A Tool-Set to Query Source Code for Crosscutting Concerns

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

Abstract

Software systems are continuously evolved to enhance or correct functionality of earlier releases, to integrate new requirements, or to adopt new technologies. Many of the software systems in use today have been developed over multiple iterations and several years by teams that changed over time or handed over their work. As a result, such systems have become inherently complex, difficult to understand and costly to modify and evolve.

Common techniques for addressing complexity and for improving comprehensibility of software systems include modularization of concerns and structuring of the code by applying well-established design patterns. However, non-trivial software systems unavoidably exhibit concerns whose implementation cannot be mapped onto a single programming module, but 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 and the design.

This paper shows how source code query technologies can be used to improve comprehensibility of software systems, and particularly of crosscutting concerns. We illustrate general concepts of employing code queries for software analysis and comprehension through a tool-set for identification and documentation of crosscutting concerns in source code.

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 task is particularly challenging 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.

A major challenge to understanding, maintaining and ensuring the quality of software systems is the presence of *crosscutting concerns*. In well-designed systems, crosscutting implementation of concerns stems from modularization limitations of the employed programming paradigms which prevent the mapping of each concern in that system to a dedicated programming module. For example, the module for a business object that requires secured access implements authentication checks as a secondary concern, on top of the core domain functionality. Similarly, typical implementations of system-wide concerns like logging or error handling result in multiple responsibilities, i.e. concerns, *tangled* in one module, such as a class or a subroutine.

Tangling of concerns hampers program comprehension by making it difficult to distinguish between the different concerns implemented by one module. Furthermore, crosscutting concerns end up *scattered* over multiple modules harming the quality of the code [9].

The problem of crosscutting concerns has received significant attention from the software research and practice community, most recently through the popularity of aspect-oriented software development (AOSD) techniques [10, 20, 4]. The benefits of using aspect-oriented techniques for implementing typical crosscutting concerns like logging, security or transaction management have been discussed by various authors [23, 14, 6, 7]. However, a different line of argument goes that AOSD techniques require expert users and they have yet to prove their value in improving code comprehensibility and maintainability in the long run and for a wider range of concerns [26, 7].

In this paper we present an approach to the management of crosscutting concerns that is a complement to AOSD techniques, but that can also be applied independently. In particular, we focus on the identification and comprehension of crosscutting concerns by demonstrating a tool-set that employs query technologies to search for and to model typical crosscutting relations in source code.

The next section gives a background overview of such topics as crosscutting concerns and source code queries. In

Section 3 we look at design considerations for a consistent approach to the identification and modeling of crosscutting concerns. Section 4 describes the tool support of the proposed approach to concern management. We discuss a parallel between query-based approaches and AOSD, challenges to these approaches and interesting directions for further investigation in Section 5, and conclude in the last section.

2 Background: Crosscutting Concerns and Code Queries

This section gives a brief introduction into the topics of crosscutting concerns, concern identification and modeling, and source code queries.

2.1 Crosscutting Concerns

In order to manage complexity, today's most popular programming paradigms provide language mechanisms to decompose and organize a software system into modules. Consider, for example, a parsing application as the one implemented by the Eclipse IDE [1] to analyze Java source code and create Abstract Syntax Tree (AST) representations of the code. The application processes Java code such as the method declaration shown in Figure 1 into a representation as the one displayed in Figure 2

The core (concern) abstractions in the design of the parsing application include types for the AST nodes to represent Java syntactic constructs, such as method or local variable declarations, assignments, comments, etc. Typical implementations of these nodes allow for storing and accessing properties of the constructs they represent, like the start position and the length in the code of a declaration, or the name of a variable in a declaration.

However, all node elements in the AST framework are also required to implement a "3 step program" pattern

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

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

Figure 1. The *VariableDeclaration.setName* method implements a "3 step program" pattern.

Figure 2. AST View: The AST for the *setName(SimpleName)* method.

that is part of a design decision to ensure, for example, that all changes to the state of an AST node meet certain pre-conditions and trigger notification events to other elements. This pattern is implemented, for instance, by the *setName(SimpleName)* method, shown in Figure 1, that belongs to the AST node for *variable declarations*: first, the method invokes *preReplaceChild* to check method preconditions and report events and changes in the state of the AST node object; Then, the core functionality is executed, which consists of simply assigning and storing the name of the variable in the declaration; The last step consists of a call to *postReplaceChild* to report post-change events.

