Employing Query Technologies for Crosscutting Concern Comprehension

Marius Marin
Accenture
The Netherlands
Marius.Marin@accenture.com

Abstract

Common techniques for improving comprehensibility of software systems include concerns modularization and code structuring by applying well-established design patterns. However, non-trivial software systems will unavoidably exhibit concerns whose implementation cannot be mapped onto a single programming module, but will cut across the system's structure. These crosscutting concerns lead to implementations where multiple responsibilities are tangled inside the same module, making it hard to distinguish them and hampering the comprehensibility of the code.

This paper looks into how query technologies can be employed to improve comprehensibility of crosscutting concerns. To this end, we look at a tool-set aimed at supporting the identification and documentation of such concerns in source code. We discuss desired extensions to this tool-set and main challenges to query-based approaches for crosscutting concerns comprehension.

1. Introduction

Independently of whether they work on building new systems or, most often, on changing and extending existing ones, software developers need to gain a quick insight into complex software applications and into how they provide solutions to specific problems. This challenge is even more present in today's software development environments where maintenance and evolution is carried out for ever growing (legacy) applications and involves changing teams, in various different locations.

Program comprehension techniques are aimed at enabling efficient analysis of software systems for design recovery and creation of (mental) models of the system under investigation. An important subset of these techniques consists of query-based approaches, with code search tools integrated in modern development environments.

Query technologies have been used in various ways to aid program comprehension, from code metrics to design assessment and patterns detection, and to code style check and coding conventions enforcement. A relatively new area of applications for query technologies is the discovery and modeling of crosscutting concerns.

Crosscutting concerns are a major challenge to program comprehension and maintenance due to specific characteristics of their implementation, such as *code tangling*. In well-designed systems crosscutting concerns and tangling stem from modularization limitations of the employed programming paradigms, which prevent the assignment of a dedicated module to each concern in that system. As a result, a modularization unit in a program, such as a class or a method, must implement multiple responsibilities, i.e. concerns. For example, the module for a business object that requires secured access would implement authentication checks as a secondary concern, on top of the core domain functionality.

The problem of crosscutting concerns has received considerable attention from the research and practitioner community, most recently through the popularity of aspect-oriented software development (AOSD) techniques. However, AOSD techniques have yet to prove their value in improving code comprehensibility and maintainability in the long run. As an alternative, as well as an intermediate step towards AOSD adoption, the approach presented in this paper employs query technologies for improving comprehensibility of crosscutting concerns.

This paper looks at a tool-set for source code queries aimed at identifying, modeling and documenting crosscutting concerns. We conclude with a discussion on our experience with using these and other query tools and identify a number of challenges and desired enhancements in this application area for query technologies.

2. A Way from Tangled Code to Design

Today's most popular programming paradigms, such as object-oriented programming, allow us to define a single view on a software system by means of decomposing the system into modules, such as classes and methods. In this

view, each module maps onto a core concern, such as the *setName(SimpleName)* method in Figure 1 whose core concern is to store a new value for the *typeName* attribute.

However, the *setName* method follows a "3 step program" pattern, with the first part (*preReplaceChild*) implementing pre-condition checks and reports of various events and changes in the state of the method's object, and with the final part (*postReplaceChild*) reporting post-change events. This pattern is documented in the source code of the Eclipse Java Development Tools (JDT) project[1], and required to be implemented by all the classes representing elements of the Java programming language in the Eclipse abstract syntax tree (AST) framework. Such classes include, for instance, those for type declarations (*TypeDeclaration*). This counts for more than 100 methods implementing multiple concerns following the pattern above, all of which are part of the requirements for the complete functionality of these methods.

In the decomposition of our system, the *node change reports* end up as secondary concerns and, as a result, their implementation is tangled with that of other concerns and scattered over several modules. To understand the (relatively simple) *setName* method, we need to distinguish between each of its concerns by defining additional (mental) views for each of these concerns.

