Leveraging Program Comprehension with Concern-oriented Source Code Projections

Jaroslav Porubän and Milan Nosáľ

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice
Letná 9, 042 00, Košice, Slovakia
`jaroslav.poruban@tuke.sk, milan.nosal@gmail.com`

Abstract

In this paper we briefly introduce our concern-oriented source code projections that enable looking at same source code in multiple different ways. The objective of this paper is to discuss projection creation process in detail and to explain benefits of using projections to aid program comprehension. We achieve this objective by showing a case study that illustrates using projections on examples. Presented case study was done using our prototypical tool that is implemented as a plugin for NetBeans IDE. We briefly introduce the tool and present an experiment that we have conducted with a group of students at our university. The results of the experiment indicate that projections have positive effect on program comprehension.

1998 ACM Subject Classification D.2.6 Programming Environments, D.2.11 Software Architectures

Keywords and phrases concern-oriented source code projections, program comprehension, projectional editing, code projections, programming environments

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.35

1 Introduction

Program comprehension is a process of retrieving information and knowledge about a software system by studying its source code. It is a process of recreating the mapping between the problem and the solution (implementation) domain that was created during the implementation phase of the software. Solutions in this area aim to reduce the amount of time needed for understanding the program source code. More radical solutions, like for example literate programming [12] or elucidative programming [14], were not adapted in the industry, probably because they were too distant from the industrial practice. In our previous work in this area presented in [11] we proposed a concept of *concern-oriented source code projections*. Concern-oriented source code projections overcome static structure of the source code by providing means to dynamically request a concrete view of the source code based on its concerns. In our method proposal specific concerns are associated with the source code by a metadata facility.

In this paper we continue in research presented in [11] and we discuss the process of creating projections. In addition we provide a case study that explains projections on a Minesweeper implementation in Java programming language. We have also implemented a tool prototype to experimentally evaluate the significance of concern-oriented source code projections for program comprehension.

© Jaroslav Porubän and Milan Nosáľ;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 35–50

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Concern-oriented Source Code Projections

Many times one source code is considered good by one programmer and bad by another one. In our work we recognize that the problem of multiple viewpoints to the source code quality is, to some degree, a consequence of *static structuring* of the source code. As *time* and programmer's *experience* changes, the programmer would choose different design, different code structure as best. The evaluation also depends on the *problem* that the programmer currently deals with. The problem of current approaches such as the object-oriented programming or aspect-oriented programming is that they allow the structure to meet some concrete needs, but adaptation of the structure to the new needs requires rebuilding the whole solution. Each new programmer has to work with the design that was previously chosen by someone else. They have to grasp a mental model of someone else even when the original design decisions are not relevant anymore. The comprehension of such a code is significantly impeded.

Our code projections are based on dynamic structuring of program's source code. By dynamic source code structuring we mean multiple different structures of the source code at the same time. In a specific situation the programmer would be able to choose the structure that he/she currently considers the most relevant. One structure is *base structure* (we can look at a base structure as a serialization structure of the system). Base structure is used to initially implement the system. Other concern-oriented source code structures are dependent on the base structure. A concern-oriented source code structure is a view of the source code that takes into account source code concerns (int this way it relates to an aspect from AOP). Concern-oriented source code projections can overlap; the same piece of code can belong to multiple projections.

These concern-oriented structures have to be properly presented to programmer; otherwise they would be useless for program comprehension. A code projection maps a set of base source code structures to a set of *view*s. The special *Identity* projection defines a view that is identical to base source code structure; therefore it has to fully describe the system. A single view consists of source code fragments that share a concern (or a set of concerns). We will call these fragments *view member*s. Relations between view members may be explicitly expressed in the view – e.g., a view can be graphical. A concrete code projection is specified by a sentence in a program query language (PQL). Practically any PQL can be used; however, it has to support querying some form of custom metadata. E.g., then a projection can query for code that implements logging, etc.

A programmer creates a *projection query* that specifies which concerns are relevant to his/her current situation. Projection queries can be shared and stored for later reuse, or modified if necessary. The concept of the code projections is outlined in Figure 1.

To provide code projections there has to be a tool that would be able to create a view while managing the source code in its base structure. In case of code projections we see as a best option utilization of the IDE thanks to its approach to handle language in its infrastructure (considering IDE is an integrated set of language tools). An IDE usually works with language on three levels – notation, model and view. The notation level is the language concrete syntax that is used to serialize a sentence of the language to the file system. The model is basically a form of abstract syntax tree that is obtained by parsing a sentence. The view is a presentation syntax of the language that is shown to the programmer. IDE components usually work upon the language model. The transition from the model to the view is done by the editor component of the IDE (there may be multiple different editors for one language). To provide code projections all that needs to be done is a substitution or

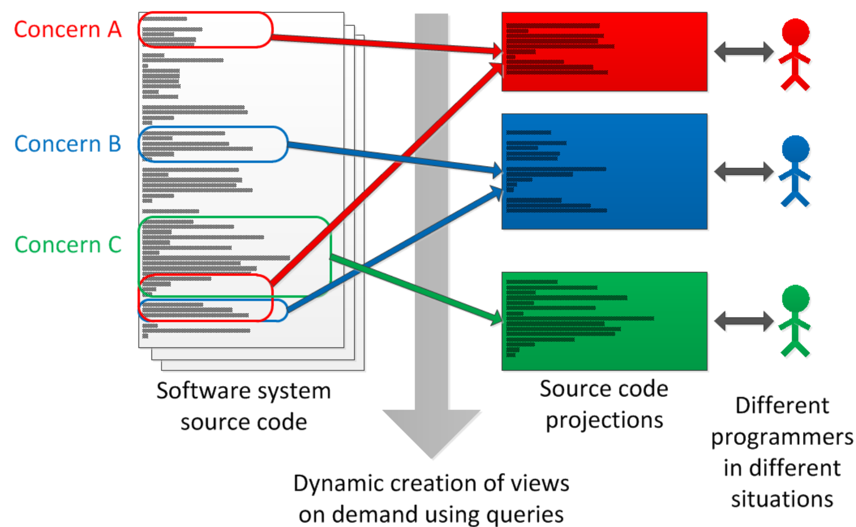

Figure 1 The concept of the concern-oriented source projections.

modification of an editor within the existing IDE. Since the editor works mainly with language model (abstract language representation), the concern-oriented source code projections are a specialized case of projectional editing discussed by Fowler in [3].

