Combining Development Environments with Reverse Engineering*

David Röthlisberger and Oscar Nierstrasz

Software Composition Group
University of Bern, Switzerland
{roethlis, oscar}@iam.unibe.ch

1. Shortcomings of current IDEs

Understanding and maintaining large software systems is a complex and time-consuming yet inevitable challenge. Most systems frequently change and evolve over time to meet new requirements. To perform these changes a software engineer must have an in-depth understanding of the inner structure and implementation of a system. However, the integrated development environment (IDE) usually provides little in the way of support to ease the understanding and changing of software systems.

A large body of research exists in the area of reverse engineering and many promising concepts to improve program comprehension have emerged, such as object or method histograms, polymetric views and class blueprints [3]. But the visualizations of reverse-engineered information about software systems are usually separated from the user's working environment, the IDE, and integrated into dedicated reverse engineering tools such as Moose or CodeCrawler. This means that a programmer working on maintenance tasks does not have access to important information about the structure or the dynamic behavior of a software system, such as a view presenting how a class communicates to other classes.

For instance, due to the lack of dynamic information (*e.g.,* collaborators of a class) the developer is forced to frequently browse a large code space without the benefit of goal-directed navigational support. Empirical studies report that an engineer performing maintenance tasks on a system spends at least 35% of the time in navigating dependencies between source artifacts (*e.g.,* which class uses which other classes) [2]. Because so much time is spent navigating source code, current IDEs obviously do not provide adequate means to support the navigation and browsing of software artifacts.

2. Combining Forward and Reverse Engineering within the same IDE

Integrating reverse engineering tools and visualizations into the IDE is one possible way to mitigate the navigational load weighing on a developer, because such visualizations quickly reveal information about the architecture and the structure of a system, hence the user can more efficiently locate important artifacts in a large code space.

However, just integrating reverse engineering tools into one single environment is not enough, because there is a mental gap between the goals of reverse engineering and development environments. A developer normally thinks in terms of static entities, *i.e.,* source artifacts such as classes and methods, and analyzes them to get an understanding of the behavior of a system. This metaphor of understanding and developing software is well established and cannot easily be changed. But this metaphor needs to be enriched, for instance we want to quickly get from an abstract feature (*e.g.,* a login feature in a Wiki application) to the developer's view, this is, the source artifacts realizing the feature. For instance, if a developer knows exactly with which other classes a given class dynamically collaborates to realize a specific feature, she can much faster navigate all entities relevant to a certain task, *e.g.,* correcting a defect in that feature [6].

What is needed is not only an integration of ideas, concepts, tools and visualizations gained by reverse engineering into the IDE, but a combination of the conventional, often purely structural view of today's IDEs with a dynamic view on a system and its features. In particular when working with object-oriented languages where polymorphism and late binding is applied (such as Java or Smalltalk) analyzing the dynamic be-

* In: *FAMOOSr 2007 (Ist International Workshop on FAMIX and Moose in Reengineering)*, 2007

havior is an efficient and reliable way to get a complete understanding of a program.

If we manage to tightly integrate dynamic information into the IDE and especially into the user's workflow to navigate and browse a software's code space, we may be able to greatly reduce the time required to understand the implementation of a software system, to maintain and evolve it. For short-term adaptations an IDE with integrated reverse engineering is certainly beneficial, but also long-term evolution is probably easier to achieve within such an IDE, especially if also historical information about a software system are accessible in the IDE [1].

3. Steps to Integrate Reverse Engineering into an IDE

We aim to study the following concrete enhancements and improvements of IDEs for program comprehension, navigation of source artifacts and maintenance of software systems:

- *Related Entities* By dynamically analyzing an application we can for instance reveal how a class collaborates with which other classes. We simply track all the receivers of messages sent from within a class. Making this information accessible helps the programmer to understand the dynamic behavior of a system, because she knows the dynamic collaborators of a class which are difficult to find out by only statically analyzing source artifacts.
- *Object Tracking* When just looking at the static source code we do not know what kind of objects a certain variable (instance or local) will contain when the system is running. By analyzing the system runtime we can memorize the objects that have been stored in a variable and *e.g.,* display the classes of these objects in the IDE while studying the static source code [4].
- *Feature Identification* By identifying source artifacts participating in a specific feature of an application, the developer can more efficiently navigate and also understand all source entities realizing that feature when she has to focus only on a subset of all the source artifacts of the whole system [6].

To actually realize these items we need to perform dynamic analysis. In Smalltalk tools and framework for dynamic analysis are already available, that's why we will develop a prototype of our IDE in Squeak Smalltalk. Even though the low level tools for gathering dynamic information already exist, the challenge is still to integrate and present this information in a useful manner.

This challenge consists of at least four technical issues we need to address: First, to collect dynamic information we have to run the application under study. This means that we have to continuously collect dynamic information on every run of the system to get information as accurate as possible. The second issue is the large amount of data which naturally emerges when performing dynamic analysis. This forces us to focus on a small percentage of available information that has proven to be useful and to filter out all other information. The third issue is to find good techniques to visualize and present the dynamic information dependent on the context the developer is in. A fourth issue we need to address is how to store and update the dynamic information accurately. In order to be useful the information has to be displayed without letting the user wait and it has to be accurate, so old information from an previous run of the systems needs to be invalidated.

A last but yet important step is the validation of the resulting IDE. For a serious validation of our work the developers need to be involved, this means that we will perform several empirical studies to report on the usefulness and the added value of combining reverse engineering with development environments.

References

[1] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[2] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 125–135, 2005.

[3] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[4] A. Lienhard, S. Ducasse, T. Gîrba, and O. Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[5] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, Nov. 2004.

[6] D. Röthlisberger, O. Greevy, and A. Lienhard. Supporting software maintenance with interactive feature driven browsing. In *Proceedings IEEE International Workshop on Visualizing Software for Understanding (Vissoft 2007) (tool demonstration)*, 2007.