In this section, we look at an integrated set of tools that employ (relational) queries to define complementary views over a system to capture crosscutting concerns. In particular, these tools help to search for code smells of crosscutting concerns in source code and then to model and persistently document these concerns.

2.1. Advantages of Queries

Queries present several advantages for understanding crosscutting concerns and for gradual adoption of new programming techniques to implement these concerns. First, they allow one to navigate structures and relations that implement crosscutting functionality in isolation from other concerns.

[1] http://www.eclipse.org/jdt/

```
public void setName(SimpleName memberName) {
    if (memberName == null) {
        throw new IllegalArgumentException();
    }
    ASTNode oldChild = this.memberName;
    preReplaceChild(oldChild, memberName, NAME_PROPERTY);
    this.memberName = memberName;
    postReplaceChild(oldChild, memberName, NAME_PROPERTY);
}
```

Figure 1. Tangled concerns - method invocation idiom.

Second, queries help one to think in terms of typical (idiomatic) relations that have to be queried in order to capture crosscutting implementations. This latter aspect further allows one to understand what crosscutting concerns are encountered in practice or in specific application domains, and how they are typically implemented. This way, the use of queries can also serve educational purposes by promoting reasoning about code in terms of concerns and design (patterns).

From an adoption perspective, queries do not require modifications to the code base, a condition which is often seen as a safeguard against undesired changes.[2]

2.2. What and How to Query

A main challenge in searching for and representing crosscutting concerns stems from the broad range of examples of such concerns, from logging to exception handling and state change notification, to persistence, and to business rules and consistency enforcement. This challenge poses questions such as *what to query and what to find*, *how to query* and *how to turn identified code smells into consistent concern representations*.

Answering these questions requires to recognize common characteristics of crosscutting implementations, for example, by analyzing examples from literature, using one's experience, or looking into common approaches to addressing these concerns.

Such an analysis shows that many concerns, like (method level) security checking, logging, or events notification follow similar implementations idioms, namely the invocation of a specific action. This idiom is also present in our earlier example shown in Figure 1, for the concerns of event reports.

Aspect-oriented languages, like AspectJ[3], provide a dedicated mechanism for replacing crosscutting implementations based on the method call idiom, namely *pointcut and advice*.[4]

Our query-based approach distinguishes crosscutting concerns implemented using a method invocation idiom as a distinct class of concerns, i.e., the *Consistent Behavior* sort. Similarly, other idioms are described by different concern sorts [3]. For example, concerns like serialization, or the observer role implemented by a "view" element in a model-view(-controller) design are described by a *Role Superimposition* sort.

[2] Arguably, queries can be less efficient for enforcing behavior or policies, yet our case is not to use queries as an exclusive alternative to approaches like AOSD.

[3] http://www.eclipse.org/aspectj/

[4] The pointcut is an expression that indicates all the execution points in a program where the crosscutting action should execute, for instance, at the end of all public methods. The advice is a construct that is used to implement the action that needs to be executed at each point in the pointcut.

