Embedding Programming Context into Source Code

Alexander Breckel and Matthias Tichy
Institute of Software Engineering and Compiler Construction
Ulm University, Germany
Email: {alexander.breckel, matthias.tichy}@uni-ulm.de

***Abstract*—Programmers use diverse tools for code understanding to access various types of context information like interface definitions, revision histories, and debugging values. Integrated development environments support specialized visualization mechanisms for such context types. While these mechanisms in principle enable programmers to access required information, the diversity of visualizations as well as the distance between code locations and related information may slow down development. We present a generic approach to embed various types of context uniformly into the main source code view in close proximity to the relevant source code by using a concept called code portals. Furthermore, embedded content can be organized and manipulated directly using operations already familiar to programmers. We illustrate the approach using different types of context, and present preliminary results of a qualitative study indicating that our approach is usable and improves program comprehension and productivity in general. The approach is implemented in a prototypical source code editor.**

I. Introduction

Programmers spend a significant part of their time reading and comprehending source code by accessing various types of programming context [1], [2]. They navigate between connected code fragments, look at type signatures of called and calling methods, and read documentation comments and revision histories. During debugging, they inspect execution states and control flow by checking the values of variables and expressions.

These different types of programming context are often distributed over various artifacts like source code files, documentation files, and databases, which creates the need for ways to access and keep track of this context. Integrated development environments (IDE) provide such ways through a set of specialized features and visualizations dealing with context in various representations [3].

However, empirical research has shown that users in general [4] and developers in particular [5], [6], [7] use only a small subset of generic features and disregard many available specialized features. Potential reasons for that are the time and difficulty to learn and remember specialized features as well as their proper usage. Instead, they often use widely applicable features and perform subsequent manual adjustments [8]. Also, since many types of information are presented through subwindows, a simultaneous usage of different program comprehension features often leads to visual clutter.

In this paper, we present our approach to manage programming context by embedding it into the main source code view, close to the location where each information is relevant. The content is not displayed as tool-tips, overlays or nested views, but instead textually included into the surrounding source code with low visual overhead. We use a novel mechanism for this that we call *code portals* [9], a lean and non-destructive way to augment textual documents with additional content. While allowing arbitrary textual embeddings, the resulting code document remains fully editable and compilable.

We see several benefits of embedding programming context directly within the main source code view: First, the visual proximity to the code leads to less scrolling, searching, and context switching. Secondly, all information is anchored to specific code locations, which provides a persistent mapping and allows multiple simultaneous embeddings. And finally, embedded content can be manipulated using tools and concepts already familiar to programmers.

We have implemented our approach including several program comprehension features like code inlining and live-evaluation in a stand-alone source code editor. We evaluated our approach by conducting a qualitative study using this implementation.

The main contributions of this paper are:

- The concept of code portals to embed textual information into source code in order to aid program comprehension.
- Different ways how existing program comprehension features can be adapted to utilize code portals.
- Preliminary evaluation results confirming the applicability and usefulness of our approach.

In the following section, we present ways in which programming context can be embedded into source code, and illustrate them with concrete examples. Section III provides a short description of our text-editor implementation. We further present preliminary results of a study in Section IV. After a discussion of related work in Section V, we conclude and give an outlook on future work.

II. Embedding Programming Context

Code portals enable the embedding of arbitrary *textual* content into code documents. While a plain-text representation seems overly restrictive at first, we have found that many types of programming context are predominantly text-based or can be converted to plain-text without a loss of information. For our purposes, we distinguish between the *explicit* and *implicit* programming context, which will be further discussed in the following two sections.

A. Explicit Context Information

Often, developers access context information in the form of definitions and documentation of currently used APIs. Such definitions typically consist of code fragments that are located in physically accessible source files. Normally, an editing environment would allow us to navigate to these locations, or display them inside tool-tips. Our approach, however, allows developers to embed these fragments seamlessly into the code they are currently working on, by opening a nested region showing the respective code fragment. We call these nested regions *code portals* to emphasize the link to its original code location.

