

A Review of Source Code Projections in Integrated Development Environments

Ján Juhár and Liberios Vokorokos

Department of Computers and Informatics

Technical University of Košice

Letná 9, 0420 00 Košice, Slovakia

Email: {jan.juhar, liberios.vokorokos}@tuke.sk

Abstract—The term *Projectional editor* is commonly used for tools that can work directly with the program’s abstract syntax tree. They are able to provide different views of the program, according to the specific editor used. The ability to look at the program from multiple views is often requested as a mean to simplify program comprehension. During their evolution, the Integrated Development Environments were equipped with tools that provide such possibilities. Many of them already work with the parsed abstract syntax tree of the code and thus can be considered for projections. In this paper we review projections available in 6 widely used IDEs. The review categorizes existing projections and shows that significant number of IDE tools depend on the knowledge of program structure, but also that data from other integrated tools are used to enhance the projections.

I. INTRODUCTION

THE ability to evolve a software system depends on the programmer’s ability to comprehend its source code. The source code comprehension (also program comprehension) is a cognitive process that involves analysis of the source code and retrieval of information and knowledge about the analyzed system. This process tends to take up to a half of a programmer’s time during the software development and maintenance [1], [2].

The main hindrance that programmers face while comprehending the source code is the wide semantic gap that exists between the problem domain and the solution domain. Due to this gap, the mapping of a specific program feature to the code that implements this feature (or vice versa) is not straightforward. Factors like programmer’s personality, experiences and skills, combined with the nature of the task at hand have a strong influence on this process [2]. Program represented as a source code enforces the single structure chosen by its author onto all programmers that will work with it later, regardless of whether they will intend to reuse, extend or modify it. This is the reason why the tools that are able to provide different, context-driven views of the code are requested during the comprehension process [3].

The goal to aid programmers in the program comprehension process stands behind many tools. The Integrated Development Environments (IDEs) represent the most complex toolset for working with a source code. The ultimate purpose of an IDE and all of its tools is to cover all phases of the software life cycle and to increase the productivity of programmers. For this reason, an IDE typically contains code editors, browsers and

analyzers, refactoring and build automation tools, debuggers, and other tools. The research on the comprehension strategies of professional programmers by Maalej et al. [2] points out the importance of integrated environments: comprehension tools that are not a part of the IDEs are virtually not used at all in the practice.

The main advantage modern IDEs have over pure text editors (even the advanced ones) is that they can “understand” the structure of the source code. After loading the source code contained in text files, they parse it into an abstract representation of the code – a form of its abstract syntax tree (AST). Many operations that IDE performs, e.g., refactoring or contextual code completion, are performed against this AST [4]. Whenever the code or the AST changes, the other is accordingly updated to stay in sync.

There are IDEs that take this idea to the next level and use an AST as a base program representation. The program is stored in the files as a serialized form of this AST, e.g., using XML notation. After loading such file into the editor, a *projection* of this base representation is presented to the programmer. There is no need for parsing and when the programmer is editing the program, he or she is basically directly editing the AST [5]. Such base representation abstracted from concrete language notation can be presented through projections in multiple forms and consequently the notation that programmer deals with can be tailored to best suit a particular domain. Editors of such IDEs are therefore called *projectional editors* [4], [5]. The most notable example of such IDE is the JetBrains Meta Programming System (MPS) which is used not only to develop programs in domain specific languages, but also to create these languages along with the appropriate language notations and editors.

In the case of MPS-like tools the idea of projections is well-defined. It is the core of their functionality. As we already hinted, modern source-based IDEs also operate on the abstract representation of the code, even though it needs to be parsed from the textual notation and it exists only during code editing. Yet, this gives an opportunity to exploit the idea of projections even by these IDEs.

We believe that source code projections are used by source-based IDEs on a large scale. However, the term *projection* is not well established in their context. This may be the reason why tools that use them are seldom recognized or referred to

as projectional. Our goal in this paper is to identify, review and categorize projections that are used by popular IDEs.