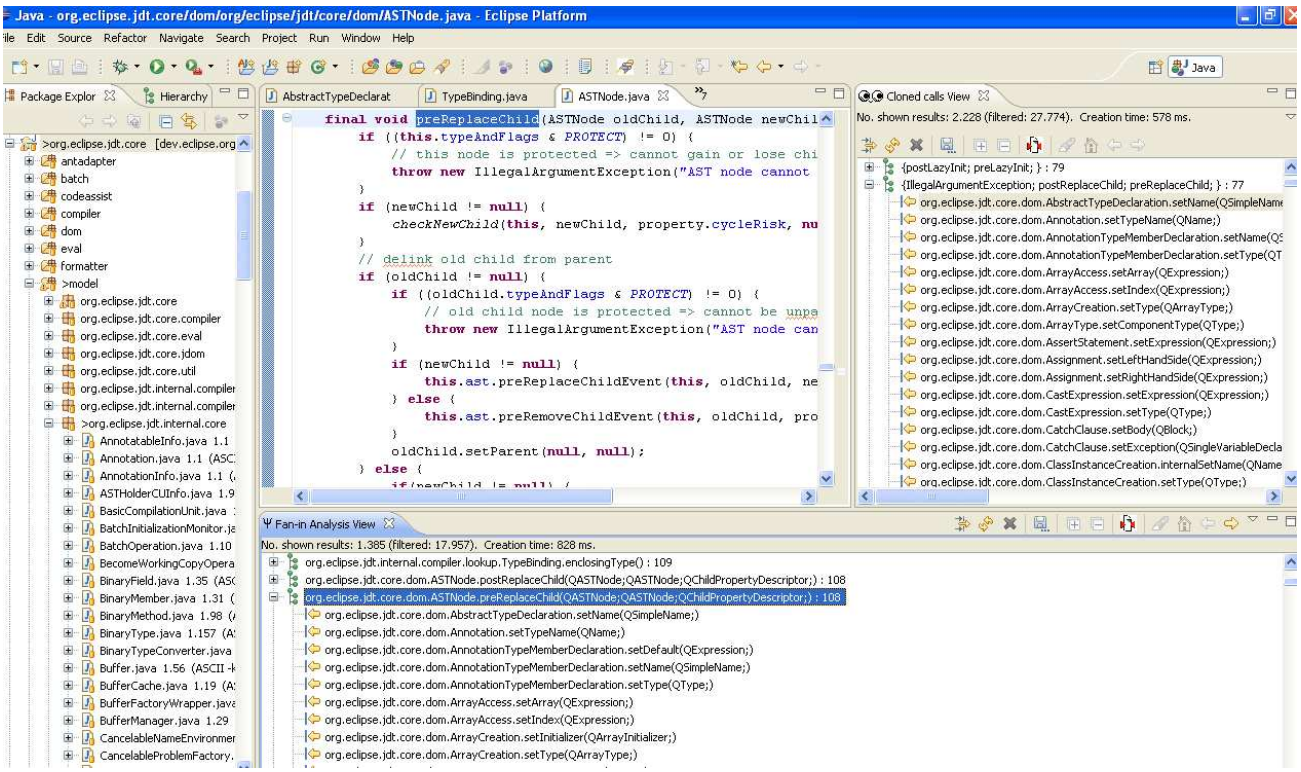

Figure 2. FINT**: Results for Fan-in Analysis and Grouped Calls Analysis.**

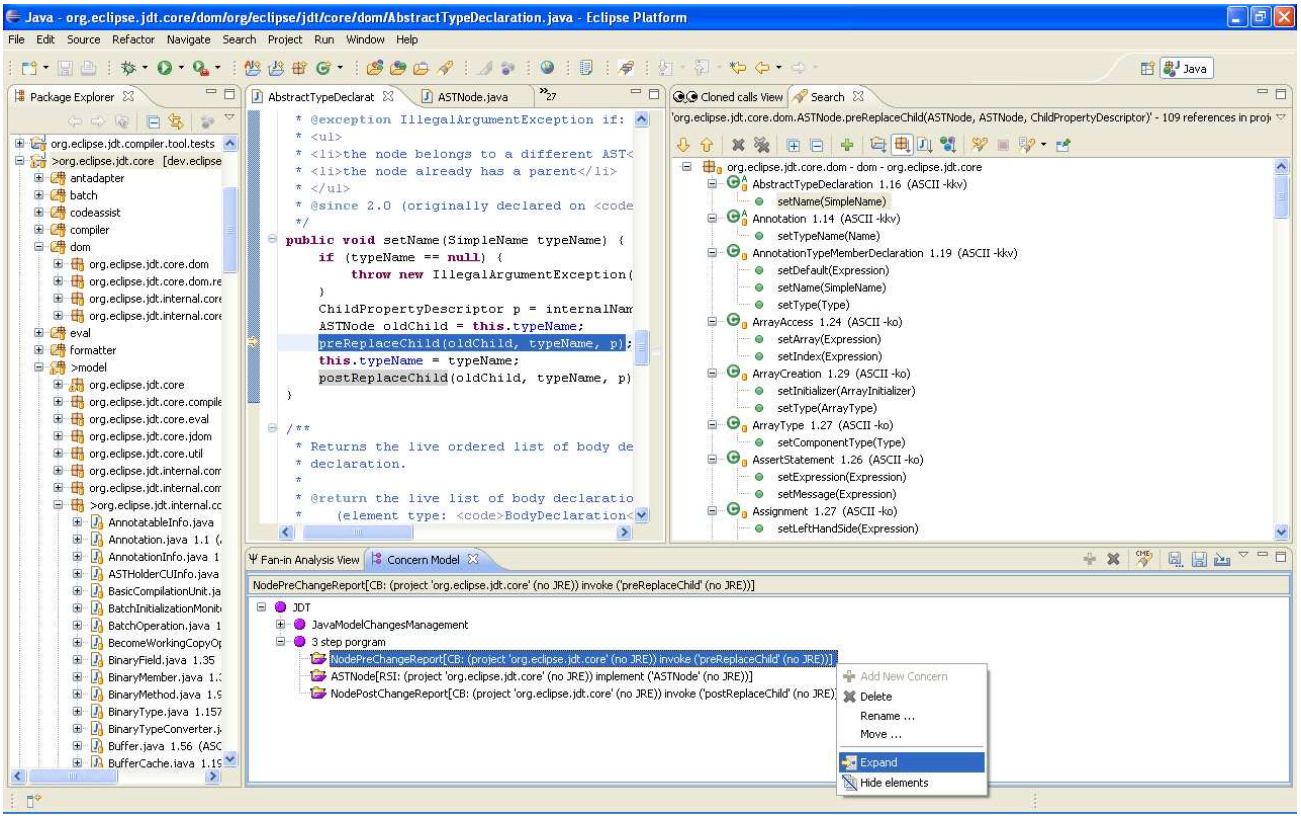

Figure 3. SOQUET **documentation of the concerns in the "3 step pattern" implementations.**

The classification of concerns in sorts based on specific implementation idioms allows us to define a predicate for the idiomatic relation associated to each concern sort. We use these predicates to search for specific idioms in source code, and further on to document (identified) crosscutting concerns.

2.3. Crosscutting Concern Identification

The first component of our tool-set is FINT [5], a tool aimed at identifying smells of crosscutting concerns in source code. The tool implements a number of techniques that analyze the code and search for the implementation idiom of a specific concern sort.

Figure 2 shows the results of two of the techniques in FINT that search for concerns of the *Consistent Behavior* sort. The Fan-in analysis, which is shown at the bottom of Figure 2, reports all the method call relations that satisfy a number of conditions that can be set via the tool [2]. These conditions include a check for a required minimum number of callers for a method to be selected as a result, or the condition that the calls to a certain method have the same position. The relations meeting the conditions are then displayed in a tree structure, with the callers as leaf-elements.

A second technique implemented in FINT is Grouped calls analysis, which is shown on the right side of Figure 2. This technique searches for maximal group of methods that share a minimum number of callers, preset by the user in the tool's configuration.

Both these techniques have been able to point us to the crosscutting concern illustrated in Figure 1, due to the large number of callers of each and both of the two event-report methods.

2.4. Crosscutting Concern Modeling and Documentation

The second component of our tool-set is a tool called SOQUET [6]. SOQUET provides a set of query templates for a number of concern sorts, as well as user-interfaces to configure these queries [4]. For instance, the user can run a simple query to return all calls to a specific method that originate from methods in a given package or project. This query can be saved in the *Concern model* view of the tool, shown at the bottom of Figure 3, and re-run at a later time by selecting the *Expand* option.

