to assess such migration schemes.

\(^1\)http://scg.unibe.ch/asa2
2 Research plan

Main applicant Oscar Nierstrasz
Project title Agile Software Assistance (ASA3)
Keywords software evolution, software analysis, domain models, recommender systems

The goal of this project is to explore ways to provide timely (“agile”) assistance to software developers that is relevant to their current development context. The project is designed as four thematically-related PhD topics that will enable and encourage collaboration between the research staff members without imposing critical dependencies.

2.1 Current state of research in the field

The field of Software Engineering has undergone a major shift over the past twenty years, with an ever-growing focus on empirical studies, program comprehension, and better support for developers to solve practical problems. Since there is a vast body of literature in these areas, we focus here only on some of the key works leading to the specific research questions that motivate this proposal.

Speculative software analysis. An Integrated Development Environment (IDE) in principle offers a single entry point for all the tools a developer needs to support development activities. Historically, IDEs tend to focus on low-level technical tasks, such as editing source code and building the object code or debugging it, and do not support well many of the high-level tasks that arise in development, such as assessing software quality or estimating the impact of planned changes. Although modern IDEs such as Eclipse, NetBeans, and IntelliJ\(^2\) all support a plugin architecture that allows new tools to be integrated as add-ons, assistive support for specific high-level development tasks is generally missing. Even if a relevant plugin exists, a developer may not be aware of this. For this reason software developers often turn to mailing lists or dedicated Question & Answer forums such as Stack Overflow\(^3\) to answer their questions.

The need for a dedicated environment to model and query software artifacts was recognized quite early. Muller’s Rigi [MK88] is an early example of a system to model and manage software-related information. More recently, the Rascal MPL (Metaprogramming Language) [KvdSV09] from the team of Vinju and Klint exploits the Eclipse plugin-architecture to offer advanced support for querying, manipulating and transforming source code. The BOA language and infrastructure [DNRN13] on the other hand offers support for mining information from software repositories.

Considerable research has been carried out in recent years into program comprehension, software analysis, mining software repositories, and recommender systems for developers. For example, Lanza’s team in Lugano has a strong track record of research in these areas [LM06, MML15, PBDP+14, PSB+17]. A recent study by the team of Mezini suggests that evaluations of code recommender systems do not take the evolving context of the developer into account [PANM16].

The team of Gall in Zurich has a strong track record in similar topics, and particularly in carrying out empirical studies [PPB+16]. Various researchers have studied data that can be mined from mobile app stores, notably Harman [HASJ+16] and Zeller [AKG+15]. At Microsoft, Bird and Zimmermann have pushed forward the use of data analysis techniques for analyzing software

\(^3\)https://stackoverflow.com
Hassan (Canada) has studied how data analysis techniques must be adapted to deal with the specific characteristics of software data [GMH17, TMHM16].

Increasingly research is aimed at better understanding the developer’s needs, such as what kinds of analyses are of value to developers [SMDV06, BZ14, SMK17], how much time developers spend in various activities [MML15, PNAM17], how developers understand what is meant by productivity [MFMZ14], and what developers needs are for program analysis [BZ12, CB16]. Data science is playing an increasingly important role in software development teams [KZDB16].

Brun and colleagues have explored how possible development actions can be speculatively analyzed by assessing the impact of these changes on future states of the software [BHEN10, MBH+12]. Google has developed the Tricorder program analysis platform in an attempt to better integrate program analysis tools within the developer workflow [SvGJ+15].

Given that there is a clear need for assistive support for high-level developer tasks while the only available tools require considerable expertise and training to exploit, we pose the following research question: “How can software information be speculatively analyzed, and results be automatically presented that are relevant to the developer’s task at hand?”

The challenges include: (i) how to recognize from the developer’s working context questions of interest; (ii) how to turn common developer questions into queries over software information; (iii) how to identify interesting (for the developer) trends and outliers; and (iv) how best to present (or visualize) results in a way that supports developers without disrupting their work flow.

**Executable domain models.** Since the earliest days of Software Engineering the importance of modeling and simulation of domain concepts has been recognized [Ran68]. Many approaches have been explored since then to bring code closer to domain models.

The first object-oriented language, Simula, was conceived as a language for simulating the real world, but it was quickly realized that the paradigm of programming as simulation had broader applications [Dah04]. Alan Kay seized on “computation as simulation” as the design principle behind Smalltalk, the first purely object-oriented language [Kay77]. As OO languages gathered momentum in the 1980s, practitioners advocated that object-oriented methods could bridge the gap from domain modeling and requirements collection, through analysis, to design and implementation, by using OO modeling concepts throughout the development process [SM88].

Model-driven Engineering (MDE) [Sch06] promoted the vision of application development being driven by transformation of models at higher, platform-independent levels down to actual platform-dependent implementations. Classically in MDE, models are static, and only serve to drive the transformation to code — they are not executable themselves, and do not form part of the final running software system. Although MDE has become widespread in industry, its main use in practice is to document software architecture [WHR14]. A recent study suggests however that significant benefits can be achieved from MDE when the models themselves are executable, thus assisting developers in specification, simulation, testing and analysis [RTL+17].

Another study of MDE practices revealed that domain specific languages (DSLs) are widely used in industry [HWR14]. DSLs have a long history, having been used both for technical domains (such as job control languages, query languages, and configuration languages) as well as for business domains (telecommunications, multimedia, hardware design) [DKV00, Fow10]. Dedicated language workbenches have been developed to support the design and implementation of
DSLs, such as mbeddr [VKS+17], a state-of-the-art workbench that avoids parsing the concrete syntax of DSLs by using “projectional editing” to directly edit the underlying syntactic structures. While DSLs clearly help to raise the abstraction level of specifications in technical or business domains, their utility and impact on software maintenance must be carefully assessed [ACG+15]. A particular concern is that the proliferation of languages in modern, heterogeneous software systems can negatively impact understanding and communication (due to the need to learn and understand multiple languages) [MKL17]. In contrast, Evans’ notion of a Ubiquitous Language in Domain Driven Design [Eva04] is that of a common language that rigorously defines the domain vocabulary used by both developers and users, thus enabling communication and understanding.