3 How to Create Projection Specification

As we have argued in previous section, our motivation for having different views of the software implementation is the variability of forces that affect the evaluation of the source code quality. In this section we will discuss the process of creating projections that provide programmers with those views.

Considering how a particular projection is created, we have following projection types:

- *annotation-based projections* that are based on explicit embedded metadata that are attached to source code just to create a projection,
- *configuration-based projections* that are based on explicit configuration or code conventions that are used to configure some framework or a tool, and
- *source intrinsic projections* that are based on the analysis of the rest of the source code.

A special case of projection is a projection composition that composes multiple different projections. Component projections may belong to different categories according to their creation. An example of a composite projection is a projection that shows all controllers (MVC pattern) that work with user profiles. First component of the projection is a projection showing all controllers and the second one a projection showing classes working with user profiles.

3.1 Need of Annotations

One of the biggest problems of program comprehension is the *semantic gap* between program representations on different abstraction levels. This covers also notoriously known semantic gap between model and implementation. However, the same can be observed on more fine-grained levels, such as in different source code representation. For example, let

us consider a software system implemented in Java. The implementation represented in Java source code will provide the reader with comments. Compilation, however, discards comments. Compilation will reduce explicit information about the program from its current representation. Moving from higher abstraction level to lower one decreases the amount of problem domain information.

The process of mapping the program representation from its current abstraction level to a higher one is the topic of program comprehension. Although there are attempts to aid comprehension with automatic program analysis (reverse engineering) these attempts are still not effective enough to significantly help the programmer. Usually the reverse engineering tools just provide a few source intrinsic projections of the program (e.g., visual projection showing the class diagram).

This is the reason why we believe that we have to record the mapping between the problem domain (model) and the solution domain (implementation) explicitly in every program representation. To record this knowledge in the source code any appropriate embedded metadata format can be used. Source code annotations, structured comments, code conventions or any other embedded format that will preserve the knowledge close to the source code (a brief comparison of metadata formats can be found in [9]). Explicit recording of the design decisions and high level semantic properties is a basis for annotation-based projections and moreover, it tightly couples the requirements and model to the implementation.

Annotation-based projections are the most expensive to create because they require the programmer to explicitly add declarative marks to the source code. These embedded metadata do not have semantics in a formal meaning, their presence in the source code does not change the behaviour of a program. However, they significantly narrow the semantic gap between the problem domain and the solution domain and this fact is a motivation for their creation. Even comments can be used to create annotation-based projections if they are structured. Using the method we proposed in [10] we can also use the concern-oriented annotations to generate documentation for the source code.

There are two scenarios for creating annotation-based projections. In the first scenario source code author records his *design decisions* and *program requirements* implemented by particular source code element seamlessly along with implementation. In the second scenario a new programmer is trying to comprehend existing source code and records his current understanding (in other words he is making notes).

3.2 Configuration-based Projections

Configuration-based projections use additional information in the source code that was not primarily intended for projections. For example, let us assume that a programmer uses Java Persistence API (JPA) compliant ORM mapping framework (such as Hibernate) for object persistence. To define mapping between the database and classes he/she has to provide an appropriate configuration. This could mean marking all the entity classes with the `@Entity` and `@Table` annotations and their fields with `@Column`, `@Id`, `@Basic`, etc., respectively. This configuration information is processable by a projection tool to provide views that are specific for a domain for which the framework was implemented (in case of JPA the data persistence domain).

This category of projections covers also code conventions. If we have classes to follow Java Beans convention we can easily identify all the methods that are used to access fields of classes. They all start with '`get`' prefix. In [9] we have included naming and type conventions as a metadata format. Here we argue that every convention can be considered a metadata

format and in consequence is a viable input for concern-oriented source code projections. Using conventions we can provide multiple projections for existing projects without requiring any additional effort from their authors.

Conventions have been and still are used as a configuration format in multiple frameworks and tools and their popularity grows with application of the *Convention over Configuration* (COC) design pattern. COC requires users of a framework to follow some conventional decisions (such as a particular naming convention, etc.) and for that they do not have to specify configuration for the framework that is usually complex. If the programmers do not want to spend a lot of time configuring the framework, they can choose to go with default settings and enjoy fast product delivery. Let us take a look on the Hibernate framework again. Since Hibernate uses COC design pattern all we have to do to have an entity class persisted is to mark it with the `@Entity` annotation. If we do not configure deviations from the default the framework conventionally maps class names to the identically named database tables and the fields to its columns, respectively. Therefore the configuration knowledge about the entity class is present in the source code (although less explicit). Since COC is usually applied with the *Principle of Least Astonishment*, the information "recorded" by conventions can be easily extracted from the source code. E.g., if we want to extract the names of persisted fields of an entity class, we can just take names of all the field of the class that are not marked with `@Exclude` annotation (`@Exclude` explicitly marks a field as not persisted).

This type of projections requires additional work from the programmers as well, however, this effort is motivated by the benefits gained from using external frameworks or tools[1]. However, the semantic gap between the problem and solution domains is wider. A configuration domain is usually an implementation domain (e.g., model-view-controller pattern) or a very specific problem domain (e.g., object-relational mapping domain).