The "3-step program" illustrates the overlapping of responsibilities in one modularization unit, i.e., the tangling of concerns: the *setName* method implements not only the main concern of storing the variable name, but also crosscutting ones, such as notifications of change events.[2] This tangled implementation hinders the comprehensibility of the method as well as of each of its concerns.

Furthermore, the implementation of the pattern is scattered over more than 100 methods in the Eclipse Java Development Tools (JDT) project and is a requirement for any

[2] The crosscutting concern of model change notifications is common with well-known design solution such as the *MVC* or the *Observer* pattern [11].

new class in the AST framework that represents elements of the Java programming language. Therefore, changes to theimplementation of this concerns need to consider the full concern coverage and to be applied in many different places.

2.2 Identification of Crosscutting Implementation of Concerns

To extend a software system consistently with its design, one has to gain insight into the various concerns of that system and into their code *extent*, i.e., the set of program elements and relations that make up the concern. For example, the extension of the Eclipse's AST framework to support new Java language constructs requires awareness of the 3-step pattern described earlier and understanding of its program model. This understanding is particularly challenging for those concerns that are crosscutting, as they are not clearly distinguished in the structure of the code through dedicated modules.

The current common approaches to the identification of crosscutting concerns in the source code of existing systems (also known as *aspect mining*) employ various code analysis techniques to discover scattered and tangled implementation of concerns [17, 25, 13]. These technique aim at returning sets of elements that overlap in a relevant degree with the extent of crosscutting concerns in the analyzed code, which allows for recognizing these concerns.

2.3 Concern Modeling

Separation of concerns is a well-known technique for managing complexity in software systems [21]. Popular programming paradigms, such as object-oriented programming, provide language mechanisms to modularize concerns as means to achieve this separation. The decomposition realized by mapping a system's concerns onto dedicated modules defines the so-called dominant decomposition of that system [28].

Concern modeling techniques address the limitations in fitting all the concerns of a system into the dominant decomposition. These techniques provide mechanisms to group program elements in distinct "views" and navigate code relations in isolation from other concerns. Existing approaches include, for example, concern graphs [24] or the Concern Manipulation Environment [12].

2.4 Source Code Queries

Query and code search 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 [5, 2, 3, 1, 27]. Furthermore, many modern software development environments integrate code search tools, such as regular expressions matchers, to support code browsing and navigation.

A relatively new area of applications for query technologies is the discovery and modeling of crosscutting concerns [16]. These applications aim at developing software analysis and program comprehension techniques to enable design recovery, representation and navigation of crosscutting relations in source code.

In this paper, we look at queries techniques as means to extract relations of interest between program elements in a code base. More specifically, we consider a representation of a software system as a set of primitive relations (e.g., call, inheritance, declaration) between program elements – the end points of the relation –, and use source code queries to search for specific (composed) relations and/or end-points.

For example, Figure 3 shows a code query in the .QL language [8] for selecting all the method call relations and their end-points where (1) one of the end-points is the method specified by its name and the name of the declaring type, and (2) the other end-point is any caller-method declared in a given package.

3. A Way from Tangled Code to Design

In this section we set to discuss several design considerations for an approach to answering the question of *How to cope with crosscutting concerns in software source code?* Our approach aims at discovering such concerns in source code and at making their implementation explicit by using concern modeling techniques.

We focus our discussion on Java software systems, but most of the considerations presented here apply to other languages or programming paradigms as well.

3.1. How to Represent Crosscutting Concerns?

This question is important not only for the modeling of concerns but also for designing consistent approaches to their identification. A main challenge to answering the question consists of the broad range of examples of concerns that cover (non-)functional requirements, program features or design and implementation rules. These example include, for instance, business rules, persistence and logging, exception handling, events notifications or architecture rules enforcement.