In contrast to both MDE and DSLs, Naked Objects unify domain objects and software entities [Paw04]. Business logic is encapsulated in the domain objects and the user interface is generated from these domain objects. The domain model and the executing runtime are thus tightly coupled. So far, however, the naked objects approach has found little traction in industry.

Model checkers are tools used to verify certain properties of software systems (such as safety and liveness properties in concurrent systems). A large number of such tools has been developed over the years [BBF+01]. Many operate on dedicated specifications of models, while others reason with models extracted from the software source code. To our knowledge, however, none operates on a model that is embedded in the software system itself.

Despite the long history of diverse approaches to close the gap between domain models and code, managing this gap remains a significant challenge. We therefore pose the research question: “How can domain models be specified and deployed as executable software artifacts suitable for testing, expressing requirements, and driving design and implementation?”

Particular challenges include: (i) identifying suitable languages or tools to express executable domain models, (ii) incrementally eliciting and updating domain models from stakeholders, (iii) leveraging the executable domain models for various development tasks (such as testing).

Domain-specific software quality. Automated tools to assess software quality have a long history, starting with “lint”, a tool to help developers find common errors in C code [Joh78]. The best-known modern equivalent is likely FindBugs, an analogous tool for Java developers [AHM+08].

Since the first attempts to encode software design best practices as “design patterns” [GHJV95], interest has also grown in automatically identifying so-called “anti-patterns” [BMMM98] and “code smells” [FB99]. Code smell detectors classically apply rules representing a bad practice to a representation of the source code (i.e., source text or an abstract syntax tree) and then present a report (possibly a visualization) of the violating source code [vM02]. For example, detection strategies for common code smells can be defined in terms of object-oriented software metrics, and the results presented using lightweight visualizations [LM06]. Continuous assessment of code quality integrated into the IDE can be highly effective, as demonstrated by InCode, an Eclipse plug-in to perform various software analyses [GVM17]. Many developers, however, are not aware of the notion of code smells, and better tool support is needed [YM13].

Code smells are also referred to as “technical debt,” indicating that short-term gains obtained by bad software practice may lead to high long-term costs, a notion that can be traced to Lehman’s “Laws of Software Evolution” [Leh80]. Currently a wide range of tools exists to assess technical debt, and even those focusing on particular issues demonstrate significantly different results for
different projects [ZVI+14]. The state-of-the-art is therefore still quite immature.

Quality assessment tools have been developed for a variety of bad practices, ranging from low-level code smells to high-level architectural smells [ABT+16, GPEM09]. While much effort has been invested in general-purpose quality tools, less work has been done on domain-specific quality concerns arising either in a specific business application domain, such as web shopping sites or insurance applications, or a technical domain, such as web sites or mobile apps.

One current domain of interest is that of security smells, especially for mobile devices, as app developers are often not trained software professionals, and may not be aware of the security impact of their design choices. Even professionals are not always aware of the issues, as delivered functionality is often given priority over non-functional quality concerns. A study of 1,100 Android apps revealed pervasive violations of security concerns [EOMC11]. Various tools have been developed to assess security concerns in Android apps, such as aDoctor, which applies lightweight heuristics to detect a number of Android-specific code smells [PNP+17], and Mudflow, which detects Android malware by analyzing the flow of sensitive data to abnormal sinks [AKG+15]. Although a large number of tools exist, few are mature enough to be used in practice [RBGI+16].

We conclude that, although many software quality assessment tools exist for a variety of quality concerns, only the most basic tools are widely adopted. Many quality concerns are not yet well-served. We thus pose the following research question: “How can domain-specific quality concerns and their corresponding corrective actions be effectively specified and monitored?”

Some of the challenges include: (i) mining and expressing quality concerns as specific code smells, (ii) avoiding false positives, (iii) interpreting code smells as actionable advice for developers.

**API client migration.** Software evolution has been studied at least since the late 1970s [Leh80]. The topic of automated support for migrating from one version of a software system to another originated a decade later in the context of automated schema evolution for object-oriented databases [BKKK87]. Shortly afterwards, Casais, Griswold and Opdyke independently defended the first PhD theses on the topic of “refactoring” object-oriented software [Cas91, Gri92, Opd92].

The first automated refactoring tool was developed for the Smalltalk IDE [RBJ97]. On the one hand, refactoring entered the mainstream [FBB+99], and automated support was slowly adopted in other IDEs, such as Eclipse; and on the other hand, program transformation started to become mainstream, and dedicated platforms were developed, such as Stratego/XT [Vis04].

As this technology became more mature, the subject of automated API migration became a target. Dig and Johnson showed that about 90% of API-breaking manual changes are refactorings, suggesting that such changes should be automated [DJ06]. Classically, when an API changes software developers are informed, with the help of automated tools, that the old API is “deprecated” and should no longer be used. This information is sometimes accompanied by advice on how to migrate the code, and less often an automated patch is provided. A study of API deprecation in a Smalltalk ecosystem showed that while many API deprecations can have a large impact, the supplied guidelines are often substandard; nevertheless there are ample opportunities for automation [RLR12]. A recent study showed that just 61% of deprecated APIs offer an alternative API fix [KMP+14]. Xavier et al. find that mature projects do not stabilize their APIs over time, in fact they introduce more breaking changes than newer projects do [XBV17], rendering the problem of API migration an issue during the whole lifespan of a project.
Robillard et al. have carried out an extensive survey of API property inference techniques, including techniques for mining API migration mappings from changes to software versions [RBK+13]. Although most migrations consist of simple refactorings, such as renaming an interface, or moving it to a different module, several techniques focus on more complex mappings involving multiple APIs. Tansey & Tilevich infer refactorings to migrate from legacy towards annotation-based frameworks, such as from the JUnit 3 testing framework to JUnit 4 [TT08]. Li and Thompson report initial success with a tool to generate API migration refactorings for the Erlang programming language [LT12]. A radically different approach is to deploy API adaptations at run time [PGS+11].

Various approaches have been developed to measure the impact of API changes. Chianti estimates the impact of API changes on test suites [RST+04]. Raemaekers et al. [RvDV12] introduce metrics to assess the historical stability of an API. SemiDiff [DR11] recommends adaptive changes to client programs by analyzing how a framework has been adapted to its own API evolution.