3.3 Source Code Queries

The last type of projections are source intrinsic projections. In this case there is no need for any additional effort in form of conventions, annotations or comments from the programmer to enhance the source code. The bare implementation is projected using just queries in a PQL. An example of such a projections are projections implemented in current IDEs. Find usages projection shows all the usages of a particular program element (e.g., all method calls). Reverse engineering deals with this category of projections. E.g., a reverse engineering tool can build a class diagram of the implementation thus realizing a graphical projections of the system.

In this case we expect each line of code to directly effect the execution of the program. This is a direct benefit of writing the source code. However, the semantic gap is wider then in the other two projection types. A programmer transforms requirements in the problem domain to implementation in the solution domain (general purpose language). During this transformation the problem domain dissolves in the implementation.

3.4 Summary

These three projection types differ in effort and costs needed to prepare them. Figure 2 summarizes their nature. Source intrinsic projections can be done upon any implementation

[1] A programmer is not marking an entity class with `@Entity` to include it to a projection. He/she does it so the class would be processed by the JPA tool. The projection is just a by-product.

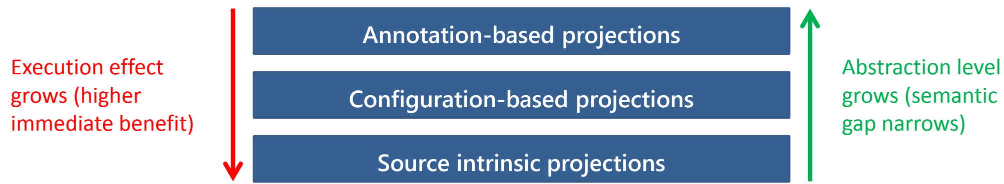

Figure 2 Different projection types.

but they lack proximity to the problem domain. Annotation-based projections require the code author to explicitly record his decisions and requirements that are implemented by the source code. This way annotation-based projections push the implementation closer to the problem domain. However, they require additional effort from the programmer. A compromise between the two are configuration-based projections that use conventions or configuration metadata. These metadata slightly narrow the semantic gap and their use is motivated enough even without projections.

4 Minesweeper Case Study

To illustrate concern-oriented source code projections on a meaningful example we will take a look at a Minesweeper game implemented in Java. The game is a simple reincarnation of the notoriously known Microsoft Minesweeper game that was distributed with the Windows OS family. Our Java Minesweeper has a text-based console user interface and has standard features such as monitoring the game state, opening tiles, marking tiles, etc.

In Figure 3 there is a simplified class diagram of the whole project. All the classes of the project represent the base structure of the implementation. However, Figure 3 is already a manually created projection – it hides the implementation of the methods and shows only the overall structure of the program.

4.1 Game State Semantic Concern

One of the core concepts of the game is the need to monitor the current state of the game and to behave according to that. The game starts with the state of generating the playing field. In this state the game is not playable. After the field is populated it gets the playing game state. In this state a player can play the game, he/she can open and mark tiles. Then we need at least two other states representing victory and game loss. These game states indicate that the game ended and that the player should not be able to interact with the game field anymore. A change of the state triggers some additional actions, such as recording a new high score or notifying the player that the game is over. In Figure 4 there is a state diagram of the game that represents a requirement for the implementation.

We had explicitly recorded the 'Game state management' concern of the source code in the implementation. To illustrate the 'Game state management'-oriented projection we have created a concrete view with our prototype implementation and it is shown in Figure 5. Its view members include the enumeration type for the game state and an attribute and methods of the `Field` class that are used to manage the current game state. Comments highlighted in green explain mapping the view back to the base structure of the program.

A skilled programmer might argue that in this case all we needed to do was to find all places where the `Field.state` is set. This is true. However, the knowledge that `Field.state` attribute is the right place where to start can be done only by code examination (unless we

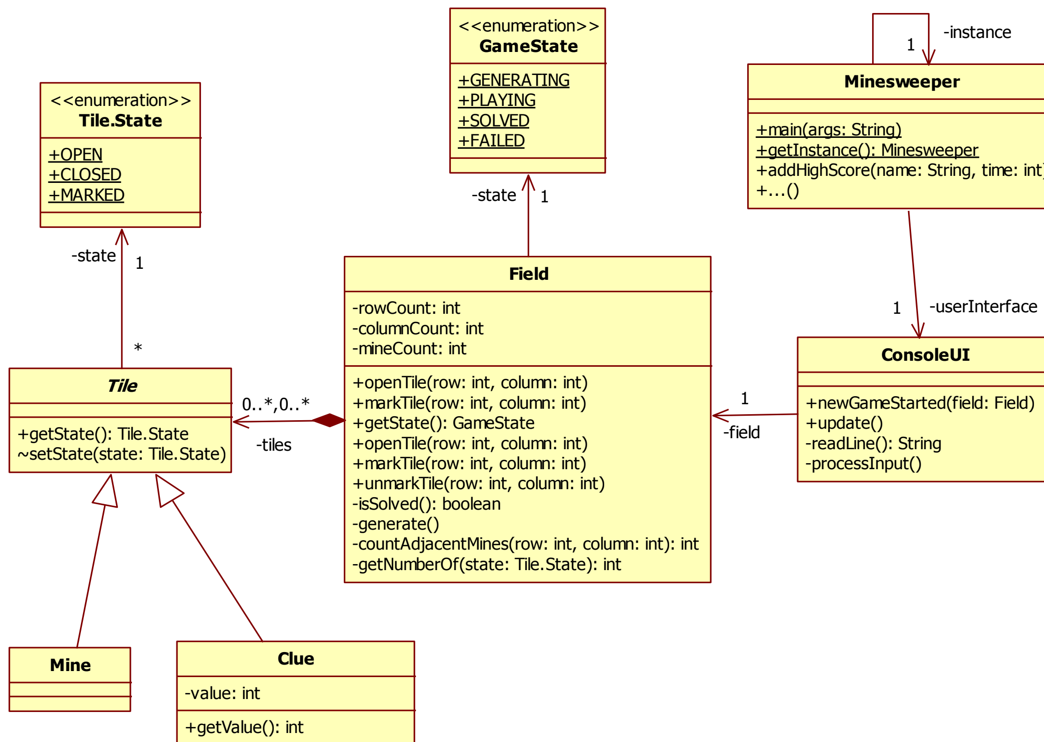

Figure 3 Simplified class diagram of the Minesweeper game.