To fit all these examples within a consistent approach we base our representation of crosscutting concerns on (cross-module) code relations. The representation of concerns such as logging or events notifications, for example, implies a call relation with the crosscutting (notification/log-

```
from Method caller, Method callee
where caller.calls(callee)
        and callee.getDeclaringType().getName() == "<type_name>" and callee.getName() == "<method_name>"
        and caller.getDeclaringType().getPackage().getName().matches("<package_name>")
select caller, callee
```

Figure 3. A (.QL) query for searching method call relations between specified end-points.

ging) method at one end-point and the crosscut elements at the other end-point of the relation.

Similarly, inheritance relations can be used to represent crosscutting concerns in classes that implement multiple roles; Such concerns are common, for example, with design patterns like *Observer* or *Visitor* [11] that introduce dedicated, crosscutting roles, such as *Subject*, *Observer* (*Listener*) or *Visitable* (*Element*). Each of these roles declares dedicated members that need to be implemented by the class implementing the specific roles on top of the their core concerns.

3.2. How to Identify Crosscutting Concerns?

The variety of examples of crosscutting concerns also poses challenges to defining the search goals for identifying such concerns in the source code of a software system. We need to ask what to search for and how to recognize if the search results point to a crosscutting concern.

Our proposed approach starts with an analysis of a large base of examples of concerns from literature, own experience, and significantly large software systems, such as JHotDraw, Sun's Java PetStore, Apache Tomcat, JBoss or Eclipse (JDT) [16, 17]. The outcome of this analysis consists of a number of typical implementation idioms and relations of crosscutting concerns [18], which we shall use as search goals for identifying crosscutting concerns.

The search for typical idioms, such as scattered method call relations, (- see, for example, concerns like logging or events notification -) can introduce false positives that need to be efficiently filtered. To this end, we shall use refining techniques to reduce the manual effort of recognizing concerns among the results of the search for concerns.

3.3. How to Integrate Mining Results and to Model Concerns?

In order to make crosscutting relations explicit, we look for a solution to consistently turn the results of the identification step into concern representations, and then to model and document these concerns. To this end, we base our modeling approach on the same representation strategy (namely, code relations) and set of implementation idioms employed at the identification step.

Our solution for modeling crosscutting relations uses code queries to specify these relations and document them in a persistent way: First, we extend our set of implementation idioms to a classification of crosscutting concerns, and associate each idiom to a distinct category (or class) of concerns that we call *concern sort*. Then, each concern sort is associated with a code query that specifies the code relation specific to the implementation of concerns of that sort. The query allows us to fit all concerns of the same kind/sort in one modeling solution.

One example of concern sort is *Consistent Behavior* that groups together concerns implemented using method call relations, such as the earlier examples of events or model-change notifications, logging, or JTA transaction delimitation[3]. The query for this sort specifies a method call relation between configurable end-points, such as, for instance, between a given logging method and any method declared in a given Java project.

For more examples of concern sorts and queries we refer to [18] and [16].

4. A Tool-Set for Managing Crosscutting Concerns

Our approach to managing crosscutting concerns is supported by a tool-set that consists of two main components:

- FINT is a tool that employs static analysis techniques to search for code smells of crosscutting implementation of concerns. The tool implements three techniques to search for relations specific to two concern sorts: *Consistent Behavior* and *Redirection Layer*.

- SOQUET is a tool for concern modeling and documentation. The tool provides a set of pre-defined queries that search for sort specific relations between two end-points. The definition of the end-points can be customized via dedicated user-interfaces.

Next, we discuss in more detail each of the two component tools.

