Holistic Software Evolution

by Michele Lanza

In order to analyze and understand large, complex, and evolving software systems, we take a holistic stance by combining diverse sources of information.

Software evolution research has two objectives: given a system's present state, evolutionary information is used to understand its past, and predict its future.

But, where does evolutionary information come from? Traditionally, developers use software configuration management (SCM) systems, such as SVN and git. These systems record snapshots of the code, and store them in code repositories, which represent access and synchronization points for development teams. Given the widespread adoption and public availability of SCM systems, it is no wonder that researchers have recently been focusing on mining such software repositories, creating a new and challenging research field.

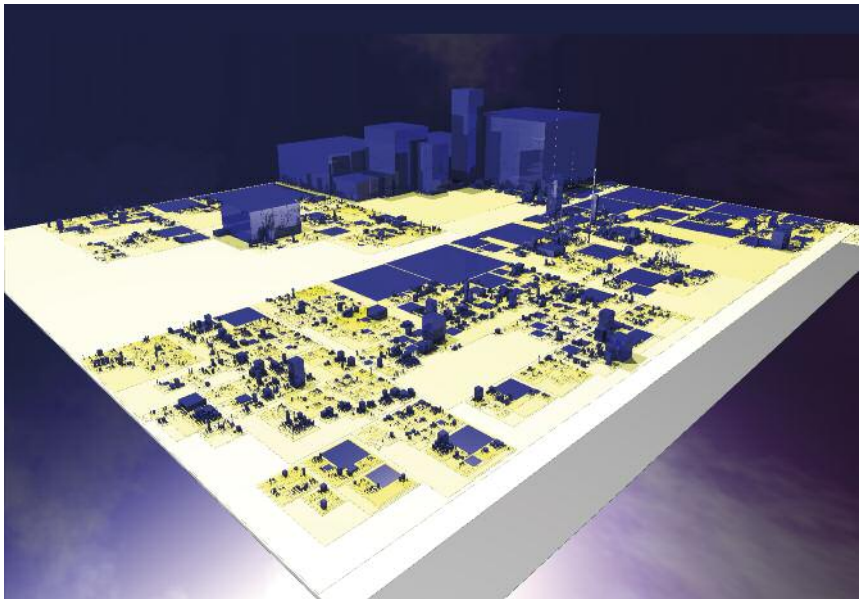

Figure 1: A depiction of the Eclipse IDE (approx. 4 million lines of Java source code) as a software city. The 1,900 packages making up the system are rendered as districts, which contain close to 29,000 classes, rendered as buildings. The height of the buildings represents the number of methods of the classes; the base size represents the number of variables.

With software evolution data it is easy not to see the wood for the trees. Besides SCM systems, there is a great deal of highly relevant data recorded in different types of repositories, such as bug trackers, mailing lists, newsgroups, forums, where system-relevant information is stored. The plethora and diversity of data represent a major research challenge in terms of how to extract relevant information and how to present it in a meaningful and effective fashion.

REVEAL - The REVEAL group at the University of Lugano approaches research from a holistic angle: The central theme is to piece together the various sources of information in a comprehensive whole, to have a better grip on the complex phenomenon known as software evolution. We are striking three new research paths to unify, reify, and see software evolution.

Unify - It is not obvious how to integrate information that comes from various sources and in a variety of forms. While certain repositories are easy to mine (especially source code, where the data is well structured), others store "humane" information, written by humans for humans, such as emails. We are currently experimenting with techniques dealing with fault-tolerant parsing (such as island parsing) and information retrieval (IR) to model the data in a unified way.

The PhD thesis of Marco D'Ambros (2010) demonstrated that by integrating various types of data, such as evolutionary data, code-specific data, and data on defects, it is possible to develop efficient defect prediction techniques that go beyond the state of the art. The ongoing work of Alberto Bacchelli is providing evidence for the usefulness of humane data: understanding what developers communicate about is often crucial for the understanding of what they produce.

Reify - Researchers often face the problem that the data produced by current mainstream versioning systems is of poor quality. The snapshot-based nature of SCM systems induces information loss: evolutionary data is only recorded when developers explicitly commit their changes to the SCM system. What happens in between is lost forever. Allegorically speaking this is similar to watching a movie when only every 10th frame is shown. While it may still be possible to understand the general theme of the movie, it certainly does not make for a great viewing experience.