II. INTEGRATED DEVELOPMENT ENVIRONMENTS AND PROJECTIONS

There are many tools that programmers can use during development for code editing. They range from pure text editors that provide the basic text manipulation operations, through the advanced ones that add support for syntax highlighting or word completion, to fully featured IDEs with all the tools mentioned in the previous section. However, there is no generally recognized definition of what exactly is an IDE. Due to a great number of existing tools and an effort to differentiate themselves, there is no clear boundary between what is still “only” a text editor and what is already an IDE.

For our purposes, we have already laid out some requirements on what we consider for an IDE, mostly with regard to projections we want to explore. We will understand an integrated development environment as an application that is designed to support the majority of the software development life cycle – that is, at least the implementation, testing and maintenance phases – by providing an environment for programmers that includes tools for the associated tasks. In addition, we will require that the included tools (not necessarily all of them) are able to take advantage of the program structure – they should operate on the abstract representation of the source code.

As for the term *projection*, in the context of the MPS-like projectional editors it is used to represent an interactive, customizable rendering of the AST [5]. This can be true even for the source-based IDEs that operate on a parsed AST, though possibilities are limited here by the need to preserve parsability of the base source code. Through projections, source code can be conveyed in multiple views. The source code projections can be thus considered for a mapping between a set of base source code structures and a set of dynamically created views [6]. Such views usually focus on some aspect of the system and convey it in a concise way. Different projections can be useful in different contexts, but generally they can have positive effects on program comprehension by providing a higher-level view of the system.

III. A REVIEW OF SOURCE CODE PROJECTIONS

In order to select which IDEs will be used for the review of existing code projections, we referred to their popularity ranking *Top IDE Index*¹ that was compiled from Google search trends as of April, 2015. We focused on the first ten ranks of the index, as listed in the table I.

However, the criteria according to which the tools were added to the index were less strict than our understanding of an IDE. As a result, the index also contains tools like Vim, Emacs and SublimeText. Although these are very flexible and extensible tools for programming, we consider them for code editors, because they work with the source code only

¹<http://pypl.github.io/IDE.html>, accessed April 2015

TABLE I
SELECTION OF IDEs FOR THE REVIEW

Rank ¹	Name	Category	Reviewed	Version
1	Eclipse	IDE	✓	4.4.2
2	Visual Studio	IDE	✓	2013 Ultimate
3	Vim	Editor	–	–
4	NetBeans	IDE	✓	8.0.2
5	XCode	IDE	– (n/a)	–
6	SublimeText	Editor	–	–
7	Komodo	IDE	✓	9.0
8	IntelliJ IDEA	IDE	✓	14.1
9	Emacs	Editor	–	–
10	Xamarin	IDE	✓	5.9

at the textual level. Thus, these were ruled out of the review. Additionally, we had to exclude XCode as it is available only for Apple Mac platform, to which we did not have access. Six remaining IDEs marked in the “reviewed” column of the Table I were used in the following review of existing projections. The reviewed versions of used IDEs are also listed in the table.

The review of projections was conducted by evaluating features of main application menu and code editor of each selected IDE for projectional properties. Within the single IDE, the features were checked for multiple languages and project types.

Four categories of projections were identified. In the following subsections we describe these categories and list the relevant projections along with their requirements.

A. In-editor Projections

The code editor is a very significant part of each IDE. It was already mentioned that it projects editable representation of the AST constructed by parsing the source code. Of course, this projection will work only for languages that are supported by the IDE. Code editor displays the loaded file with exactly the same content as persisted on the storage medium. The projectional properties manifest themselves in the additional features that augment the text, and these are reviewed below.

The first is the *code highlighting*. It is able to convey information not only about the code syntax (e.g., by highlighting the keywords of the language), but also about its structure. It takes into account the scope of the variables – it distinguishes between local and global ones even if they have the same name. Furthermore, it can highlight all occurrences of the same program element (see fig. 1), identify unused statements and more. This is where the structural information provided by the parsed AST are exploited. However, the level to which the IDE is capable of these features depends on the detailness of the parsed AST and also on the language properties. The more a language has “dynamic”² properties, the less complete (or reliable) knowledge about actual program structure during

²Dynamic languages are mostly defined by support for dynamic typing or powerful reflection that allows extensive run-time modifications of a program behavior.

runtime can be obtained from the static structure of the AST. These properties have influence on all projections that utilize the AST.