[3] http://java.sun.com/javaee/technologies/jta

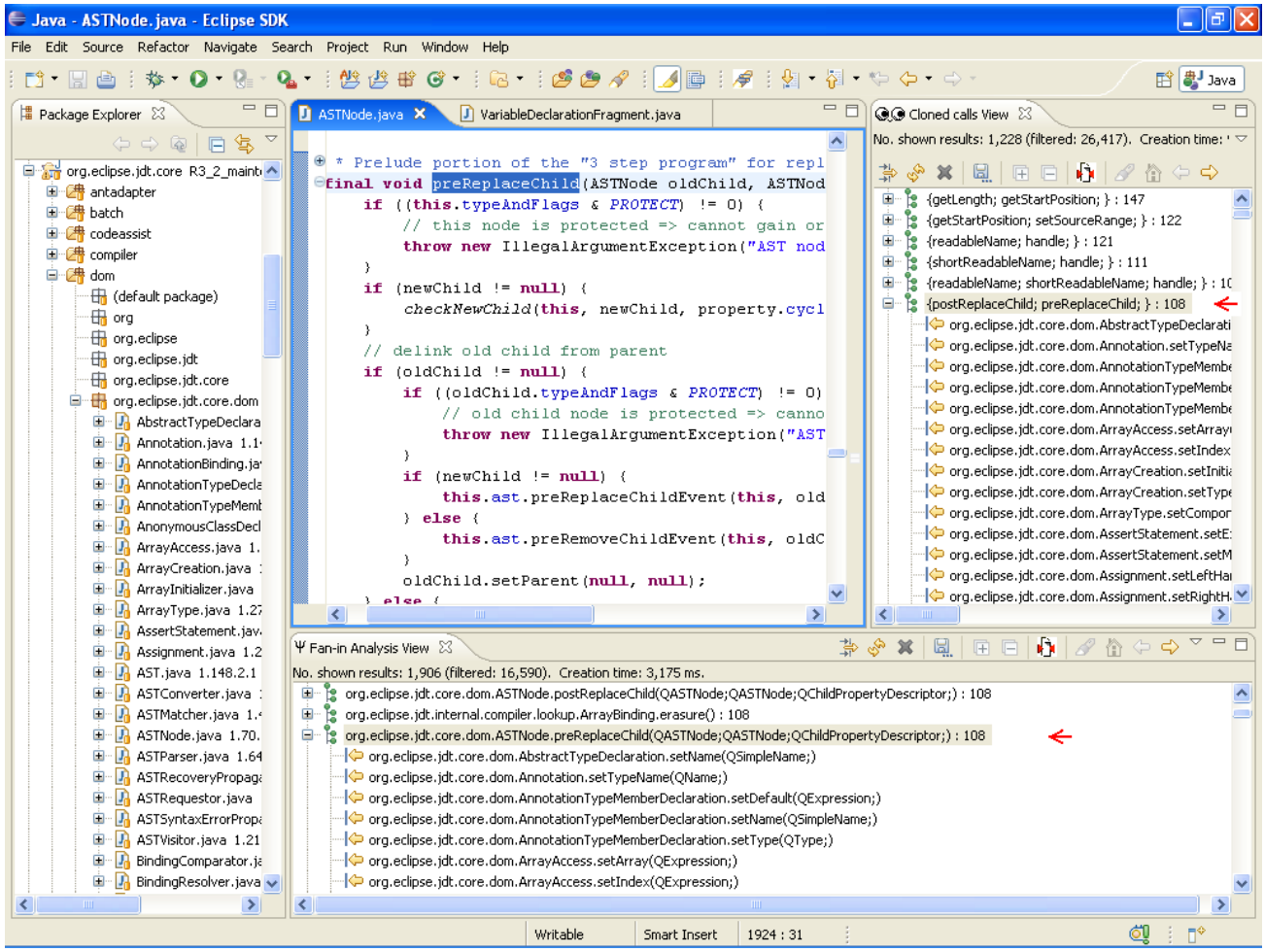

Figure 4. FINT**: Results of Fan-in Analysis and Grouped Calls Analysis in** FINT**.**

4.1. Crosscutting Concern Identification

FINT [4] implements a number of techniques to analyze the source code of the software system under investigation and search for the implementation idiom of specific concern sorts.

The analysis in FINT starts with the parsing of the source code in one traversal and the building of an in-memory code model that can be queried to select relations and calculate metrics of interest.

Figure 4 shows the results of two of the techniques in FINT that search for concerns of the *Consistent Behavior* sort:

Fan-in analysis is shown at the bottom of Figure 4. The technique allows the user to search for scattered method call relations by reporting all the call relations in the analyzed code and providing a number of options to filter the results. These filters allow one, for example, to set the minimum number of callers for the relations of interest, to ignore calls to getter/setter methods, or to analyze structural relations between the callers of a method, such as the position of the calls in the caller's body [17].