To illustrate this, suppose a programmer is writing a class definition in Java that implements an existing interface. In order to provide correct method implementations, the programmer needs access to the corresponding interface definition. Our approach is to create a code portal to the interface definition, either by hand or by using an automated editor feature:

```
 1  class PersistentDB implements DB {
76  interface DB extends Serializable {
77      void store(String k, String v);
78      String retrieve(String k);
79  }
 2  }
    ...
76  interface DB extends Serializable {
77      void store(String k, String v);
78      String retrieve(String k);
79  }
```

The code portal, visible at the top, is highlighted with a colored border and background, but otherwise uses the same font and mono-space layout as the surrounding code. The dotted area at the bottom represents the source region of the code portal further down in the file. Most text operations can safely ignore all portal boundaries and treat the document as a linear sequence of characters when performing their modifications. This, for example, allows the developer to copy the two embedded method headers into the surrounding class and subsequently adapt the code. Furthermore, the visible context can be expanded by also embedding the interface definition of `Serializable`.

Due to the existing link, modifications within the code portal affect the original code location. Conversely, changes to the original code locations, e.g., when performing refactoring operations, are also immediately visible within the code portal. This allows the programmer to extend the interface definition, or work on both locations in parallel without having to navigate to the corresponding code locations.

Code portals can also be nested. A typical use-case for this is to inline identifiers, or to unroll recursive function calls, by replacing them with portals to their respective definitions. In fact, we do not restrict code portals to well-formed hierarchies, but instead allow arbitrary nestings and overlappings to support intermediate and syntactically malformed editing states which often occur during programming. If, at some point, a code portal is no longer needed, it can be closed and removed from the code document without any side-effects to the source file. Nested code portals are closed recursively.

The aforementioned example embedded a definition in the location of its usage. Sometimes, however, the opposite direction can also be useful, namely to embed usages of an identifier at its definition site. Imagine a scenario where a new parameter needs to be added to an existing function definition. Normally, this also entails modifications to all call sites. By embedding these scattered locations alongside the function definition, the programmer can keep all affected code fragments in view simultaneously, as shown in the following example:

```
 1  int update(int x, bool y) {
13  int error = update(0, false);
24  int k = fetchKey(name);
25  int r = update(k, true);
26  assert(r != 0);
77  if(update(x, false) != -1) {
 2    verify(x);
```

Here, a new parameter needs to be added to the function `update`. Providing a simultaneous embedding of all call sites simplifies assessing the amount of required changes. Additionally, since code portals are editable, the developer can also update all usages collectively in one place. To facilitate this, we support a subsequent resizing of code portals to include more context, as shown in the green portal.

Of course, other types of context can be embedded as well, as long as the content can be referenced bidirectionally. This includes, for example, similar code fragments, code clones, or other relevant code locations that one wants to keep visible while programming. The ability to easily resize and reorganize existing code portals allows a work-flow where a programmer keeps several sets of relevant code fragments open in different code locations, and proceeds to adapt them as his needs change during the course of a programming session.

B. Implicit Context Information

In contrast to context that is explicitly stored somewhere within the project, programmers often need access to context information that can not trivially be referenced and therefore embedded in editable code portals. This is the case for information that needs to be generated from the underlying source code, like evaluation results or inferred types. The following code listing shows a Haskell expression containing several code portals containing such context:

```
1  let x = 42 :: Int
2      y = 2 + 3 == 5
3  in x - y == 37
```

The code portal in line 1 contains the inferred type `Int` of the expression `42`, whereas both code portals in lines 2 and 3

contain evaluation results of the expressions to their left. This can be useful for testing and debugging purposes, as well as for inspecting code written by someone else. In our implementation, we update all inferred values with each keystroke, which removes the need to start the program or switch to an interpreter. Our evaluation indicates that this immediate feedback improves program comprehension and productivity in general, and also allows example-driven programming.

Besides displaying related and inferred content, code portals can also be used to access the revision history of source files. Here, developers are often interested in comparing code artifacts of different versions, or in showing currently uncommitted changes. Existing revision control systems offer visualizations of this data. However, our implementation displays this information directly within the editor view by embedding a read-only delta of unstaged changes:

```
1  <package>
-    <uid>agent007</uid>
2    <version>1.7</version>
-    <name>AgentControler</name>
3    <name>AgentController</name>
4  </package>
```

Red code portals contain lines that have been removed since the last commit, whereas green regions highlight new or modified lines. Developers can use this to get an overview of their current changes, or selectively undo individual modifications. Also, this helps in resolving merge conflicts.