A related projection is the *error highlighting*, which brings another informational layer to the code editor. The simplest to detect are syntax errors. Strongly-typed languages can benefit from possibility to check type assignment and visualize the errors.

Another common in-editor projection is the *code completion*, visualized as a pop-up list of available program elements in the specific context. Yet again, the knowledge of structural relations of elements in the edited source code is required for realization of this feature and the level of list completeness depends on the AST detailness and language properties.

Apart from above described in-editor projections that are available in every selected IDE, there are some that exist only in one of them. IntelliJ IDEA is able to project method or class implementation in the pop-up window over the selected class or method name with action called *Quick Definition* (see fig. 1). It is a different form of the *go to definition* projection (described in section III-C) that causes less navigational overhead, as the pop-up can be easily closed and user does not need to switch to other editor tab or window. The projected code is, however, not editable. Visual Studio provides feature called *Peek definition*, which creates a projection with similar purpose. This one is inserted directly in-line within the editor, as can be seen in fig. 2, and is fully editable. The use of this projection recursively inside the in-lined view replaces the whole view and adds so-called “breadcrumbs” for switching among the already opened views.

Source code in the editor can be further augmented by meta-data available in the AST. To give an example, the presence of documentation annotation `@deprecated` in a Java code can be projected by crossing out any usage of the annotated element in the editor to visually warn the programmer about usage of a deprecated API.

In-editor projections are also used to project information not obtained from the AST. While running the debugging session, the IntelliJ IDEA projects the actual values of the variables next to their occurrences in the code editor as the user steps

through the program execution. *CodeLens*, a feature of Visual Studio, decorates declarations of methods with a reference counter, as displayed in fig. 2. In addition to this, it can show a unit tests passing score and a code changes history, if the data are available. Only the reference counter value can be obtained by analyzing the AST of the program. The later two pull required data from the integrated test runner and version control system, respectively.

B. Structure Projections

Each of the selected IDEs contains tools that provide overview of the project structure. The most basic one – the file browser – can be considered for an *identity* projection of the project’s files, with exactly the same structure as stored on a storage medium.

With the exception of the Komodo IDE, the selected IDEs contain panels that show hierarchical, tree-like project structure according to packages, namespaces, modules, classes, or other structural elements of the programs. These high-level views of the program structure are augmented with graphical symbols, informing the user about the visibility of the structural elements or distinguishing their type (class, abstract class, interface, and others).

The view of the same tool in a particular IDE differs depending on the used language or the project type. For example, in IntelliJ IDEA, *Project* panel shows more detailed structure in the case of Java files than, say, Python or JavaScript files. When working on a web project in the NetBeans IDE, the *Projects* panel extends the tree structure to include even the remote files that are linked from inside of the HTML pages.

Individual IDEs also differ in the level of the details displayed by these tools. Particularly detailed is the Package Explorer of Java projects that is available in the Eclipse IDE. As shown in fig. 4, it goes down to the level of individual class members and also provides their signature. At the file level the view is further extended with repository information of the latest edit. Another tool that goes deeper into the program structure is the *Architecture Explorer* in Visual Studio. In addition to listing the class members, it can recursively show

