Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1529–1532

Concern-oriented Source Code Projections

Matej Nosáľ, Jaroslav Porubän and Milan Nosáľ
Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Email: matej.nosal@gmail.com, {jaroslav.poruban,milan.nosal}@tuke.sk

***Abstract*—The quality of the source code structure is a matter of the point of view, one programmer might consider one structure the best, the other not. A concrete structure can help in certain situations with the program understanding. Therefore we propose using dynamic structuring that allows assigning multiple structures to one source code to aid program comprehension. Concern-oriented source code projections facilitate this dynamic structuring expressed by custom metadata and provide multiple views of the source code that reflect logical structures provided by the dynamic structuring. This way in a specific situation a programmer can get a structure (by a view) that meets his/her current needs the best.**

I. Introduction

PROGRAM comprehension is a process of retrieving information and knowledge about a software system by studying its source code. Software system maintenance and evolution consumes up to 80 percent of system's lifetime [8]. Program comprehension tries to reduce this time. However, a more radical solutions, like for example literate programming [7] or elucidative programming [9], were not adapted in the industry, probably because they were too distant from the industrial practice.

We recognized that good design and source code structure are properties of *the point of view*. We believe that one static structure that is prevalent nowadays is not sufficient. Concern-oriented source code projections (or shortly code projections) contribute to the field of program comprehension by providing means to simultaneously express and use multiple structures of the source code.

II. Motivation

OOP uses classes and techniques such as dynamic binding to increase modularity, but on the other hand it also tears the source code structure to smaller parts (to submit it to single responsibility principle). AOP goes even further; it tears the structure even more according to concerns. It may help with modularity, but since the code as a whole (and thus the sequence of instructions) is scattered in more files, it is harder to follow the program logic. Considering which approach is better depends on the qualities taken into account – it is a matter of the point of view.

The same way it is difficult to find the best design, the best structure in a given paradigm. The quality of a design is a matter of point of view too. Let us consider a simple explanatory example from the AOP. An OOP method from listing 1 does some work and afterwards logs the process to the standard output.

Listing 1. Simple `doSomething()` method that is logged to standard output

```
public void doSomething() {
    ...
    // logging to standard output
    System.out.println
       ("Something␣was␣done!");
}
```

The AOP divides the source code into concerns that are implemented by aspects. Here a programmer Jack will identify a concern of logging in the system and will create a new aspect that will implement the logging concern. However, a programmer Jill will identify a concern of printing to standard output (for example to ensure that she will be able to easily switch from standard output to some other interface). So she would want to put the same line of code to the standard output aspect. So in this example Jack is interested in the logging concerns, while Jill in printing to standard output concern. However, these two concerns do not have to cover the same source code. Printing to standard output does not necessarily be logging. In this scenario Jack would say that creating logging aspect is better structuring that creating printing to standard output aspect, while Jill would say exactly the opposite.

Currently the source code has to have exactly one design, one structure. The concerns cannot overlap. The classes cannot either. Multiple design decisions can have good reasoning, but none of them tackles all the problems – e.g., sometimes the OO structuring is more useful, other times AOP structuring is advantageous. Usually it depends on whether the target of interest is a feature (OOP) or a concern (AOP).

The problem is based on the observation that the "best" structuring of the source code depends on the point of view. There are three main factors that influence the current best structuring:

- *Person* – each person has his/her own experience and opinion. Therefore each person has his/her own point of view.
- *Time* – even one person changes his/her opinion in time and in consequence his/her point of view. Usually the early structuring of the program evolves over the time to the required one even when there is only one programmer that authors it.

978-1-4673-4471-5/$25.00

- *Character of the problem/solution* – of course the structuring also closely depends on the character of the problem or the solution. Not only the people involved in the software development do evolve in time, but also the character of the problem.

III. Concern-oriented Source Code Projections

In our work we recognize that the problem of multiple points of view is a consequence of *static structuring* of the source code. The problem of current approaches such as the AOP or OOP is that they allow the structure to meet some concrete needs, but does not support easy adaptation to new needs. Metaphorically speaking, using AOP instead of OOP is analogical to tearing down a house and building it again rotated in 90 degrees just to provide a view from side. Wouldn't it be more effective if the house would stay untouched and instead a programmer would walk around the corner? The same way as in the analogy, we don't want to change the building process, we just want to provide a programmer with a view that he/she wants.

A. Static vs. Dynamic Structuring

The problem is current technologies support merely one structure of the source code. This structure has to fully describe the system and does not allow any duplication of code. If current structure of the source code is not viable for current needs, the programmer has just bad luck. He/she has to work with the current structure, or refactor it and hope that the original structure won't be needed later. Or after refactoring the code[1] he/she has to hope that there will not be a need of the original aspect.