Similar to this, it can also be useful to access remote context like bug reports. An issue ID included in, e.g., a documentation comment, can be replaced by a code portal showing its description and meta-information, which helps in documenting and reconstructing bug-fixes. However, complex formatting and non-textual media so far can not easily be converted into an appropriate textual representation suitable for our approach.

The fact that embeddings contain plain-text and can be nested makes it possible to chain or compose different editor features to achieve helpful synergies. Suppose, for example, a developer has embedded some related definitions to comprehend a piece of code. To further inspect the reasoning behind the inserted definition, he recursively embeds its version history, revealing a linked issue number, and proceeds to inline the respective issue description. As each operation presents its results in a uniform way within the editing view, all of this can happen without scrolling or navigating to different locations. The wide applicability of individual features lets programmers quickly perform complex operations and often replaces the need for specialized features.

III. Implementation

We have implemented our approach in a prototypical text editor called Inline [10] (Figure 1) to demonstrate feasibility and further explore the possibilities of context embedding. All features described in this paper have been implemented in the

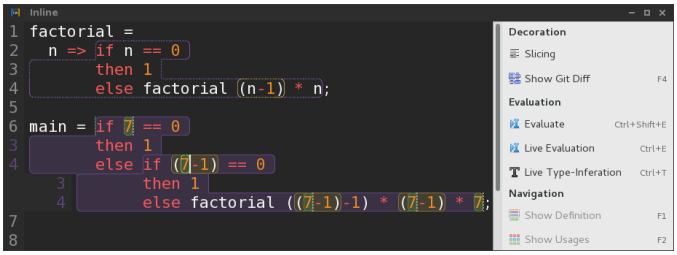

Fig. 1. Screenshot of Inline showing two recursive function call inlinings

presented ways, namely inlining, code search, revision history, live evaluation, and re-organization of embeddings.

The addition of code portals breaks the usual linearity of text documents by allowing text fragments to appear out of order and in multiple locations. Unfortunately, most features of existing IDEs, like Visual Studio or Eclipse, are built with an inherent assumption that textual content is linear, and would require major invasive changes to support code portals. We therefore chose to implement our approach as a standalone prototypical source code editor.

IV. Evaluation

We have conducted a qualitative study to further evaluate our approach and answer the following questions:

- RQ1 Is the concept of code portals inside a code document usable by programmers?
- RQ2 Does embedding context into the source code view improve program comprehension?
- RQ3 How quickly are programmers able to learn the usage of code portals and our implemented features?

While a full evaluation is currently ongoing, this section includes a description of the settings and preliminary results. The study was conducted in individual sessions with a total of 9 participants from the field of computer science with varying degrees of programming experience. After a pre-questionnaire, each participant was given a 30-minute introduction to our Inline editor and the programming language used in the study, and the chance to get familiar with the available features. This was followed by 40-60 minutes of hands-on work, in which each participant was asked to solve a set of 6 programming tasks. Participants were asked to think aloud and were recorded using a screencast and a microphone. Afterwards, a semi-structured interview was conducted to further capture the overall experience.

The tasks were designed to be easily solvable using the aforementioned editor features, without always telling the participants which features to use. Available features were: showing or inlining a definition, finding and resolving all usages of an identifier, and a live-evaluation and type-inference of expressions. A custom minimalistic functional language with Haskell-like syntax and type inference was used for all tasks, with participants reporting varying degrees of experience with functional programming languages.

A. Feedback to code portals in general

All participants, including those not familiar with the concept of code portals, quickly learned their usage and gained productivity within one hour of hands-on experience. They reported unanimously, that the concept of code portals in its current look and feel is intuitive, including the fact that code fragments can be viewed and manipulated in different locations simultaneously. Most participants mentioned that their typical programming workflow involves a lot of scrolling and navigating between related code locations, and that our approach would reduce this overhead significantly. However, as most editor features available in this study created code portals across the code document, some participants initially got confused as to where these portals are pointing to. Even though participants noted that this confusion probably will fade with extended usage, we plan to add more visual clues to make distant portal targets more traceable.