The relations meeting the conditions set via the filters are then displayed in a tree structure, with each method unfolding its callers, as illustrated in Figure 4.

The results of applying fan-in analysis to Eclipse JDT show the methods in the "3 step program" at the top of the results with a high number of callers. This is due to the high level of scattering (i.e., over 100 methods) and the consistent implementation of the concerns in this pattern. Similarly, the technique identifies typical crosscutting concerns like logging in JBoss or *valve pipeline* - chain of responsibility implementation in Tomcat [17, 16].

A second technique implemented in FINT is *Grouped calls analysis*, which is shown on the right side of Figure 4. The technique applies formal concept analysis [15] to search for maximal group of methods that share their callers. The tool's options include the setting of the minimum number of shared callers, as well as filters similar to *Fan-in analysis*.

The results of this technique also point to the "3 step program" by identifying the group of 3 methods in the pattern that are invoked by the same methods.

The third technique implemented by FINT is aimed at finding crosscutting relations specific, for example, to implementations of design patterns such as *Decorator*. The technique searches for classes whose methods have a one-to-one call relations with methods of another class in order to redirect the incoming calls, before of after executing specific actions. The predicate for finding redirections is shown in Figure 6 using the .QL language [8].

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

More details about the techniques in FINT, experimental results and user manuals are available in [16].

4.2. Crosscutting Concern Modeling and Documentation

The component to support concern modeling in our toolset is called SoQueT [5][19]. SoQueT provides a set of six predefined queries for concern-sort-specific relations, such as method invocations, class role implementation via inheritance and member declaration, declaration of dedicated method parameters to pass information along a call chain, or wrapper layers (e.g., Decorators [11]).

Each query can be customized via a dedicated interface that allows the user to restrict the end-points of the queried relation to a certain set of elements. For example, to model the *event notification* concern in the "3 step program", one selects the query for the *Consistent Behavior* sort and restricts the callee end-point of the relation to the *preReplaceChild(..)* (or *postReplaceChild(..)*) method, and the callers end-point to the set of all method in the *org.eclipse.jdt.core* project. The query returns all calls to the *preReplaceChild(..)* method in the *org.eclipse.jdt.core*, as shown in the *Search* view in Figure 5.

Further on, the query can be saved in the *Concern Model* view to persistently document the crosscutting concern of *events notification*. The query can be re-run at any later time by selecting the *Expand* option, as shown at the bottom of Figure 5.

The *Concern Model* view supports two types of user-defined elements: (1) Query elements, as the one for the *event notification* concern, are described by a name set by the user to document the concern (e.g., *NodePreChangeReport*), the concern's sort (e.g., CB – *Consistent Behavior*), and the queried relation; (2) The composite concern elements, such as *JDT* or *3 step program*, allow the user to

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

```
predicate redirect(Method m, Method n) {
  // m calls n
  m.calls(n)
  and
  // m calls no other method in the class class of n
  not exists(Method k |
             k.getDeclaringType() = n.getDeclaringType()
             and m.calls(k) and k!=n)
  and
  not exists(Method l |
             l.getDeclaringType() = m.getDeclaringType()
             and l.calls(n) and l!=m)
}
```

Figure 6. The predicate for methods executing redirections.