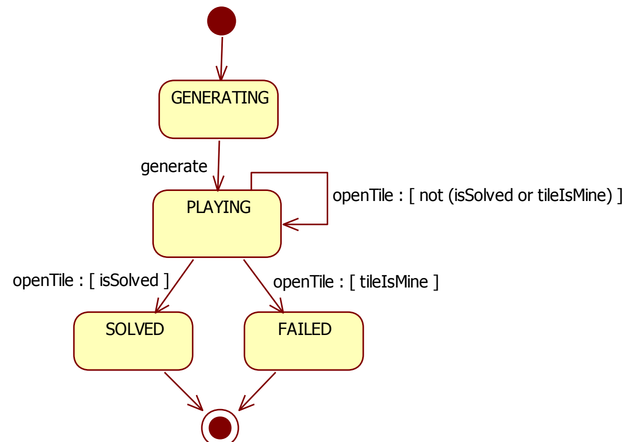

Figure 4 State diagram of the game state.

are authors of the code and we remember it). Here we used an explicit mark that annotates all program elements that manage the game state.

A source code projection based on this concern might be useful in multiple situations. It would simplify searching for a possible bug in a game state management, ease its comprehension for a new contributor to the project or simplify adding a new state to the game if that would be needed. Locating all the source code artefacts that would be affected by a state addition will be much faster with a projection based on this concern.

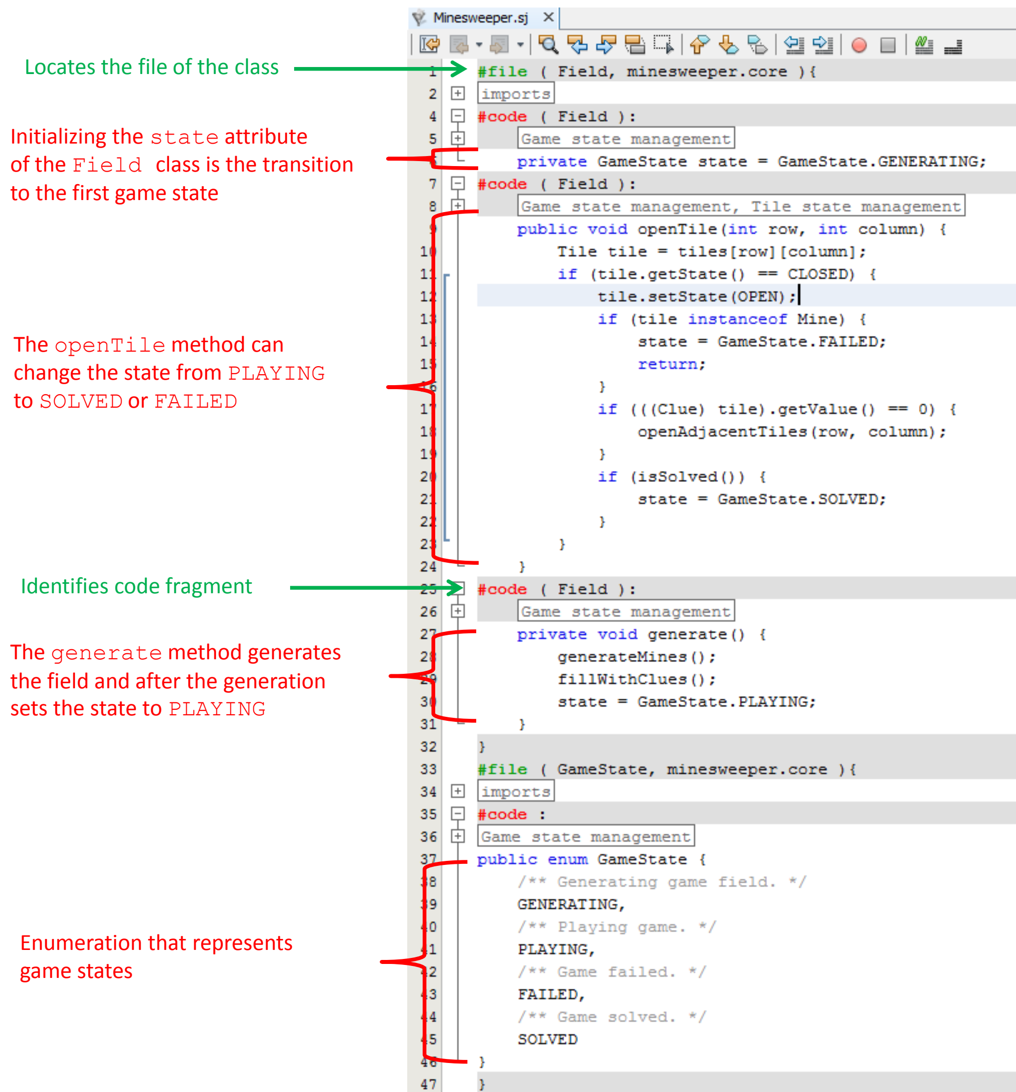

Figure 5 'Game state management'-oriented view of the Minesweeper game.

