Integrating Concerns with Development Environments

Ján Juhár

Department of Computers and Informatics, Technical University of Košice, Slovakia

jan.juhar@tuke.sk

Abstract

Program comprehension is an essential process in programming and many researchers report that it tends to take up to a half of a programmers' time during their work with a source code. Integrated development environments (IDEs) facilitate this process but there still are only restricted possibilities for narrowing the gap between concerns of a problem domain and a source code that implements them. In our work we utilize projectional properties of modern IDEs to make them able to process concern-related metadata and to provide customizable code projections. These projections preserve the original code structure while they show it from an alternative perspective regarding the contained concerns. We plan to evaluate the effect such code projections will have on program comprehension tasks.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Integrated environments; K.6.3 [*Software Management*]: Software maintenance

Keywords program comprehension, integrated development environments, software concerns, projectional editing

1. Research Problem

The primary factor that complicates a program comprehension process is a wide *semantic gap* created by abstraction between a problem domain and a solution domain. Much information about high-level concerns present in a problem domain is lost or scattered during a transformation to the source code. That makes it hard to answer questions about the intent behind a particular piece of code [7].

Another issue of program comprehension lies in a single dominant system decomposition. It is hard to find an optimal design or structure for a particular problem in a particular programming paradigm. The eventually used decomposition is influenced by code author's experience and by particularities of the task at hand. Other programmers working with the same code later must adapt to this decomposition and the mental model behind it, or if it does not suit them, perform a permanent refactoring with a hope that the original structure will not be needed later [9].

Many tools were created to support program comprehension. Findings of Maalej et al. [8] tell us that particularly IDEs are the most used tools for reading the code by industry programmers. The IDEs themselves contain tools designed to facilitate the comprehension and, as Kosar et al. show in [6], they improve the correctness of program comprehension tasks.

However, as IDE tools are designed to be universally usable "out-of-the-box", they work with source code-intrinsic information only. This means they work with any source code written in a supported language (optionally combined with a supported technology) without any additional information needed. This also means they operate only in a solution domain with implementation-level concerns and leave the discussed semantic gap wide. Furthermore, IDE code editors adhere to the standard file-based model and display source code in the same units as those persisted on a storage medium. Programmers cannot create custom code views that would more closely convey their mental model of a system and that would also preserve the original structure.

There were attempts to deal with these issues by changing storage formats of software systems. For example, storage layer of the software configuration management system *Stellation* [2] individually tracked class methods and fields instead of whole files and provided a query language for building *virtual files* out of these smaller code fragments. However, this approach required import and export steps through an XML format to achieve interoperability with external tools.

More recent tools tackle the discussed issues from a programming task point of view. *Mylyn* (formerly *Mylar* [5]) for *Eclipse* IDE filters software project structure views (e.g., package explorer) to the most often used elements in a specific task context. Canvas-based editors (e.g., *Code Bubbles* [1]) break the file-based paradigm and allow to create custom layouts of editable code fragments on a pannable canvas. These tools focus on specific code exploration or debugging tasks but lack in the area of layout reusability across

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2984047

different tasks. Moreover, they still use primarily source intrinsic information, although augmented with information gained by tracking programmer's activities.

2. Motivation

Concern-oriented tools, like those for feature location, produce some sort of metadata regarding concerns (or features) they identify in the source code. These metadata provide linkage between source code elements or fragments and concerns—concepts from a problem domain. And it is the fact that IDEs currently do not utilize such custom metadata that we, similarly to Nosáľ et al. [9], consider for the main reason why these tools cannot bring the code more closely to the problem domain.

Adding concern-related metadata to the source code is the core idea behind the *Concern-oriented source code projections* [10]. This approach uses code-level decorative markings (e.g., Java annotations) to assign high-level concerns to program elements. These metadata are utilizable by both external text-processing tools and also by IDE tools that can search for all usages of particular annotations. Additionally, the approach is complemented with an IDE plug-in that can construct a *projection* of all the code fragments marked with user-chosen annotations into a single *virtual file* that can be edited with an included editor. This is an attempt to address both the issue of the wide semantic gap and of the dominant system decomposition without giving up the direct tool interoperability.

The motivation for our work is in further development and evaluation of this approach in several aspects that could promote the role of high-level concerns in IDEs. First, the idea of concern-oriented projection can be extended to other IDE tools besides code editors. Next, we want to provide declarative means of building the projections from program elements belonging to chosen concerns. And finally, we want to focus on a visualization design for a virtual file projection with regard to its effectiveness for program comprehension.