We investigated this issue through the PhD theses of Romain Robbes (2008) and Lile Hattori (2012). The common theme is to complement classical versioning with information constantly recorded while developers work in their integrated development environment (IDE). The central idea is to embrace change by reifying it into a first-class concept. The consequence is almost perfect historical data. Through a number of studies and experiments we have demonstrated that the crystalline nature of the obtained information can fuel real-time recommender systems (such as code completion, conflict detection, etc.), which proactively help developers in their daily chores. Our

vision is to turn the "I" in IDE into "Intelligent".

See - What should we do with holistic information, once it's there? The challenge resides in the sheer bulk of data, and what it all means to a developer who is hardly interested in having more information at hand. The point is to have useful, understandable and actionable data. We believe that software visualization will play a major role in this context. It is the graphical depiction of software artifacts, and leverages the most powerful sense humans have: vision. While many shy away from the human-centric nature of this type of research, it is based on surprisingly simple notions that leverage how our brain processes visual input. The PhD thesis of Richard Wettel (2010) successfully explored a city metaphor to depict complex and evolving software systems, demonstrating that it is indeed possible to visually grasp a phenomenon as complex as software evolution.

Beyond - Software evolution research is only slowly coming of age, turning into a research field whose intrinsic complexity is further augmented by the fast pace of technical innovations. A holistic take is the only chance to prepare for unexpected consequences.

Please contact:
Michele Lanza
University of Lugano, Switzerland
Tel: +41 58 666 4659
michele.lanza@usi.ch
http://www.inf.usi.ch/lanza/

A One-Stop-Shop for Software Evolution Tool Construction

by Mark Hills, Paul Klint, Tijs van der Storm and Jurgen Vinju

Real problems in software evolution render impossible a fixed, one-size-fits-all approach, and these problems are usually solved by gluing together various tools and languages. Such ad-hoc integration is cumbersome and costly. With the Rascal meta-programming language the Software Analysis and Transformation research group at CWI explores whether it is feasible to develop an approach that offers all necessary meta-programming and visualization techniques in a completely integrated language environment. We have applied Rascal with success in constructing domain specific languages and experimental refactoring and visualization tools.

The goal of our research is to rapidly construct many different (experimental) meta-programs that perform software analysis, transformation, generation or visualization. The underlying motivations are (a) to do experimental research in software evolution and construction and (b) to transfer our results to industry in the shape of tools. We need innovative tools that contribute to understanding and improving very complex software systems.

The challenge is diversity: the mixture of programming languages and data formats one encounters is overwhelming when solving problems in software evolution. The range of questions to answer is even larger; they range from general (eg computing a call graph or a widely used metric) to application specific (eg checking in-house coding standards or migrating to a different API). As a result, software evolution problems are often solved with different tools that are glued together. The gluing itself introduces yet another kind of complexity.

Rascal is a domain-specific language that tackles this challenge by integrating all aspects of meta-programming. It is a language for performing any or all meta-programming tasks such as the analysis, transformation and visualization of existing source code and models and the implementation of domain-specific languages. Rascal is completely written in Java, integrates with Eclipse and gives access to programmable IDE (Interactive Development Environment) features using IMP (the Eclipse IDE Meta-Tooling Platform). We are aiming to provide a complete "One-Stop-Shop" for analysis, transformation, generation and visualization of software systems.

Rascal has many applications in the software evolution domain, for example, when studying the effects of choosing between design patterns. Design patterns provide reusable, named solutions for problems that arise when designing object-oriented systems. While in some cases it is clear which pattern should be used, in others multiple patterns could apply. When this happens, the designer has to carefully weigh the pros and cons of each option as applied both to the current design and to plans for future evolution of the system.

The specific design choice we focused on was between structuring AST-based (Abstract Syntax Tree based) language interpreters according to either the Visitor or the Interpreter design pattern. While it seems clear that either pattern will suffice from a functional point of view, it is unclear what the non-functional quality of the interpreter will be in each case. In theory, the Interpreter pattern might have lower method call overhead because it does not involve double dispatch, it should allow easier extension with new language features, and it should be easier to add local state to AST nodes. In theory, the Visitor pattern should allow easier extension with new kinds of operations on AST nodes and should allow better encapsulation of state required by such operations. The question remains how this choice affects the design in practice, given the plethora of other factors of software maintainability and efficiency.

Our research method is as follows. Given is an implementation of an inter-