Bavota et al. studied the evolution of Apache, a large software ecosystem, and found that, over time developers are increasingly reluctant to adapt client programs, presumably due to the high cost of adaptation [BCP+13]. A study of the Pharo Smalltalk ecosystem [HRA+15, Hor14] confirmed that many developers do not react to API deprecations due to the high cost of adaptation, even though many adaptations could in principle be supported by automated tools.

Numerous researchers have also studied the co-evolution of different parts of complex software systems, such as design and implementation [DDVMW00], production code and tests [ZVRDvD08], and even source code and the build systems [Ada09].

While most of the existing work on API migration has focused on mining automated migrations from existing code, we seek a more integrated approach to define client adaptations as part of the API evolution process. We therefore propose the following research question: “What is a suitable model for specifying, reasoning about, and automating API client migration?”

Challenges include: (i) developing a suitable model for modeling API migrations beyond simple textual substitutions, (ii) capturing API client migrations at the same time that new APIs are defined, (iii) ensuring (some degree of) behavior preservation of migrated application.

2.2 Current state of own research

The PI, Prof. Oscar Nierstrasz, founded the Software Composition Group (SCG) at the University of Bern in 1994. The group has produced fundamental research results in several areas, notably component-based software engineering, object-based concurrency, object-oriented reverse and re-engineering, software evolution, and software modeling and analysis. In 2013 Nierstrasz was awarded the prestigious Dahl-Nygaard Prize for contributions to Object-Orientation. The team has produced over 250 peer-reviewed journal and conference publications, and 35 PhDs, many of whom were supported by an unbroken series of SNSF-funded research projects.

Dr. Mohammad Ghaafari (U. Bern) is a postdoctoral researcher in the Software Composition Group since January 2016. He has strong expertise in software testing, software analysis, and particularly security for mobile apps. Since joining the SCG, he has contributed to four journal

\[\text{http://www.aito.org/Dahl-Nygaard/}\]
\[\text{http://scg.unibe.ch/publications/scg-pub}\]
\[\text{http://scg.unibe.ch/publications/scg-phd}\]
\[\text{http://p3.snf.ch/person-17147-Nierstrasz-Oscar}\]
papers, 15 international peer-reviewed conference papers, and numerous workshop papers. He has been actively co-supervising PhD, Masters and Bachelors students together with the PI.

This proposal is a continuation of the SNSF project, “Agile Software Analysis (ASA2),” (ASA2), which is itself a continuation of the project “Agile Software Assessment” (ASA1). ASA2 has thus far produced 4 peer-reviewed journal papers, 20 peer-reviewed conference papers, and 6 PhD theses. We will report here mainly on the progress in ASA2 relevant to the current proposal, and only make reference to particularly relevant older work.

Speculative software analysis. In this section we summarize our recent activity related to analysis of software ecosystems and software visualization towards supporting developer activities.

One of the key artifacts produced by the SCG has been Moose, a platform for software and data analysis [NDG05]. Moose has served as a platform for much of the research into software modeling and analysis at SCG [NL12], and is now managed by an independent consortium. We continue to use Moose extensively for our own research.

Most of the tracks in ASA2 are directly relevant to this new track. Jan Kurš studied the problem of agile modeling, namely how to rapidly construct models of complex software systems using approximate semi-parsing technology. He developed bounded seas, [KLIN15] a novel approach to island parsing in which syntactic “islands” of interest in source code can be more easily extracted than with standard parsing technology, and he developed new techniques to efficiently compose parsers of sub-languages of interest [KVG+17]. The software models constructed using this semi-parsing technology are then available for dedicated analyses. He completed his PhD in 2016 [Kur16], and is now working at Google Zurich.

Boris Spasojević explored various means to extract useful information from the software ecosystem (i.e., the set of projects, libraries and repositories somehow related to a given software system) of a project and to exploit this information to give useful feedback to the developer [SGN16]. He completed his dissertation on Developing Ecosystem-aware Tools [Spa16], and is now working on virtual machine technology at Oracle Labs Switzerland. He continues to collaborate with the SCG, supervising student projects related to Oracle’s research.

Nevena Milojković studied techniques for lightweight inference of types in dynamic languages, including exploitation of type hints in method argument names [MGN17b] and mining inline cache data from the runtime system [MBGN17]. She completed her PhD in this topic [Mil17], and is currently a postdoctoral researcher at SCG carrying out an in-depth study of technologies related to Speculative software analysis.

Haidar Osman carried out numerous ecosystem analyses during the course of his studies. In particular he studied the evolution of exceptions in long-lived Java systems [OCC+17, OCS+17], and he tracked the use of null checks in open-source Java systems, uncovering that fully a third of
all conditionals in such systems are dedicated to null checks [OLLN16]. Since null exceptions are amongst the most common bugs to arise in Java (and other) software systems, this work led to a deeper analysis of bug prediction methods, revealing that, somewhat contrary to expectations, the data analysis methods used for bug prediction must be carefully adapted to each individual project to yield useful results [OGN17b, OGN17a, OGNL17]. Osman completed his PhD in this topic [Osm17], and is now working as a data scientist at Swisscom. He continues to collaborate with the SCG, supervising student projects related to software engineering data science.

Finally, Leonel Merino\(^{16}\) has been carrying out an extensive study of known software visualizations and how effective they are at answering questions developers have about software [MGN16a, MGN17a]. MetaVis is a tool to explore the space of available visualizations [MGN+16b]. Currently Merino has been evaluating the potential for 3D and VR interaction to explore software models [MFB+17, MGAN17]. He is expected to defend his thesis in the Spring of 2018.

**Executable domain models.** We have previously argued that software systems should be model-centric [NDR09] to enable graceful software evolution, that is, models of both the application domain and software itself should be available at run time. Earlier examples of this principle seen in our research include Helvetia [RGN10], an infrastructure to enable the integration of domain-specific languages into the toolchain of the development environment, and object-centric debugging [RBN12], an approach to enable more domain-specific debugging interactions by putting domain objects (rather than run-time stacks) into focus.

Within ASA2, Claudio Corrodi\(^{17}\) has explored ways to extend object-centric debugging with declarative breakpoints that can interrupt execution with more domain-specific predicates over the program state than are possible with conventional debuggers [Cor16].