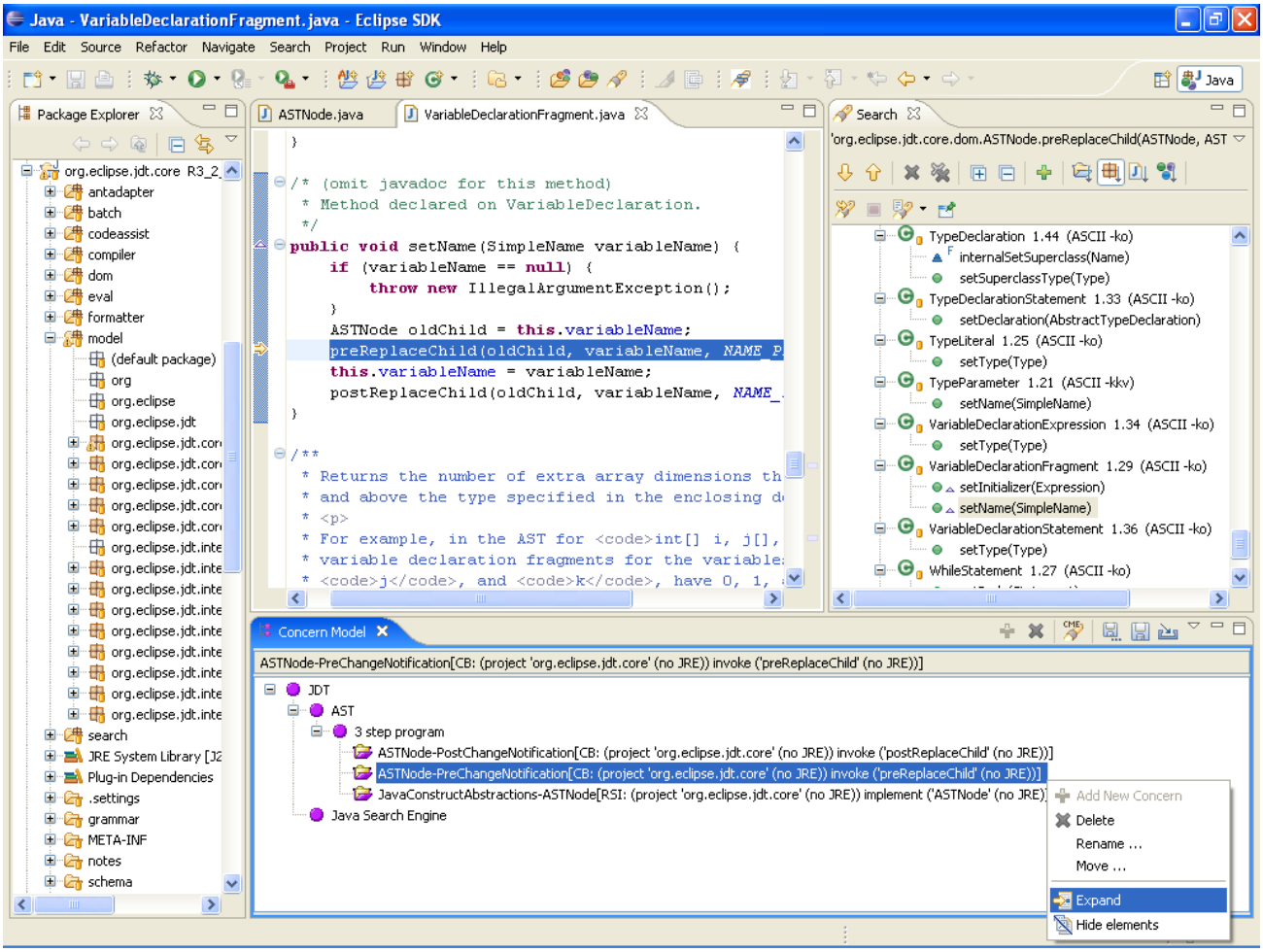

Figure 5. The modeling of concerns in SoQueT: the "3 step program" example.

group together, in a hierarchical structure, related query elements or other composite concerns. In our example, we group together all the query elements that document the "3 step program". All the concern documentations are child elements of the node representing the JDT project.

The user can save a concern model to an (XML) file and re-load into the tool for a later work session.

5. Discussion Points

Query technologies and AOSD Crosscutting concerns, just as software aging [22], are inherent challenges to comprehensibility and evolution of software systems. The approach discussed in this paper complements well-known software engineering practices in software design and implementation, as well as programming techniques for modularization of concerns, such as AOSD.

AOSD techniques have seen significant adoption recently, with implementation in languages like AspectJ or integration with popular frameworks, such as Spring or EJB 3.0. Query-based approaches complement the AOSD ones in a number of ways, such as enabling a better understanding of crosscutting concerns and their typical implementation.