To deal with these problems we propose to use *dynamic structuring*. By dynamic source code structuring in this context we mean a case when a source code of one system has multiple different structures at the same time. In a particular situation the programmer would be able to choose the structure that currently the most relevant. This concept of dynamic structuring changes the role of refactoring. Instead of refactoring in sense of dropping the old structure and building a new one, with dynamic structuring a programmer is able to arbitrarily add a new structure to the source code or remove an existing one.

The usual representation of the structure is the source code itself. This will not suffice in case of dynamic structuring. It would be too difficult and cumbersome to write multiple versions of system source code and to keep them consistent through the time. Instead of this "physical" structures' representation dynamic structuring should use logical representation. Source code itself would be written (and stored) only once using some *base structure*. Adding new structures should be done by adding metainformation about their new relationships and properties.

[1]That is not an easy task even with current tool support for code refactoring. Tools support automation of merely trivial tasks such as changing a name of a variable, etc. However here we talk about structure on higher abstraction level, like the choice of using inheritance instead of composition.

The idea of dynamic structuring that allows having multiple different source code structures at the same time is the core of the concern-oriented source code projections method. We consider this idea the main contribution of our work.

B. Views

Code projections are based on dynamic structuring of program's source code. These structures have to be properly presented to programmer; otherwise they would be useless for program comprehension. Code projections map a set of base source code structures to a set of *views*. A view is an abstract structure of source code that is presentable to programmer. A view does not have to fully describe the system and multiple views can overlap (one code fragment can be a part of multiple views). The *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 are somehow related. We will call these fragments view members. Relations between view members may be explicitly expressed in the view – a view can be graphical.

A code projection is specified by a sentence in a program query language[2]. Practically any PQL can be used; however, it has to support querying custom metadata too.

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 their current user is not satisfied with their current state. The concept of the code projections is outlined in figure 1.

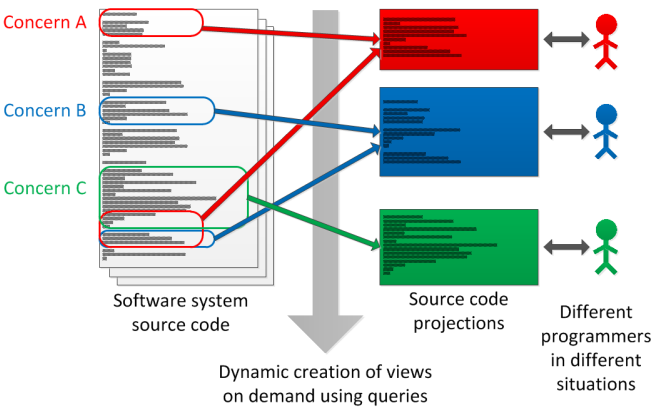

Fig. 1. The concept of the concern-oriented source projections

Our hypothesis in this work is that *current tools don't support flexible creating of views for viable price by using custom metadata*.

C. The Role of the Metadata

We will use term *software system metadata* (from now on only metadata) for the total sum (the set) of what one

[2]There are already programming query languages (PQL) in the world, this is not a new idea. However, PQLs are usually used to do source code checking (e.g., [4]). Our method uses PQL to provide a code projection.

can say about any program element, in a machine or human understandable representation.

Current IDEs provide programmers with code projections that operate on *intrinsic* metadata[3]. Navigator view uses the class intrinsic metadata to present the class members. Navigating to implementation through Ctrl and left mouse click uses the program element identifier. IDEs can use inheritance hierarchy to show implementations of interfaces or classes. Find Usages view uses also program element identifier to bind the implementation to its usages and provides a very useful projection of the source code.

In an example with Jack and Jill (listing 1) Jill would get desired result merely using intrinsic metadata. She can just query for any `System.out.println` call.

However, a situation would change if the logging was encapsulated in a `Logger` class (listing 2) that would provide a `PrintStream` that should be used for logging.

Listing 2. Logger implementation

```
public class Logger {
   private static PrintStream stream
      = System.out;

   public static getStream() {
      return stream;
   }
}
```

Listing 3 shows a modified `doSomething()` method with obscured printing to standard output. In this case it would be much more difficult to create a query that could find all the lines printing to standard output.

Listing 3. Obscured usage of standard output for logging

```
public void doSomething() {
   ...
   // logging to standard output
   Logger.getStream().println
      ("Something was done!");
}
```

Therefore we propose the utilization of the *custom* metadata to provide more information about program elements and to use that as a basis for different source code projections. Custom metadata are a tool that can be used to enrich the source code with metainformation about the semantic or design intents of the program elements at a higher abstraction level than the GPL is itself. In this way projections can use this metadata to present code to a programmer at a higher abstraction level too. A projection maps *concern-enriched* source code to a view. For listing 3 one could use for example simple marker annotations `@Log` and `@WritesToStandardOutput`.

D. The Role of the IDE

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 *Integrated Development Environment* (IDE) thanks to its approach to handle language in its infrastructure (considering IDE is an integrated set of language tools).

IDE infrastructure usually works with three language representations shown in figure 2, Notation, Model and View level. We want to utilize the editor to dynamically modify a view of the language.