A particularly interesting direction has been that of moldable tools in software development [CGK+17], studied extensively by Andrei Chiș\(^{18}\) in his PhD work. While most IDE tools offer only generic functionality, moldable tools can be easily adapted to a particular application domain. A conventional debugger, for example, offers only standard features to step into, over, or out of a particular run-time stack frame in order to explore the running system. A moldable debugger [CDGN15], on the other hand, is aware of relevant domain concepts, and can exploit them to offer more useful interactions. For example, a moldable debugger can be adapted to an event-based system to step through to the next observer triggered by a particular event, or it can be adapted to the parsing domain to be aware of parsing rules to step through to the next rule of interest. Such interactions are next to impossible with conventional debuggers.

Other moldable tools developed according to these principles include object inspectors, search tools, and editors (ongoing work) [CGK+16, Chi16b]. The moldable tools infrastructure has been integrated into Pharo,\(^{19}\) a popular open-source Smalltalk environment used widely both in research and industry, as well as in Moose. Andrei Chiș completed his dissertation in 2016 [Chi16a] and is continuing research and development at feenk GmbH on novel development environments. He continues to collaborate with the SCG by supervising student projects related specifically to moldable tools, and more generally to Smalltalk technology.

---

\(^{16}\)Funded by the University of Bern and a Chilean Scholarship, but fully integrated into ASA2.

\(^{17}\)Funded by ASA2.

\(^{18}\)Funded in ASA1, but completed his dissertation during ASA2.

\(^{19}\)http://www.pharo.org
Domain-specific software quality. The SCG has been active for many years in the domain of object-oriented reengineering, with particular attention being paid to strategies for detecting software quality issues in legacy software systems [DDN02].

Within ASA1 we have studied architectural quality issues arising in practice in industry [CLN14], and we have developed a high-level DSL for specifying architectural constraints and monitoring their violations during development [CLN15].

In ASA2, Yuriy Tymchuk has studied how quality rules that express programming best practices can be productively integrated into the developer’s workflow. Although tools to check such quality rules have been available in Pharo for many years, developers largely ignored them, as they imposed additional overhead to run them. QualityAssistant, in contrast, integrated quality feedback directly within the development tools, and provides only feedback related directly to the current developer task [TGN16, TGN18]. A unified model for quality rules called Renraku [TGN17] facilitates the integration of new kinds of quality rules. A dedicated interactive 3D visualization exposes how quality violations in a complex system evolve over time as the quality rules also evolve [TMGN16]. Yuriy Tymchuk completed his PhD in 2017 [Tym17] and is now working as a data scientist at Swisscom.

More recently we have begun to explore more domain-specific quality rules, in particular security smells in open-source Android software. We have reviewed commonly occurring violations of best practices and developed lightweight analysis tools to detect them [GGN17]. Pascal Gadient recently completed his MSc thesis [Gad17] on this topic, and is now a PhD student funded by ASA2. He plans to continue research in this area in ASA3.

API client migration. Our early work on reengineering patterns [DDN02] explored strategies for migrating legacy systems to cleaner designs, but did not specifically target the problem of API migration. Later we studied the use of dynamic analysis to support automated migration of testing code from the JUnit testing framework API to JExample, an extension supporting cascaded tests [HKN08]. We have studied how to detect hidden dependencies between different entities within a system in the context of coevolution [GDK+07] and runtime dependencies [LGN07]. This can serve as the base model for API migration. More recently we studied the use of first-class contexts to support dynamic updates of running systems [WLN13, TWDN15]. We also carried out a study of the need of API developers and users to understand the impact of changes [HLSN14].

In ASA1 we developed semi-automated migration support to eliminate dependency cycles in the package structure of a software system [CALN16].

In ASA2, Manuel Leuenberger has studied how to infer the likelihood that methods may return null values (a common source of errors) by analyzing their client API usage [LOGN17a] using a dedicated infrastructure to automatically collect and harvest such information [LOGN17b]. This work was started as part of Leuenberger’s MSc thesis [Leu17], and he is continuing to work on this topic as a PhD student.

20 Transferred from USI to SCG to complete his PhD while funded by ASA2.
22 Funded by ASA2 since Feb. 1, 2017.
2.3 Detailed Research Plan

This project, like its predecessor, is designed as four thematically related, but independent PhD topics. Although the topics are related both in terms of goals (e.g., advising developers) and techniques (e.g., mining and analyzing software information), thus offering ample opportunity for collaboration, we avoid any critical dependencies between tasks. All tracks have either started already, or are starting now, so funding is requested for the remaining three years (plus two months) of research, starting January 1, 2019.

The goal of this project is to explore ways to provide timely (“agile”) assistance to software developers that is relevant to their current development context. The four tracks aim to assist software developers with advice that is targeted at different kinds of development tasks. (1) Speculative software analysis proactively analyzes diverse sources of software information (software repositories, version histories, issue tracking systems, etc.) to identify trends and outliers that are likely to be relevant to the developer’s current working context; (2) Executable domain models capture domain knowledge and requirements as part of the software system under development, and can be exploited to guide and assist the developer in tracking the correspondence between software artifacts under development and the underlying domain concepts; (3) Domain-specific software quality advice is offered by capturing and monitoring quality constraints that are highly specific to a particular technical or application domain, such as software security concerns for mobile apps; (4) API client migration advice leverages knowledge about the evolution of frameworks and libraries used in application development to guide migration activities, and then partially or fully automate such tasks wherever possible. The four PhD students will be supervised by Prof. Oscar Nierstrasz, and the third and fourth PhD students will be co-supervised by Dr. Mohammad Ghafari.

As all tracks are concerned with supporting software development activities, there will be ample opportunity for interaction and collaboration between the four doctoral students. For example, common motivating examples and case studies, such as security for mobile apps, can be used across several tracks. Similarly, underlying techniques related to software modeling, data analysis, and software visualization can be leveraged across several tracks. On the other hand, there are no strict dependencies, so no track risks failure due to difficulties encountered in another track.

In addition, as in our previous SNSF projects, we will also recruit Masters and Bachelors students to contribute to specific tasks suitable for thesis work.