4.2 Singleton Design Concern

The same way we can use annotations to record also design decisions. As an example of a design decision for Minesweeper implementation we decided to implement `Minesweeper` class as a singleton. The `Minesweeper` class is a main class of the application. It manages game instances, it is responsible for the user interface choice (in future we want to support graphical UI), it will monitor game time and manage connection to database (to store high scores), etc. Although it did not have to be a singleton, it was our design decision and the class was implemented that way.

In Figure 6 there a 'Singleton design decision'-oriented view of the Minesweeper implementation. `Minesweeper` class is the only singleton in the implementation. For this projection we could also use just the naming convention of the `public static getInstance()` method. However, then if somebody would create a method with this signature that would not be a part of the singleton pattern the projection would be incorrect.

```
1  #file ( Minesweeper, minesweeper ){
2  imports
3  #code :
4  Singleton
5  public class Minesweeper {
6
7      private static Minesweeper instance;
8
9      public static Minesweeper getInstance() {
10         return instance;
11     }
12
13     private Minesweeper() {
14         instance = this;
15         newGame();
16     }
17
18     public final void newGame() {
19         Field field = new Field(8, 8, 10);
20         (new ConsoleUI()).newGameStarted(field);
21     }
22
23     public void addHighScore(String name, int time) { /* ... */ }
24
25     public static void main(String[] args) {
26         new Minesweeper();
27     }
28 }
29 }
30
```

Just a single singleton in the game implementation

Figure 6 Singleton projection of the Minesweeper game.

This projection could be useful for software architect that wants to be sure that all the singletons in the implementation are indeed implemented as singleton, or in a similar scenario.

5 Prototype Implementation

In this section we present a prototype of concern-oriented source code projections tool. The tool was named *Sieve Source Code Editor* (SSCE) and it was used in an experiment with the students at our university. The SSCE tool was designed and implemented as a plugin for the NetBeans IDE[2]. This tool works with the code written in the Java programming language.

5.1 Expressing Concerns

In our prototype tool the concerns and the design decisions are preserved using an embedded form of software system metadata – structured comments. For our prototype tool we have use special comments that starts with the `SsceIntent` keyword. The follows enumeration of concerns separated by semicolon that crosscut the commented program element. For example we can take a look at a source code in listing 1 with the `addHighScore` method that besides other things is also responsible for persisting the current high score to database. These comments map the implementation details (JDBC connection, statement, etc.) to the problem domain (high score implementation, high score persistence).

[2] `https://netbeans.org/`

Listing 1 'Database connection' concern recorded using SSCE structured comment.

```
//SsceIntent:High score; High score persistence;
public void addHighScore(String name, int time) {
  ...
  Connection connection =
              DriverManager.getConnection(URL, USER, PASSWORD);
  Statement stm = connection.createStatement();
  ...
}
```

5.2 SSCE User Interface

In the time of the experiment we supported only marker structured comments that do not have parameters. A programmer can mark program elements with tags expressing their design or semantic properties – concerns. Then when another programmer is interested in a particular concern the SSCE can be used to select ("sieve") the relevant code according to these markers. To provide such a view SSCE defines an editor that is designed to present the result of the projection. An example of a view by a code projection using Sieve editor was already presented in Figure 5 where the user selected the Game state management source code concern. Another example was presented in Figure 6 where the selected concern was the Singleton design pattern application.

The result of the SSCE concern-oriented projection is a view that contains relevant code fragments (view members), which are presented in a single stand-alone document. View members in this document are wrapped in the contextual frames `#file()` and `#code()` to make clear the connection of the presented fragments to the base structure of the source code and to express the context of the code fragments. In Figure 5 these frames are annotated with comments highlighted in green. The `#file()` frame encapsulates code fragments from the same base file. The `#file()` frame is divided to the `#code()` frames that specify a connection of the code fragment to its parent in program elements tree (e.g., to which class a method belongs, etc.). These frames are used to provide context of the code fragments and to allow consistent editing of the source code in concern-oriented views. This way any code can be unambiguously mapped back to the base code structure. The mapping frames are generated and protected by the editor (protection is done using guarded sections feature of NetBeans IDE).

The Sieve editor utilizes code folding, guarded sections and syntax highlighting techniques for better orientation and readability of the result of the projection. E.g., by default all code fragments in the view are collapsed to provide better initial overview.

Concerning code projections, the SSCE tool provides two more user interface components for the user. One interface is called *Intents manager* and it can be used to edit concern tags of the selected program element. The Intents manager is shown in Figure 7. A programmer can mark program elements directly through the editor using the SSCE structured comments, or he/she can use this Intents manager to pick existing intents (concerns) and the SSCE will mark currently selected program element for him/her. This interface was designed to provide easier manipulation of the explicit concerns in the source code.

The last interface component is the *Intents filter* that serves for creating simple concern-oriented queries. The Intents filter is shown in Figure 8. Using the Intents filter the programmer can pick a concern or concerns from the list of all concerns present in the source code. The selected concern or concerns define a projection that will be used by the

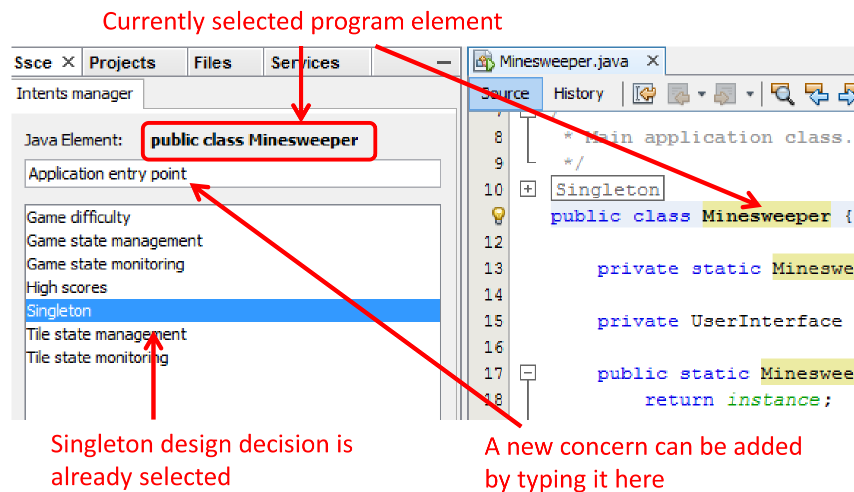

Figure 7 The SSCE Intents manager.