3. Approach

The ability of modern IDEs to work with an abstract code representation was pivotal for transition from a pure textual code editing to a projectional editing, as described by Fowler in [4]. In an IDE supporting projectional editing the editor is parser-based[1], because source code is parsed into a special form of an abstract syntax tree (AST) and used as a model for structure-aware views (e.g., editors, project structure browsers) and operations (e.g., contextual code completion, refactoring). We intend to leverage this IDE property in our work presented in the following three steps: (1) integration of concerns, (2) projection building, and (3) projection visualization.

[1] Parser-based editors are distinct from AST-based editors that can be found in projectional language workbenches like JetBrains MPS.

Integration of concerns. Metadata describing concerns can be found in different forms. Above we described an approach that encodes concerns into annotations and they can be included similarly also in structured code comments [10]. Feature location techniques and tools often produce lists of relevant program elements [3]. These data can be used to augment the abstract code model built by an IDE in a unified format and then utilized in its views to make them aware of the identified concerns. We plan to implement this as an IDE extension able to transform and add concern-related data of different origins to the code model.

Projection building. A concern-oriented projection is a view of program elements that are related through associated concerns, and possibly also through code structural relations. There needs to be a way for a programmer to specify required properties of program elements that should be included in a projection.

Projection building can be realized through a special query language, a *projection query language*, that would operate on our augmented code model. Such query language can be based on existing program query languages, but it has to support querying custom metadata [9].

Projection visualization. The goal of concern-oriented projections is to provide alternative perspectives on a source code to facilitate program comprehension. Thus, the visualization needs to be understandable and unambiguous with regard to the original code. We expect this to be achievable by preserving a sufficient context from the original code fragment scope in a projected view.

An adequacy of a provided context will be significantly affected by a level of program elements granularity used to describe concerns in a code. We conducted a preliminary code tagging case study with 5 participants to learn which concern granularity levels would be used by programmers in source codes written in object-oriented and procedural languages. We found out that (in addition to the usual *class* and *function/method* granularities) the *statement* granularity level was used by every participant and that this level was used at least once in 80% of identified concerns. Moreover, for 47% of concerns identified in procedural code the *statement* granularity was the coarsest one.

Described results suggest that it might be beneficial to consider the *statement* granularity in the projection design. Assessing whether it brings real advantage for comprehension (considering the added value against increased cost of concern identification) will be the subject of the evaluation part of our work.

4. Evaluation Methodology

Our goal is to evaluate our design of concern-oriented projection regarding its efficiency for program comprehension tasks compared to standard IDE tools and other concern management tools. We want to address the two following research questions.

RQ1. Does our concern-oriented projection constructed on the basis of multiple concern-related metadata make the program comprehension more effective than the tools providing the metadata?

RQ2. Does the *statement* granularity level of a concern-oriented projection increase effectiveness of programmers in program comprehension tasks compared to a projection restricted to coarser granularities?

Experiments. In order to answer *RQ1* we will prepare an experiment with a software project with concern-related metadata originating from several tools. We will design a program comprehension task and assign it to two groups of participants. Both groups will have the metadata available, but the first one with the original tools and the second one with our projection. By tracking metrics like time needed to complete the task, correctness of the solution, number of code navigation operations, combined with a questionnaire for qualitative evaluation, we will evaluate the effectiveness of participants.

For answering *RQ2* we will prepare an experiment with a software project having both a simpler *method*-level concern granularity and a derived, more detailed *expression*-level concern granularity. Two groups of participants will have our projection tool available during a program comprehension task, in which they will be required to modify some project features with scattered code. Again, tracking multiple quantitative program comprehension metrics will allow us to evaluate effectiveness of participants. In the evaluation we will need to consider also the additional effort needed to create the more detailed version of concern description.

Acknowledgments

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-0008-10.

References

[1] A. Bragdon et al. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of CHI*, 2010.

[2] M. C. Chu-Carroll, J. Wright, and A. T. T. Ying. Visual separation of concerns through multidimensional program storage. In *Proceedings of AOSD*, 2003.

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. In *Journal of Software: Evolution and Process*, 25(1):53-95, 2013.

[4] M. Fowler. Projectional Editing. `http://martinfowler.com/bliki/ProjectionalEditing.html`.

[5] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of AOSD*, 2005.

[6] T. Kosar, M. Mernik, and J. C. Carver. The impact of tools supported in integrated-development environments on program comprehension. In *Proceedings of ITI*, 2011.

[7] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Proceedings of PLATEAU*, 2010.

[8] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the Comprehension of Program Comprehension. In *ACM TOSEM*, 23(4):31:1-31:37, 2014.

[9] M. Nosáľ, J. Porubän, and M. Nosáľ. Concern-oriented source code projections. In *Proceedings of FedCSIS*, 2013.

[10] J. Porubän and M. Nosáľ. Leveraging Program Comprehension with Concern-oriented Source Code Projections. In *SLATE*, 2014.