Partners. The following researchers will play key roles in one or more of the research tracks. Dr. Stéphane Ducasse (Research Director INRIA Lille, RMoD team) is an expert in software
modeling and analysis. He also leads the development of the Pharo\textsuperscript{23} environment, a platform heavily used in our research. We will seek his collaboration in all tracks, but especially Executable domain models and API client migration. Dr. Mohammad Ghafari (U. Bern) will contribute mainly in co-supervising the tracks on Domain-specific software quality and API client migration. Dr. Tudor Gîrba (founder, feenk GmbH) is an expert in software modeling and assessment. He curates the Moose\textsuperscript{24} platform for software and data analysis, and he leads the development of the Glamorous Toolkit,\textsuperscript{25} a novel IDE for Pharo. He will collaborate mainly in the tracks on Executable domain models and Speculative software analysis.

Collaborations. We also plan to benefit from long-standing collaborations with the following researchers with expertise in software analysis. Prof. Alexandre Bergel (U. Chile, Pleiad research laboratory) is an expert in software visualization and his collaboration will be especially welcome in the track on Speculative software analysis. Prof. Michele Lanza (USI, REVEAL research group) is an expert in software analysis and visualization. We will seek his collaboration particularly in the track on Speculative software analysis. Prof. Radu Marinescu (PU Timisoara, LOOSE Research Group) is an expert in quality assurance, software metrics, software evolution and software maintenance. We will collaborate with him in the tracks on Speculative software analysis and Domain-specific software quality. Dr. Sebastiano Panichella (UZH, member SEAL research group) is an expert in empirical software engineering and recommender systems for software development. His expertise is relevant to all tracks, and in particular Speculative software analysis and API client migration.

2.3.1 Speculative software analysis

Many questions arise during software development: Who knows this code? How should I use this API? What is the impact of this change? Many experimental tools and recommender systems have been developed to answer some of these questions, but most of them are not standard components of mainstream interactive development environments (IDEs). Although many developer questions can also be answered by analyzing available software information, most developers are not data scientists and certainly do not have time to devote to such analysis.

In this track we seek to determine from the developer’s working context (i.e., the development task or the code under development) what questions may be useful to ask, and to speculatively analyze the available information to proactively propose actionable analysis results of interest. We pose the corresponding research question as follows:

\begin{tabular}{|p{\textwidth}|}
\hline
\textbf{RQ 1.} “How can software information be speculatively analysed, and results be automatically presented that are relevant to the developer’s task at hand?” \\
\hline
\end{tabular}

The PhD student working on this track, Pooja Rani (start of PhD studies Jan. 1, 2018), has started an in-depth literature review to establish which kinds of developer questions are associated with which development activities, and a survey of existing tools and analyses available to answer such questions. The goal will be (i) to determine what opportunities exist to provide actionable

\textsuperscript{23}\url{http://pharo.org}
\textsuperscript{24}\url{http://www.moosetechnology.org}
\textsuperscript{25}\url{http://gtoolkit.org}
developer advice through speculative software analysis, and (ii) to identify suitable application
domains for case studies.

In the subsequent 36 months of this PhD project, for which funding is requested, we plan the
following activities:

— **Study of developer questions.** We will carry out empirical studies with both experienced
and novice software developers to determine what specific categories of actionable developer
questions can be identified from the development context.

— **Speculative analysis of software data.** We will study ways to transform developer
questions into queries over software information mined from various sources, and how to
identify relevant trends and outliers.

— **Integration of analysis results into software process.** We will explore ways to present
the results of speculative analysis in the IDE with the help of suitable software visualizations.

Although these three subtracks map roughly to the subsequent three years (plus two months) of
the project, in practice the activities will overlap. We envision a considerable degree of iteration, so
that evaluation of speculative analysis for certain domains can be carried out early in the project.

**Study of developer questions.** The main goal of this subtrack is to gain insight into the kinds
of developer activities and questions that would gain most benefit from speculative analysis. We
plan to first carry out surveys and semi-structured interviews with both experienced professional
developers and novices (students). This will be followed up with in-depth studies of developers
carrying out specific tasks identified as being of interest in the first phase. One technique we wish
to explore is to integrate monitoring and feedback directly within the IDE to allow us to track
the developer’s individual actions and questions for later analysis. A key challenge with this kind
of study is to obtain non-trivial participation from developers. In this track we therefore plan
to leverage our access to the Pharo community and platform, where our experiments can benefit
both from participation by experienced developers and from the reduced cost of instrumentation.

A second important goal is to gain insight into how development contexts correlate to developer
questions, and how these contexts can be automatically identified. Examples of contexts that may
be of interest range from the current software artifacts being browsed or modified (i.e., open tabs
on individual code fragments), or the specific tools being used (i.e., for debugging or testing).

As a result, we expect to identify a ranked list of developer questions and associated develop-
ment contexts ranging from easy (easy to identify and support, with good potential for actionable
advice) to challenging (hard to support with less clear potential for actionable advice).

**Speculative analysis of software data.** Previous work on analyzing software data has largely
focused on tackling specific questions (e.g., which parts of the code are most prone to bugs?). This
subtrack instead seeks to anticipate what questions the developer might ask by aggressively exploring
the available software data, such as “How have others used this API?” “What possible errors
can arise here?” “What is the best way to track down this bug?” To this end, we will experiment
with techniques to speculatively analyze software data, for example to establish semantic links
between heterogeneous data (e.g., to uncover recurring themes appearing in both issue trackers
and software repositories), to expose trends (e.g., correlation between change requests and actual fixes), and to identify outliers (e.g., extremely unstable parts of the code). Here too we plan to leverage our access to the Pharo community both in exploiting data that is available in the Pharo ecosystem, and in pro-actively proposing tools extensions for the current development context (e.g., the currently inspected object, or the current debugging session).

One of the key challenges in this subtrack will be to produce actionable advice, for example to propose specific code changes to make, or code patterns to implement. For this reason it is important that the empirical studies identify the kinds of actionable advice that hold the most promise for our experiments.

**Integration of analysis results into software process.** Most software analysis tools require explicit action by a developer to obtain any results. This presupposes that the developer is aware of the tool, is able to install it, knows how to run it, can determine if the results are useful, and can interpret the results as some action to perform. In contrast, we seek to anticipate the developer’s questions and needs by presenting the results of speculative analysis in a way that is conspicuous but not obtrusive. (We want to avoid the infamous Microsoft Office “clippy” experience.)