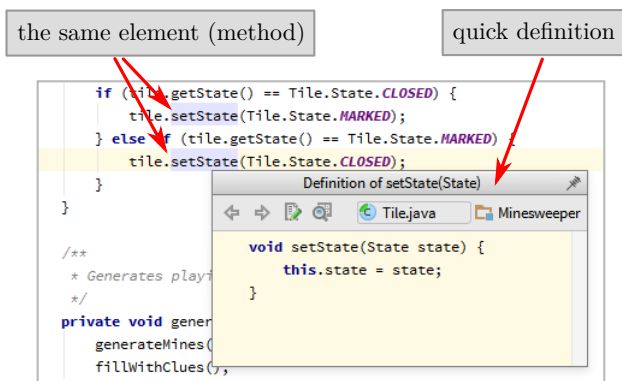


Fig. 1. code highlighting and *quick definition* in IntelliJ IDEA

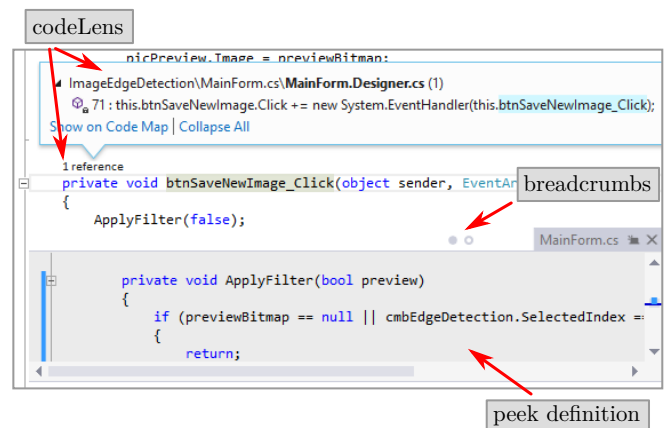


Fig. 2. *codeLens* and *peek definition* in Visual Studio

all the methods called inside the implementation of a particular method. Moreover, it allows to create a graph that represents the selected call hierarchy.

Apart from tools projecting structure of the whole projects, all selected IDEs contain more focused tools that display detailed structure of the file currently viewed in the editor. This is intended to speed-up the navigation inside a single file. A related to this projection is the *breadcrumbs* panel that shows context of the currently edited part of the file based on the position of the caret in the editor. Eclipse and IntelliJ IDEA contain also projections that can display *type hierarchy* of classes and *call hierarchy* of methods.

C. Search and Go-to Projections

The next group of projections relate to searching for specific program elements across the project. The simple text-based search is by the IDEs extended to take into account the source code structure. This helps to connect related elements and distinguish between the different elements represented with the same name (the same text).

One of the tools providing this kind of projection is used to search for all references of the same program element. This tool is called differently in each IDE, with names like *find usages* or *find all references*. It is useful for quick navigation and for discovering code dependencies. Another related tool deals with the comments. It displays all code locations where comment starts with the word “todo”, or other configured pattern, which makes possible to track tasks across the project (see fig. 4).

Similar are tools for quick navigation between different program element occurrences and within the inheritance hierarchy. These include tools with self-explanatory names like *go to declaration*, *go to implementation*, *go to super implementation* and *go to type declaration*. If they can navigate to only one place in the source code, there is no “view” created and the action implied by the tool’s name is immediately performed.

D. Domain-specific Projections

The projections reviewed so far drew the required information for their construction from the parsed AST. Many