SSCE Sieve editor to create a view of the project's source code. The available concerns are presented in a simple multi-selection list. The set of concerns present in the source code is obtained by analysing the source code base.

So far to keep things simple we decided to use such a simple interface instead of a full-fledged PQL[3]. The Intents filter allow querying the code for code fragments that share some single concern, code fragments that all share a set of intents or code fragments that are a union of code fragments with at least one of the specified intent. This is done by selecting a single concern, selecting multiple concerns and choosing the `AND` composition mode or selecting multiple concerns and choosing the `OR` composition mode respectively. `AND` and `OR` composition modes allow simple composition of annotation-based projections.

6 Experiment

The idea of the concern-oriented source code projections was introduced to 40 respondents. These respondents were in the time of experiment bachelor's degree students in the 2nd year of study. 28 of them were working with Java for less than a year. The rest stated that they were working with Java for a year or two. Only one of the respondents had experience with Java for more than two years. Thanks to their relatively short experiences with the language and programming we could assume that they would not be biased against the new technique. Since we use the NetBeans IDE on our practical lessons, all of them were familiar with it.

6.1 Experiment Setup

Our main intention in this experiment was to verify our hypothesis about code projections and their contribution to program comprehension. We therefore designed two tasks, in which the respondents had to fix a bug and change some implementation in an unfamiliar source

[3] Our experiment was performed on a group of our bachelor's degree students in the 2nd year of study. Therefore we decided to use this simple mechanism instead of a full-fledged PQL, so they would not be overwhelmed by a complexity of a PQL and could rather focus on the contributions of the code projections method.

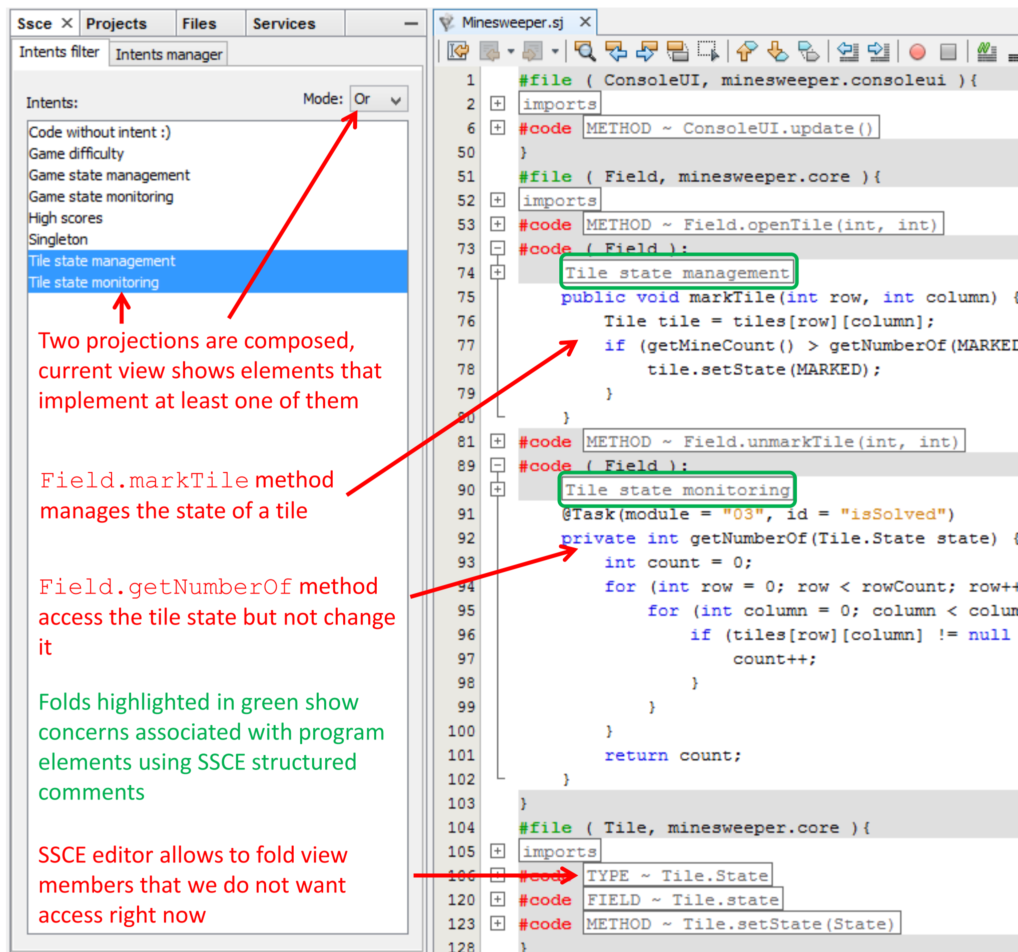

Figure 8 The SSCE Intents filter used to create simple composite projection.

code. As a source code sample we used the implementation of the SSCE plugin. The version of the plugin that was used in the experiment consisted of 33 classes with approximately 10 kLOC. Its character provided required complexity that could be compared to complexity of real world systems. The implementation was annotated with SSCE comments recording 28 different high level concerns (e.g., 'GUI update', 'Querying and concerns configuration', 'Annotations for recording concerns', 'Monitoring changes in Java classes', and more).