To this end we envision the integration of lightweight presentations and visualizations of analysis results in a “dashboard monitor,” following our previous experience with QualityAssistant. The developer should be able to give feedback, indicating whether the results are useful or not, and can explore the results in the case that they are relevant. This feedback will not only be of use to tool maintainers for improving the quality of the results, but we envision the possibility of the tool to automatically “learn” to better assess which analysis results are more likely to be of interest in which contexts.

We plan to collaborate with Radu Marinescu, Michele Lanza and Alexandre Bergel in the areas of software data analysis, and in the presentation and visualization of analysis results.

### 2.3.2 Executable domain models

Although capturing and specifying domain knowledge is considered to be an essential activity in software development, domain models typically exist only as static artifacts disconnected from the software source code. Unlike model-driven engineering, which seeks to generate code from models, we propose to express domain models as executable artifacts from the start, and to directly integrate them as part of the software under construction, to enable requirements specification, testing, live documentation, and co-evolution of models and code. In other words, we ask:

**RQ 2.** “How can domain models be specified and deployed as executable software artifacts suitable for testing, expressing requirements, and driving design and implementation?”

This research track will be tackled mainly by Nitish Patkar (start of PhD March 1, 2018), a recent MSc graduate who has explored the use of a “vision backlog” tool to track and manage the requirements elicitation process in his MSc thesis (U. Paderborn, 2018). In the first 10 months of his research, Patkar will (i) review the literature on approaches that attempt to integrate domain modeling with the software under development, (ii) experiment with state of the art tools (e.g., MDE tools, Naked Objects), and (iii) identify potential case studies. In this last activity we will collaborate closely with Tudor Gîrba of feenk GmbH, who has already gained some positive
experience developing executable domain models in the restaurant automation domain, and who is interested in developing a more general platform and methodology for commercial application.

We plan the following three subtracks, roughly corresponding to the 38 months of the project:

— **Example-driven domain modeling.** We will explore ways to specify the domain model for a software application as a collection of executable examples.

— **Embedding domain models.** We will explore and assess ways to integrate an executable domain model into the core of a running software system, and exploit it to support various development activities.

— **Evolution of domain models.** We will study how embedded domain models can be exploited to enable co-evolution of models and code as requirements change.

As in the previous track, the three subtracks are not intended to proceed strictly linearly, but will progress iteratively and incrementally.

**Example-driven domain modeling.** A standard practice in software engineering is to elicit requirements and domain knowledge by the collection of so-called “use cases” including examples of common usage scenarios. Typically such examples are specified as static documents (i.e., natural language or UML diagrams). Instead we propose to directly specify them as running code examples. In first experiments we propose to directly code them in a high-level language (Smalltalk). Both our own earlier research experience in composing tests from examples, as well as feenk’s current experiments specifying domain knowledge in Smalltalk suggest that this is feasible, though a general methodology is lacking.

We plan to experiment with case studies at various scales, starting with simple cases from our own environment, and working up to industrial examples. The goal will be to identify some common principles and mechanisms that will speed up the process of specifying examples, and generalize them to executable domain models. A second goal is to explore visual paradigms for expressing models and examples, such as state machines and transition diagrams, to enable the rapid development of executable models. We also plan to experiment with existing tools for building and specifying executable models (such as model checkers, many of which offer the ability to simulate possible execution paths).

Challenges we will address are (i) how best to express scenarios in code, (ii) how to organize the specified examples, (iii) how to elaborate the examples to a full-fledged domain model.

**Embedding domain models.** The next challenge is how to embed an executable domain model within a software system so that it can be leveraged to support common software development tasks, such as requirements elicitation, software design, testing, documentation, and communication between multiple stakeholders. We plan to rely on a so-called “onion architecture” in which the executable domain model exists in the innermost layer of the software system, and is technically independent of the outer layers, in particular of such aspects as persistence and user interfaces. A dedicated infrastructure will maintain the links between the domain model and relevant software entities in the rest of the system. This will allow a developer, for example, to navigate between domain entities and the corresponding software entities responsible for their
business logic, presentation, or persistent state. “Live documentation” can be generated that will link to executable examples coming from the domain model. We would build on Evans’ notion of a “ubiquitous language” for developers and users [Eva04] by allowing any software fact to be presented in different ways depending on the context. Software tests can similarly benefit, since executable examples can serve as “test harnesses” that dictate how the software should behave.

**Evolution of domain models.** Typically domain models evolve much more slowly than the features that a software system must support. However when domain models do evolve, they can impact many parts of a software system. This can happen when new domain concepts arise, or aspects of existing domain entities change due to emerging business opportunities (e.g., the emergence of “self-driving cars” as a new domain concept).

In this subtrack we will explore ways to exploit executable domain models to track and manage the co-evolution of domain models and code. One way is through the explicit links that are maintained between domain entities and the software artifacts that implement them. Another way is through the executable examples. Since changes to the domain model will entail changes to the executable examples, their use as test harnesses will expose those parts of the system that are impacted by the changes.

This subtrack is thematically related to the research track on API client migration, and we anticipate that there will be opportunities for collaboration, particularly in shared case studies, and possibly in shared implementation infrastructure.

### 2.3.3 Domain-specific software quality

Our previous experience with software quality recommender systems such as QualityAssistant showed that quality advice must be tightly integrated into the development tools in order for it to have an impact on the development process. The approach is (i) to mine quality issues, (ii) to codify rules to detect violations, (iii) to provide contextual advice in the IDE, and (iv) to offer developers means to provide feedback, thus improving the quality of the rules and the advice given.

In this track we seek to generalize this approach by exploring domain-specific aspects of quality, inherent to the application domain, as opposed to general quality concerns, such as common “code smells.” We propose to investigate as an in-depth case study security concerns for mobile apps, and eventually study other quality concerns arising in the mobile app domain. The tremendous growth of the market in mobile devices in the last decade has led to an enormous demand for mobile applications. Developers of these apps focus on delivering functionality, and often lack the necessary skills and awareness to properly address user security and privacy concerns. We pose the research question as follows:

**RQ 3.** “How can domain-specific quality concerns and their corresponding corrective actions be effectively specified and monitored?”

The PhD student assigned to this track, Pascal Gadient (start of PhD studies Oct. 1, 2017), built a highly customizable workflow for large scale analysis of mobile apps, and applied it in his MSc research to study the symptoms and distribution of security code smells in mobile applications. He is currently refining these issues and investigating the potential risks of exposed web interfaces used in mobile apps. In this current year he will (i) conclude his study of the field of Android
security code smells, (ii) further elaborate on the risk assessment regarding unprotected web interfaces, and (iii) prepare critical code quality audits to leverage ground-truth data.