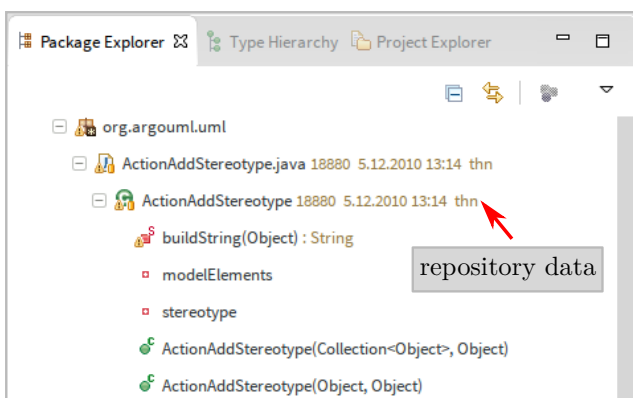


Fig. 3. Package explorer in Eclipse

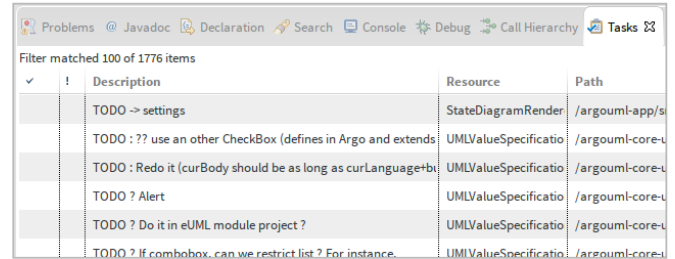


Fig. 4. TODO tasks in Eclipse

general-purpose IDEs (in our case Eclipse, IntelliJ IDEA, NetBeans and Visual Studio) have support for a number of software frameworks. And there are IDEs that are built specifically to support some frameworks (e.g., Xamarin for mobile application development). These IDEs take advantage of common application structures, conventional configurations and domain-specific APIs in order to simplify development of applications with supported frameworks. As a result, code projections that support specific framework features were found in the reviewed tools as well.

Applications that use a particular framework have (to some extent) common structure and this is exploited by these IDEs. Based on the known structure, program element across different languages can be interconnected. Web applications created with the Spring or Play frameworks in IntelliJ IDEA or with ASP.Net framework in Visual Studio can have code completion working for dynamic segments of page templates because the IDE “knows” which program element represents the context of the template.

An example of a projection based on a framework configuration is the ability of IntelliJ IDEA to view and edit classes annotated with the Java Persistence API’s annotations in the form of entity relationship diagram.

The common example of domain-specific projection among the reviewed Java-supporting IDEs – Eclipse, IntelliJ IDEA and NetBeans – is the graphical user interface builder for the Swing framework. Eclipse can project class inheriting from one of the Swing window components directly to its graphical representation and any edits made in this graphical view are reflected back to the code. To achieve the same functionality, IntelliJ IDEA and NetBeans use intermediary XML file that represents the layout of user interface components. Graphical representation is the result of projecting this XML file. In the other direction, the source code is generated from the XML. IntelliJ IDEA by default postpones this code generation to the compile time, while NetBeans updates code on each change of the design. Similar projections are available also in Visual Studio for Windows Forms framework and in the Xamarin IDE for creating graphical layouts for mobile applications. All these projections require the IDE to “understand” the API of the particular framework.

IV. PROJECTIONAL TOOLS IN THE RESEARCH

The research in the area of projectional tools is mostly associated with the tools like the already mentioned JetBrains

MPS. In their work [5], [7], Voelter et al. focus primarily on this language workbench. They discuss the concept of projections and possibilities they bring to the domain-specific language development, but also the associated issues of direct AST manipulation. They also deal with projections in the context of composition and extension of programming languages.

There is an ongoing endeavor in the research community to create tools that supports program comprehension. Few of those are generally viewed as projectional tools. However, many of the tools are implemented as IDE extensions and have projectional properties similar to those we described in the review.

Among these tools, the *Fluid source code views* [8] tool implemented for the Eclipse IDE is similar to the *peek definition* feature of Visual Studio that was reviewed in this paper. *Registration-based abstractions* presented by Davis et al. [9] are another example of IDE projections – in this case they change language syntax in the Eclipse IDE editor to achieve easier comprehension of frequently used coding patterns.

Another type of tools deal with the program concerns. The *Sieve Source Code Editor* [10] for the NetBeans IDE uses the structured code comments to create concern-oriented views of the source code. The *Code Bubbles* [11] and the *Code Canvas* [12] are alternative code editors for the Eclipse and Visual Studio IDEs, respectively. These provide projections of the source code that are abstracted from traditional file-based views and allow programmers to create free-form arrangements of the code they are working with.

V. CONCLUDING REMARKS

In this paper we presented a review of code projections that are available in today's integrated development environments. We identified four main categories of the tools that feature projectional behavior.