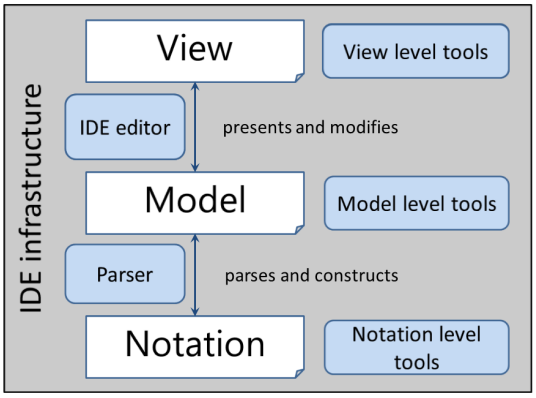

Fig. 2. Language representations in IDE infrastructure

If we will return to the analogy from the introduction to section III, concern-oriented source code projections provide a cheaper alternative to tearing down a house and building it in another angle. Instead of this invasive and inflexible solution code projections propose to change the view of an architect (analogy of programmer). Instead of moving the building merely the architect is moved to see what he/she needs to see.

IV. Related Work

We have applied a similar approach to *reduce the syntactic noise in internal domain-specific languages* in our previous work [6]. We were able to remove some undesired syntactic constructs (such as import section, class declaration, etc.) from the internal DSL based on Java language.

In modern IDEs there are many *standard projections* like the Navigator, TODOs, and others that we mentioned in section III-C. All these use projections to provide different views on the source code to provide a better orientation in the code.

Similar approach is used by Desmond et al. [1] 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.

Intentional source code views [5] are sets of related program elements that share some intention. In this sense they are

[3]Intrinsic metadata are standard metadata that define a program element. For example, for a class it is its name, its superclass, its interfaces, its methods and its attributes.

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. 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 [3]). And our projections can be used to edit the code, while Intentional views are read-only.

Source code annotations are used in [2], where Eisenberg et al. propose *simple edit-time metaobject protocol* that uses annotations as extensibility point of language. The editor is composed of multiple figures that have knowledge of what and how can be edited in them. These figures are configured by annotations.

V. Conclusion and Future Work

In this paper we argue that only one static structuring is not sufficient for software system source code. The quality of the source code structure is a matter of view, one programmer might consider one structure the best, the other not. A concrete structure can however play a significant role in program comprehension, since the point of view of looking at a code depends on the goal the programmer needs to achieve.

We presented the idea of so called dynamic structuring that allows assigning multiple structures to one source code. Our idea builds upon using logical structures that are expressed by metadata. When a new structuring is needed, it can be simply added to source code instead of overriding the existing one. Code projections utilize the view level of the language representation in the IDE infrastructure to reduce the price of their implementation and to still provide a modern professional IDE.

The *code projections* and *dynamic structuring* are the main contributions of the paper. Their purpose is to aid program comprehension. They can play significant role in documenting the source code, system maintenance and evolution.

Although we believe that dynamic structuring along with code projections may be a significant contribution to software engineering, we are aware of some problems with it. These problems we see as a space for our future work.

Dynamic structuring allows multiple, possibly uncountable, structures of the one source code. As we already argued, a good structure is a matter of the point of view. Therefore they may be just as many "good" structures as there are people working on the system. The *Babel effect*[4] is a consequence of the freedom in specifying the structure. Everybody has his/her own opinion and then it is hard to communicate. If two people are looking at two different views, it may be hard if not impossible to unambiguously talk about a specific source code fragment.

The second problem is the *price* of creating a projection. The source code annotations, or in general concern metadata only hardly can be created automatically. The source code has to be annotated explicitly either by it author or another programmer that recognize a need for a new structure. In this case a programmer has to consider the price annotating the source code to provide the space for code projections and the chance that the annotations will be reused later. If there is no real chance that the projection will be used, there is no good reason to create it. This is mainly a problem of development phase of software lifecycle, since the design decisions are made and the intended semantic properties are clearest in this phase.

[4]Named after famous story about building the Tower of Babel from the Bible.

Acknowledgment

This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the Artifacts Written in Domain-specific Languages Driven by Language Evolution.

References

[1] 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.

[2] Andrew D. Eisenberg and Gregor Kiczales. A simple edit-time metaobject protocol: controlling the display of metadata in programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 696–697, New York, NY, USA, 2006. ACM.

[3] 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.

[4] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, October 2005.

[5] 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–, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Milan Nosál', Jaroslav Porubän, and Matej Nosál'. Reducing syntactic noise in internal domain-specific languages. In *Proceedings of CSE 2012: International Scientific Conference on Computer Science and Engineering*, CSE 2012, pages 111–118, 2012.

[7] 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.

[8] Michal Vagač and Ján Kollár. Improving program comprehension by automatic metamodel abstraction. *Computer Science and Information Systems*, 9(1):235–247, 2012.

[9] 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.