Particularly in the first and third tasks he will work closely with Dr. Mohammad Ghafari.

We propose the following three subtracks for the continuation project:

— **IDE support for security code smells.** In this subtrack we will focus on integrating software quality advice in the development tools to assist mobile app developers in producing reliable and secure code.

— **Risk assessment of web interfaces used in mobile apps.** Here we will investigate various web interface issues, measure their prevalence, and mine best practices to avoid common pitfalls.

— **Software quality awareness evaluation and growth.** We will explore ways to effectively raise software quality awareness among mobile app developers.

We see various opportunities for collaboration with the API client migration track, particularly concerning the case studies, and also in the fact that actionable security advice can take the form of migration paths to more secure API usage.

**IDE support for security code smells.** The security smells we have identified pose significant challenges to app developers who may lack expertise in security issues. This may be further aggravated by Q&A forums like Stack Overflow that promote a “copy-paste” culture to resolving technical issues without taking other quality aspects into account. Building on our experience in identifying and detecting security smells, and our experience with software quality recommender systems such as QualityAssistant, we will explore similar ways to offer context-specific, actionable security advice to developers in the corresponding IDEs, such as Android Studio.

This track poses considerable challenges beyond our earlier work on QualityAssistant in terms of severity and complexity of the issues, and in how to communicate the risks. A particular challenge is to mitigate false positives, since the proposed actions may entail considerable effort. Further challenges concern the encoding of rules to recognize threats, and identification of suitable development contexts for raising security concerns (*when* to communicate risks to developers).

**Risk assessment of web interfaces used in mobile apps.** A second case study we plan to investigate is that of unprotected web application interfaces (WebAPIs). In preliminary work, we have observed that numerous retrieved uniform resource locators (URLs) were indeed unprotected, potentially enabling diverse and severe attacks. In this study we plan to extract WebAPI URLs from our large scale mobile app corpus and establish measures to differentiate between interfaces that are safe and those neglecting quality requirements, particularly security and privacy concerns. In the first step we will set the focus on code security and reliability. Further research may include the question of how to develop safe APIs, and data model reconstruction based on decompiled bytecode of apps for which source code is not available. Through this second case study we expect to gain insights into a more general approach for mining, encoding, and monitoring domain-specific quality concerns, providing valuable feedback on actual perils of development trends, and convenient remedies to maintain code quality.
Another direction to explore is to expand the scope of quality concerns beyond security, for example, to consider usability issues specific to mobile apps.

**Software quality awareness evaluation and growth.** It is common hearsay that software developers are continuously under pressure to deliver new functionality, at the cost of quality concerns. In this subtrack we will attempt to verify to what extent this is true, and identify opportunities to mitigate this effect. We will carry out surveys with developers to identify both best practices for ensuring proper attention is paid to software quality, and further opportunities for experimentation. A particular challenge will be to assess the factors that impede attention to quality, be they priorities imposed from the business side (e.g., feature availability, release cycle), voluntary priorities set from the technical side (e.g., liabilities of external components, system responsibilities), lack of awareness of issues, or inadequate tool support.

We plan to collaborate with Sebastiano Panichella both in conducting empirical studies with developers, and in designing the recommender system.

### 2.3.4 API client migration

Software projects typically make use of third-party libraries and frameworks (which we will refer to generically as *components*). Components are produced by “upstream” developers for use by *client* code produced by “downstream” developers. To ensure the correct interoperation between components and their clients, components are usually fixed at a specific version that the client code must work with. Components and their clients evolve independently as bugs are fixed, security issues resolved, existing interfaces refactored, and new features introduced. Clients have to be migrated to the new versions of components to benefit from these improvements. A new version of a component is, however, often incompatible with older versions, thus posing significant effort for downstream developers to ensure the correct behavior of their client code with the new version of a component. As a consequence, components in use are often outdated, as downstream developers lag behind the evolution of components. This imposes security risks and missed opportunities to incorporate desirable features from newer versions of components. In order to reduce the effort of migrating client code to updated components, we plan to explore ways to enable a tighter coevolution of components and clients. Consequently, we pose the following research question:

**RQ 4.** “What is a suitable model for specifying, reasoning about, and automating API client migration?”

This research is being carried out by Manuel Leuenberger (start of PhD March 1, 2017) who completed his MSc thesis on the study of methods that return null values in client code. In the first year of his PhD research he has built a pipeline for large-scale API usage analysis, and successfully used it in a case study on the inference of nullable methods in Java systems. He is currently investigating the usage of cryptographic APIs in Android apps as a first case study for connecting code and domain knowledge to infer correct and incorrect usage of an API. Additionally, he is also investigating how we can detect locations that leak confidential data without encryption, and to propose ways to avoid such leakages.

In the course of the current year, we plan to study the key obstacles to API migration by carrying out a survey with software developers. We expect to gain insights from both upstream and
downstream developers into how they currently address migration and which issues an improved migration process should address. We also inspect migrations in existing software projects to infer commonly associated tasks within the migration process that could be automated to a certain degree. As an associated case study we intend to study the migration from Roassal, a visualization framework for Pharo, to its current release Roassal 2, and that of Roassal to Bloc, a completely different visualization technology that introduces new kinds of constraints.

The following two subtracks are planned for the remaining 26 months of this PhD research:

— **Lightweight semantic change model.** We explore how we can extract a lightweight model of changes in software projects by connecting static analysis techniques and a model of the technical domain of a component, to automatically propagate changes to client projects beyond the scope of classical source code refactorings. For changes that cannot be fully automated, *e.g.*, due to custom extensions, we investigate how a semi-automated process can assist a migration as a structured procedure.

— **Extended change model.** In this subtrack we explore how we can establish trust in the correctness of a migration by migrating pre-existing test-cases and generating new ones.