In our example in Figure 3, we have added a query for all the calls from the *org.eclipse.jdt.core* project to the *preReplaceChild* method discussed earlier. By running the query, we have the results displayed in the *Search* view, shown on the right side of Figure 3. We save the query by giving it a name (*NodePreChangeReport*) corresponding to the concern documented by the query, and by choosing a parent concern in the *Concern model* view.

[5] http://swerl.tudelft.nl/view/AMR/FINT

[6] http://swerl.tudelft.nl/view/AMR/SoQueT

3. Discussion Points

The design and implementation of the query-based approach presented in this paper required to answer a number of questions about how to search and document crosscutting relations in source code. Here we revisit some of these questions and propose several discussion points based on our experience with using queries for crosscutting concern comprehension.

What are the typical implementation idioms for those concerns that hamper comprehension and how can they be captured using queries? A first question we posed to ourselves was how to define the goal of our search, i.e., what to query for and how to document the results.

Crosscutting concerns are typically vaguely defined and the examples of such concerns vary significantly in their characteristics, such as the level of granularity, the implementation idiom followed by their implementation, or the software artifacts where they are primarily defined (e.g., architecture and design enforcement concerns, functional and non-functional requirements, code level contracts, etc.).

The list of idioms we identified and the classification of concerns into sorts address problems like consistent granularity and recognition of distinct properties to address crosscutting concerns [3]. Furthermore, the list of implementation idioms gives us a good insight into how developers think about crosscutting concerns and how they understand them.

However, we believe this set of concern sorts is only a first step towards understanding how crosscutting concerns are typically implemented, what are their underlying relations, and how current query languages can express these relations.

What relations and software artifacts to query? Modern frameworks use (XML) configuration files to define relations between source code elements, such as constructor invocations or resources lookup, and to hide crosscutting complexity from the user. These relations are not transparent to source code query tools like FINT and SOQUET, although they are just as relevant for program comprehension.

Several questions that raise from here regard the way we should query and present these relations to the user, as well as the way to integrate them with a query-based representation of concerns.

Powerful queries but simple interfaces? The tools we presented are based on hard-coded queries that can be configured using dedicated user interfaces. The integration with the Eclipse IDE and its code representation model ensures a

good level of performance for our queries, but this solution is also less flexible for extensions with new queries.

A solution to improve flexibility for tools like SOQUET consists of integration with specialized query engines, such as SemmleCode [1]. We argue that this integration is beneficial for code comprehension as well, as we discuss next.

Query languages allow the user to write powerful but complex queries. However, such queries are also difficult to understand and this complexity adds to a typically steep learning curve of the languages.

Query templates and well-designed libraries of queries make the complexity of the queries transparent to the user, and enforce discipline and consistency in documenting concerns. Furthermore, we can describe complex relations and designs, like an Observer pattern implementation, for example, by using collections of simple(r) queries that point to finer grained relations in the design.

Nevertheless, deciding about the right level of granularity for the queries to be provided in a library is a topic that requires more investigation.

Acknowledgements The author would like to thank Ben Tels for commenting on an earlier draft of this paper.

References

[1] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. Keynote address: .QL for source code analysis. In *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '07)*, pages 3–16, Washington, DC, USA, 2007. IEEE Computer Society.

[2] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.

[3] M. Marin, L. Moonen, and A. van Deursen. Documenting typical crosscutting concerns. In *Proceedings of the 14th IEEE Conference on Reverse Engineering (WCRE '07)*, pages 31–40, Washington, DC, USA, 2007. IEEE Computer Society.

[4] M. Marin, L. Moonen, and A. van Deursen. SOQUET: Query-based documentation of crosscutting concerns. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 758–761, Washington, DC, USA, 2007. IEEE Computer Society.