Query technologies enable the finding of crosscutting concerns and the navigation of structures and relations making up these concerns in isolation from other concerns. This helps one to understand what concerns are encountered in practice or in specific application domains, and how they are typically implemented. The use of queries in these scenarios also serves educational purposes by promoting reasoning about code in terms of concerns and design (patterns).

Moreover, the queries showed in this paper describe typical implementations of crosscutting concerns, which allow for designing generic AOSD programming solutions, and hence for an easier adoption of AOSD programming techniques.

Another benefit of employing queries for managing concerns consists of not having to change the code base, a condition which is often seen as a safeguard against undesired changes. Arguably, queries can be less efficient than programming techniques such as AOSD ones, for enforcing behavior or policies; for example, a logging concern implemented in AspectJ automatically applies to all execution points in the program (i.e., *join-points*) that the programmer specifies, and also to any new program element that matches the selection criteria for join-points specified by the programmer (i.e., *pointcut*).

Another natural match between query technologies and AOSD is the use of queries to define more flexible pointcuts – that is, selection criteria for points of interest in a program –, which is a main challenge in providing effective aspect solutions to crosscutting concerns.

Challenges for query-based approaches and directions to investigate A first question to answer before setting to employ query technologies for concern identification and modeling is *What are the typical implementation idioms for those concerns that hamper comprehension and how can they be specified using queries?*

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 [18]. 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. Further steps consist of extending this set by, for example, looking at application domain specific concerns.

A second question to address is: *What relations and software artifacts to query for identifying and documenting crosscutting concerns?* 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 in the context of frameworks like Spring 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.

6. Conclusions

In this paper we presented an integrated approach and tool support to address the identification and modeling of crosscutting concerns in the source code of software systems. The approach employs query technologies to search for typical implementation of concerns and to document these concerns using a set of pre-defined queries for idiomatic crosscutting relations. We showed how the proposed approach aids to reverse engineer design and make crosscutting relations explicit for software evolution purposes.

References

[1] Bauhaus – tool suite for software architecture, software reengineering, and program understanding. http://www.bauhaus-stuttgart.de/bauhaus/index-english.html.

[2] PMD (and XPath). http://pmd.sourceforge.net/.

[3] SemmleCode. http://semmle.com/.

[4] The AspectJ Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide/.

[5] D. Beyer. Relational programming with CrocoPat. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 807–810, New York, NY, USA, 2006. ACM.

[6] A. Colyer, A. Clement, G. Harley, and M. Webster, editors. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools (Eclipse Series)*. Addison-Wesley, 2005.

[7] A. Colyer, R. Harrop, R. Johnson, A. Vasseur, D. Beuche, and C. Beust. Point/counterpoint. *IEEE Softw.*, 23(1):72–75, 2006.

[8] 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.

[9] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[10] R. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[12] W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Concern modeling in the concern manipulation environment. In *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software (MACS)*, pages 1–5, New York, NY, USA, 2005. ACM.

[13] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect Oriented Software Development*, 4640:143–162, 2007.

[14] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[15] C. Lindig. Fast concept analysis. In *Working with Conceptual Structures - Contributions to ICCS 2000*, pages 152–161. Shaker Verlag, August 2000.

[16] M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. PhD thesis, Delft University of Technology, 2008.

[17] 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.

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

[19] 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.

[20] G. C. Murphy and C. Schwanninger. Guest editors' introduction: Aspect-oriented programming. *IEEE Software*, 23(1):20–23, 2006.

[21] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[22] D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[23] R. Pawlak, J.-P. Retaillé, and L. Seinturier. *Foundations of AOP for J2EE Development (Foundation)*. Apress, Berkely, CA, USA, 2005.

[24] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3, 2007.

[25] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *Proceedings of the 20th International Conference on Automated software engineering (ASE)*, pages 184–193, New York, NY, USA, 2005. ACM.

[26] F. Steimann. The paradoxical success of aspect-oriented programming. pages 481–497, 2006.

[27] P. Tarr, W. Harrison, and H. Ossher. Pervasive query support in the Concern Manipulation Environment. Technical Report RC23343, IBM TJ Watson Research Center, Yorktown Heights, NY, 2004. http://sourceforge.net/projects/cme.

[28] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.