**Lightweight semantic change model.** Whereas migration is a well-addressed topic in the database community, the available tools and models in software engineering are rather primitive in comparison. Database migrations are first-class citizens in many web frameworks, enabling an automated migration of the database schema and data between different versions of the project. In contrast, migrations in software projects are scarcely provided as an automated process, but rather rely on deprecation annotations and textual migration guides. Existing automated source code migration approaches focus on rewriting method call sites, and replaying recorded or inferred refactorings. These approaches however only work in limited scenarios entailing small changes, *i.e.*, renamings, changed parameters, and the order of API invocations. We argue that these limitations stem from the narrow focus on purely syntactical changes.

The goal of this subtrack is to develop a change model that captures certain semantic aspects of changes: rather than consisting purely of syntactical modifications of a component, source code refactorings would link to an underlying model of the technical domain that captures the features provided by the component. For example, when the implementation of event listeners in a component changes from template methods to sophisticated listener interfaces, the classes implementing this interface need to be created, instantiated, and explicitly registered in the client. A semantic refactoring would capture these dependencies and ensure that all three aforementioned constraints are met to complete the refactoring. We can distribute these semantic changes to client developers alongside a new version of the component, so that the usage of the component’s API can be migrated by reapplying the refactorings on the client code, resolving non-automatable tasks by developer input to a guided process. Migrations can be written as a mapping of features and transformations to be performed on client code, so that usages of the component’s old API can be reliably detected, classified, and assigned to the appropriate refactoring process.

We plan to explore the use of formal concept analysis and clustering techniques to generate a prototype of a feature model that can afterwards be adjusted by a component developer. Component developers require knowledge about the usage of their APIs in order to create the migration
path with the least resistance. We will equip component developers with tools that allow them to estimate the impact of their breaking changes by mining client projects. Another tool will assist component developers with the creation and maintenance of the underlying domain model we rely on for our semantic change model. We will perform case studies on existing or planned migrations within the Pharo ecosystem to evaluate our change model.

We anticipate some synergies with the *Executable domain models* track, and in particular the subtrack on *Evolution of domain models*. Here we focus on support for migration rather than on the specifics of embedding and exploiting domain models.

We plan to collaborate in particular with Dr. Stéphane Ducasse, as he is leading many of the migration activities in Pharo, and has research expertise in automated migration.

**Extended change model.** The decision to migrate or not to migrate a component is often an economic one. Whereas the presence of a critical security issue in an old version of a component may force the need for migration, the tradeoff between opportunity and cost is not generally that clearly tilted towards migration. Improvements in performance and new features must be weighed against the effort required to perform the migration as well as the risk of introducing new bugs through an incorrect migration.

In this subtrack we explore how we can increase the confidence in the correctness of a migration by migrating existing tests and creating new tests that validate the migration itself. The migration of client tests is especially challenging since the component APIs are only tested indirectly through the APIs of the client, adding another level of indirection through which the changes imposed by the component migration must be reflected, *e.g.*, by setting up test fixtures differently.

To facilitate the assessment of the impact of a migration within a specific client project, we investigate how we can use our change model to estimate the complexity of migration within the scope of a single project. A realistic estimate of the cost and benefits of migration is crucial for client projects to make an informed decision in favor of or against it.

To evaluate the benefits and shortcomings of our approach, we will compare the quality of our migrations with the actual migrations between different versions of Lucene within Elasticsearch and Solr, a popular full-text search service.

Further directions we wish to explore in collaboration with Sebastiano Panichella include (i) evolution of other API-related artifacts, such as documentation, and (ii) predicting or recommending when API upgrades are possible.

### 2.4 Schedule and milestones

Here we provide a coarse timeline for each of the planned research tracks. Each row corresponds roughly to one person-year of a PhD project, with the exception of the last three rows (respectively 14, 9, and 2 months). We indicate for each year the subtrack that will be the main focus of the PhD student in that period, though it should be understood that the subtracks will in fact progress iteratively and overlap in time.
### Year 1 — 2019

| Speculative software analysis | Study of developer questions (Pooja Rani) |
| Executable domain models      | Example-driven domain modeling (Nitish Patkar) |
| Domain-specific software quality | IDE support for security code smells (Pascal Gadient) |
| API client migration          | Lightweight semantic change model (Manuel Leuenberger) |

### Year 2 — 2020

| Speculative software analysis | Speculative analysis of software data (Pooja Rani) |
| Executable domain models      | Embedding domain models (Nitish Patkar) |
| Domain-specific software quality | Risk assessment of web interfaces used in mobile apps (Pascal Gadient) |
| API client migration          | Extended change model, thesis writing (Manuel Leuenberger) |

### Year 3 — 2021/2022

| Speculative software analysis | Integration of analysis results into software process, thesis writing, PhD defense (Pooja Rani to 2021-12-31) |
| Executable domain models      | Evolution of domain models, thesis writing, PhD defense (Nitish Patkar to 2022-02-28) |
| Domain-specific software quality | Software quality awareness evaluation and growth, thesis writing, PhD defense (Pascal Gadient to 2021-09-30) |
| API client migration          | PhD defense (Manuel Leuenberger to 2021-02-28) |

### 2.5 Relevance and impact

The results of this project (as with previous projects) will be disseminated primarily through peer-reviewed full papers in top-ranked international conferences, such as ICSE (International Conference on Software Engineering), ICSME (International Conference on Software Maintenance and Evolution), SANER (International Conference on Software Analysis, Evolution, and Reengineering), and ICPC (International Conference on Program Comprehension). We also plan to submit papers to international journals such as IEEE TSE (Transactions on Software Engineering), ACM TOSEM (Transactions on Software Engineering and Methodology), Empirical Software Engineering, Journal of Software: Evolution and Process, and Science of Computer Programming.

We collaborate regularly with industrial partners to apply our techniques in extended case studies, or to carry out empirical studies (i.e., usability studies, interviews and surveys). In many cases the results of our research take the form not only of academic papers, but also tools or platforms (such as Moose and Pharo, both of which started as internal SCG projects) that have a considerable academic and industrial user and contributor base. Many of the software tools we have built (e.g., PetitParser, QualityAssistant, Moldable Debugger) have also been integrated into these platforms, and are used widely by both researchers and developers in this community.

All our publications, software, and research data are made available publicly either on our website, or on dedicated web sites (e.g., Zenodo.org), with the appropriate open source licenses (e.g., MIT, Creative Commons).
References


[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, Reading, Mass., 1995.


SNSF Proposal — Agile Software Assistance


SNSF Proposal — Agile Software Assistance