In their first task, they had to fix a bug with the `OR` composition mode of intents querying. In the implementation we have commented out few lines that took care of the `OR` mode. The main problem therefore did not have algorithmic character. It was rather a problem of program comprehension. Students had to browse the 10000 lines of code and search them for the implementation of composition modes. However, they could (and were encouraged to) use the plugin to prepare a view that would reduce the code needed to be searched. In this case it would mean selecting the 'Querying and concerns configuration' concern in the SSCE Intents filter. The SSCE editor would show them a single document with 830 LOC that consists of 1 whole class and 2 fields and 17 methods of 5 other classes. Instead of browsing all the classes the respondents could focus solely on the code that authors identified as relevant to querying the source code and concerns configuration. The view reduced the code to be searched more than 10 times.

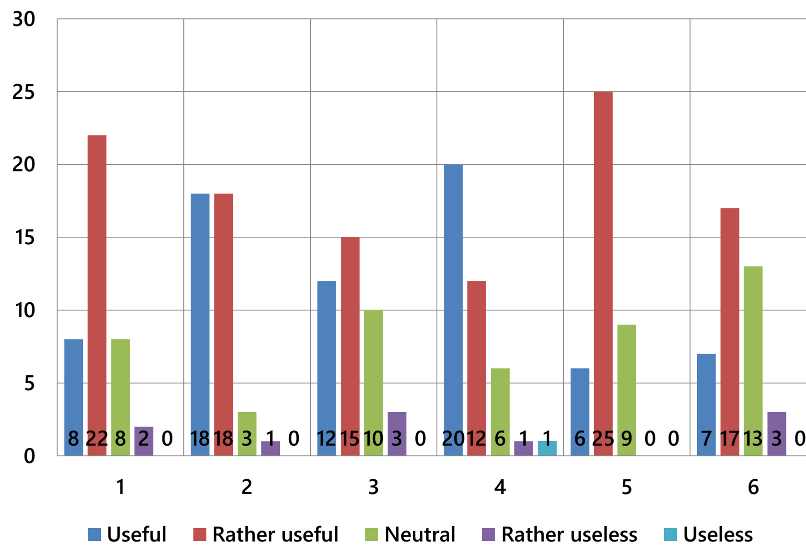

Figure 9 Usability rating of the code projections.

The second task was oriented more towards system evolution than to fixing bugs. The students had to change the structure of the SSCE structured comments. In this case it was again a matter of program comprehension, because they had to identify source code that implemented processing the SSCE structured comments. Here selecting the 'Annotations for recording concerns' concern for the view reduced the code to be searched to 500 LOC ($\approx$ 20x less than the whole code base).

6.2 Survey

After finishing these tasks the respondents were given a questionnaire about the method. They were asked to rate the significance of the code projections for the following activities:

1. Development of a new software system.
2. Maintenance of an existing system.
3. Program comprehension and understanding of the source code.
4. Orientation and navigation in the source code.
5. Testing and fixing bugs.
6. Documenting the source code.

All of these activities are related to program comprehension. The results are presented in Figure 9. In general, the code projections were rated positively, being considered useful or rather useful for all mentioned activities. Activities 2 and 4 were rated the best, indicating a positive impact on program comprehension. This experiment verified the importance of source code projections in the context of program comprehension. The results about activities 1 and 6 were more or less theoretical, because the respondents did not have to develop a whole new system and explicitly mark the source code with the SSCE structured comments. They could only experiment with tagging using the Intents manager. The problem of development and implementation of concern-enriched source code is a matter of our future research.

In summary the questionnaire results indicate that concern-oriented source code projections are beneficial for program comprehension. We believe that the relevance of the results

is even higher since the experiment was conducted with a very simple and feature-limited prototype of the tool.

7 Related Work

Our previous work presented in [11] provides motivation for source code projections and detailed description of our proposal. Section 2 provides brief introduction that covers the topic. Considering that with projections we aim to bring the formal source code representation closer to programmer's mental model, a related work to our is also the work of Kollár [5] who analyses the opposite direction in human-computer interaction. He discusses binding of informal semantics (human thinking) to semantic symbols (symbols processable by computer). His work is based on concept of language metalevels that can be used for slicing the abstraction gap between model and executable program (he uses the concept also in [6] for genetic evolution of programs).

Similar approach is used by Desmond et al. [2] in so called *Fluid source code views*. They allow viewing method bodies in place of their calls, thus reducing the need of browsing the source files. It is kind of similar to Go To Declaration projection of current IDEs, however using fluid source code views the body is shown directly in place of call using a tooltip. No explicit navigation across the source files is needed.

In modern Integrated Development Environments (IDE) there are already some built-in projections that use non-standard metadata, such as TODOs view that can show all lines of code marked with TODO comments. Or there is a projection that crosses out all the program elements marked with `@Deprecated` annotation, indicating that these program elements are deprecated and are not supposed to be used. In Jetbrains IntelliJ Idea there are multiple code projections that increase code comprehension. An example might be folding an anonymous class into lambda-like notation that is introduced by Java 8.

Intentional source code views [8] are sets of related program elements that share some intention. In this sense they are very similar to concern-oriented code projections. In Intentional views the intentions of the source code are specified using logic metaprogramming. Although they are close to our approach by providing means of defining architectural and conceptual information about source code, they differ in few rather important aspects. Intentional views require knowledge of logic metaprogramming. It is hard to expect every programmer to be a logic programmer. The Intentional View Browser is a new tool. In our code projections we want to utilize common programmer's natural environment – code projections are to be made integral part of a modern IDE. Intentional views use code conventions that tend to be fragile (see [4]).