B. Feedback to specific features

Most participants stated that the available feature set significantly helped them in understanding the provided source code. Especially the live-evaluation of expressions was used as a quick way to infer the semantics of functions. One participant, having little experience with functional languages, noted that the combination of live-evaluation and quick access to function definitions helped him understand the functional programming paradigm. Others confirmed that, at least for functional languages, the presented use-cases arise on a daily basis, and that our tool provides effective solutions.

A deep nesting of code portals, especially when inlining nested functions with short bodies, was confusing to some participants, and was generally not regarded as helpful. Concerns were also raised about the usefulness of code portals containing large code-fragments, as they often occur in imperative languages. We plan to address these issues by making the nesting levels of portals more visible, and supporting elision of large portals. Overall, the feedback was very positive, with multiple participants expressing their surprise that these features were not more common across modern IDEs.

V. Related Work

Academic literature and industrial development tools offer many different approaches to manage context. To allow developers to keep multiple code locations in view simultaneously, many editors make it possible to open and use multiple editor views side by side. Even though the developer can often choose between different layouts to optimize spatial utilization, this solution only scales up to a few open editor views. Some IDEs, like, for example, Visual Studio [11] and Brackets [12], offer ways to open nested editors showing related code artifacts. Both implementations, however, only support a single level of nesting, and introduce a spatial overhead similar to side-by-side editing views.

Different approaches like Code Canvas [3], Code Bubbles [13] and Jasper [14] allow a free placement of small, encapsulated editor views on a canvas, optionally visualizing semantic connections between code fragments. As each editor view contains only a few code artifacts instead of whole files, the available space can be utilized more efficiently. However, this has a large impact on typical programming workflow and introduces many new interaction concepts. Also, all individual editors are isolated, which makes it difficult to perform selections and modifications across editor bounds.

VI. Conclusion and Future Work

We have shown how different types of context can be embedded into source code documents to aid program comprehension using a concept we call *code portals*. In cases where the original location of context is locally accessible, embeddings remain fully editable. Otherwise, read-only code portals are used to present remote or automatically inferred context. The usage of code portals creates powerful and composable new ways of interacting with context, and provides a simple and uniform presentation medium for tool developers. Preliminary results from a case study indicate that code portals provide a practical extension of regular text editors that improves program comprehension. Also, the usage of presented features can be learned and productively applied within a matter of hours.

We plan to combine nested editors and code portals in order to support non-textual context types like rich-text documentation and dependency-graphs. Also, we want to inspect further ways to integrate our approach into existing IDEs.

References

[1] A. J. Ko, B. Myers, M. J. Coblenz, H. H. Aung *et al.*, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *Software Engineering, IEEE Transactions on*, vol. 32, no. 12, pp. 971–987, 2006.

[2] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 174–188.

[3] R. DeLine and K. Rowan, "Code canvas: Zooming towards better development environments," in *Proceedings of the International Conference on Software Engineering (New Ideas and Emerging Results)*, May 2010.

[4] E. Backlund, M. Bolle, M. Tichy, H. H. Olsson, and J. Bosch, "Automated user interaction analysis for workflow-based web portals," in *Proc. of the 5th International Conference on Software Business (ICSOB 2014), Paphos, Cyprus*, 2014, (talk slides).

[5] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *Software, IEEE*, vol. 23, no. 4, 2006.

[6] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 5–18, 2012.

[7] T. Roehm, R. Tiarks *et al.*, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 255–265.

[8] M. Vakilian *et al.*, "Use, disuse, and misuse of automated refactorings," in *ICSE 2012*. IEEE, 2012, pp. 233–243.

[9] A. Breckel and M. Tichy, "Inline: Now you're coding with portals." Submitted to *International Conference on Program Comprehension (ICPC 2016)*. [Online]. Available: https://goo.gl/cSAeHa

[10] "Inline," http://www.uni-ulm.de/en/in/pm/research/projects/inline.html.

[11] "Microsoft Visual Studio," https://www.visualstudio.com/.

[12] "Brackets editor," http://brackets.io/.

[13] A. Bragdon *et al.*, "Code bubbles: a working set-based interface for code understanding and maintenance," in *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 2503–2512.

[14] M. J. Coblenz, A. J. Ko, and B. A. Myers, "JASPER: An eclipse plug-in to facilitate software maintenance tasks," in *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*. ACM, 2006.