The *in-editor* projections enhance the textual notation of the source code with code highlighting or completion. The *structure* projections are available as a separate panels that create views of the system concentrated on its structure, providing thus higher-level view that is easier to grasp than the full source code with all of its details. The *search and go-to* projections are provided by the tools for searching the related program elements or navigating the class hierarchies. In the most cases, these three categories require for their functionality a knowledge that can be extracted from abstract syntax tree analysis. We found also projections that use other information sources, exploiting advantages of the tools that are available in the integrated environments. There are data projected from debuggers, version control systems or test runners. The last category contains *domain-specific* projections that rely on conventions of supported frameworks, such as common application structures and APIs designed for a particular domain. As a result, they can provide code completion across languages, project annotated classes as relational diagrams or enable graphical builders of user interfaces.

We have shown in the review that the projections are not only a matter of projectional language workbenches as they

are extensively used in many aspects of modern source-based integrated development environments. Their infrastructure is well-prepared for such functionality, which is also exploited by many research tools that are implemented as extensions of these IDEs. And as the MPS is approaching the usability of the standard source-based editing [5], the distinction between the AST-based and the source-based editors is becoming less apparent.

ACKNOWLEDGMENT

This work was supported by the project KEGA 008TUKE-4/2013 and by the Slovak Research and Development Agency under the contract No. APVV-0008-10.

REFERENCES

- [1] T. Kosar, M. Mernik, and J. Carver, "The impact of tools supported in integrated-development environments on program comprehension," in *33rd International Conference on Information Technology Interfaces (ITI'11)*, 2011. ISSN 1330-1012 pp. 603–608.
- [2] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the Comprehension of Program Comprehension," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 1–37, Aug. 2014. doi: 10.1145/2622669. [Online]. Available: <http://dx.doi.org/10.1145/2622669>
- [3] M.-A. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," in *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE, 2005. doi: 10.1109/WPC.2005.38. ISBN 0-7695-2254-8 pp. 181–191. [Online]. Available: <http://dx.doi.org/10.1109/WPC.2005.38>
- [4] M. Fowler, "Projectional Editing," 2008. [Online]. Available: <http://martinfowler.com/bliki/ProjectionalEditing.html>
- [5] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards User-Friendly Projectional Editors," ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8706, pp. 41–61. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11245-9_3
- [6] M. Nosál, J. Porubán, and M. Nosál, "Concern-oriented source code projections," in *Proceedings of the 2013 Federated Conference on Computer Science and Information Systems*, Kraków, 2013, pp. 1541–1544.
- [7] M. Voelter, B. Kolb, and J. Warmer, "Projecting a Modular Future," *IEEE Software*, pp. 1–1, 2014. doi: 10.1109/MS.2014.103. [Online]. Available: <http://dx.doi.org/10.1109/MS.2014.103>
- [8] M. Desmond, M. Storey, and C. Exton, "Fluid source code views," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006. doi: 10.1109/ICPC.2006.24 pp. 260–263. [Online]. Available: <http://dx.doi.org/10.1109/ICPC.2006.24>
- [9] S. Davis and G. Kiczales, "Registration-based language abstractions," *ACM SIGPLAN Notices*, vol. 45, no. 10, p. 754, Oct. 2010. doi: 10.1145/1932682.1869521. [Online]. Available: <http://dx.doi.org/10.1145/1932682.1869521>
- [10] J. Porubán and M. Nosál, "Leveraging Program Comprehension with Concern-oriented Source Code Projections," *3rd Symposium on Languages, Applications and Technologies. OpenAccess Series in Informatics (OASlcs)*, vol. 38, pp. 35–50, 2014. doi: 10.4230/OASlcs.SLATE.2014.3. [Online]. Available: <http://dx.doi.org/10.4230/OASlcs.SLATE.2014.35>
- [11] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeptura, and J. J. LaViola, "Code bubbles: a working set-based interface for code understanding and maintenance," in *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*. ACM Press, Apr. 2010. doi: 10.1145/1753326.1753706 pp. 2503–2512. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1753326.1753706>
- [12] R. DeLine and K. Rowan, "Code Canvas: Zooming towards Better Development Environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 2. ACM Press, Jan. 2010. doi: 10.1145/1810295.1810331 p. 207. [Online]. Available: <http://dx.doi.org/10.1145/1810295.1810331>