Source code views are also proposed by Lommerse et al. [7]. However, their approach provides static view – merely according to standard source code properties (such as a Navigator view of the modern IDEs). They provide three views: the *syntactic view* showing syntactic constructs; the *symbol view* showing objects available after compilation; and the *evolution view* showing different version of source file. In this context it is worth to mention *Concern Graphs* [13], an abstraction used for better understandability of concerns in source code.

Callau et al. [1] introduce a concept of *ghost classes*. A ghost class is a class that is used but not yet defined. host classes exist so programmers can use them even if they have not defined them yet. Their usage raises no errors and IDE acts as if the classes existed. Callau et al. argue that ghost classes are useful for incremental programming where they represent flexible class prototypes.

Wada et al. [16] work in model driven development research. They use source code annotations to preserve links from the generated source code to the model used for generation. Their annotations therefore can be used to create our concern-oriented source code projections.

Besides Fowler another well known name in the field of projectional editing is Markus Voelter. His research is mostly centered around JetBrains MPS[4] language workbench. MPS utilizes projectional editing. Each language implemented in MPS defines also 'editors' for language concepts that define how are the concepts projected and edited in the MPS. One of Voelter's works is presented in [15] where he describes implementation of feature variability using MPS projectional language workbench. The MPS-like concept of projectional editing is focused on removing the parsing process from language editing. Instead of text-based editor that requires the editor to parse the input to provide any help MPS uses structural editor (structure-aware editor) that directly creates language AST. Their aim is to improve notation flexibility, error prevention, etc. In our work we understand projections rather as a mean to provide *multiple different views of the same system*.

8 Conclusion

In this paper we have discussed the basics of creation of our concern-oriented source code projections. The paper included a case study that shows usage of projections in a meaningful context and therefore explains motivation for using projections. We have also presented a prototypical tool that is an implementation of simple annotation-based source code projections. This tool was used in an experiment to get feedback on our concern-oriented source code projections from our students.

Contributions of this paper are as follows:

- A discussion of creation of concern-oriented projections with respect to costs.
- A case study explaining source code projections.
- Experimental evaluation by a survey confirming projections' positive effect on program comprehension.

Our plans for future work include improving the tool to work with Java annotations and to be able to support source intrinsic and configuration-based projections. From the methodical point of view we want to analyse how to prepare annotation-based projections so that they would cost as little as possible and still give programmers benefit. We perceive that the problem of motivation for explicit recording of concerns in the code is the main obstacle for using concern-oriented projections in practice.

Acknowledgements. Research presented in this paper is supported by VEGA Grant No. 1/0341/13 "Principles and Methods of Automated Abstraction of Computer Languages and Software Development Based on the Semantic Enrichment Caused by Communication".

References

1 Oscar Callau and Éric Tanter. Programming with ghosts. *IEEE Software*, 30(1):74–80, 2013.

2 Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid source code views. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC'06, pages 260–263, Washington, DC, USA, 2006. IEEE Computer Society.

[4] http://www.jetbrains.com/mps/

3 Martin Fowler. Projectional editing. Blog entry, available at `http://martinfowler.com/bliki/ProjectionalEditing.html`, January 2008.

4 Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 195–213, Berlin, Heidelberg, 2005. Springer-Verlag.

5 Ján Kollár. Formal processing of informal meaning by abstract interpretation. In *KES IDT 2014 – 6th International Conference on Intelligent Decision Technologies*, KES IDT 2014, Chania, Greece, June 2014. Accepted.

6 Ján Kollár and Emília Pietriková. Genetic evolution of programs. In *Proceedings of the Twelfth International Conference Informatics 2013*, pages 127–132, November 2013.

7 Gerard Lommerse, Freek Nossin, Lucian Voinea, and Alexandru Telea. The visual code navigator: An interactive toolset for source code investigation. In *Proceedings of the 2005 IEEE Symposium on Information Visualization*, INFOVIS'05, pages 4–, Washington, DC, USA, 2005. IEEE Computer Society.

8 Kim Mens, Bernard Poll, and Sebastián González. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance*, ICSM'03, pages 169–178, Washington, DC, USA, 2003. IEEE Computer Society.

9 Milan Nosáľ. Software system metadata alternatives to annotations. In *Proceedings of Poster 2013: 17th International Student Conference on Electrical Engineering*, pages 1–5, May 2013.

10 Milan Nosáľ and Jaroslav Porubän. Software documentation through source code annotations. In *Informatics 2013 : Proceedings of the Twelfth International Scientific Conference on Informatics*, Informatics'2013, pages 180–185, 2013.

11 Matej Nosáľ, Jaroslav Porubän, and Milan Nosáľ. Concern-oriented source code projections. In *Federated Conference on Computer Science and Information Systems (FedCSIS), 2013*, pages 1541–1544, Sept 2013.

12 James Dean Palmer and Eddie Hillenbrand. Reimagining literate programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA'09, pages 1007–1014, New York, NY, USA, 2009. ACM.

13 Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE'02, pages 406–416, New York, NY, USA, 2002. ACM.

14 Thomas Vestdam. Elucidative programming in open integrated development environments for Java. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, PPPJ'03, pages 49–54, New York, NY, USA, 2003. Computer Science Press, Inc.

15 Markus Voelter. Implementing feature variability for models and code with projectional language workbenches. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD'10, pages 41–48, New York, NY, USA, 2010. ACM.

16 Hiroshi Wada, Junichi Suzuki, and Katsuya Oba. Modeling Turnpike: A model-driven framework for domain-specific software development. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'05, pages 128–129. ACM, 2005.