Delft University of Technology
Software Engineering Research Group
Technical Report Series

SoQueT: Query-Based Documentation of Crosscutting Concerns

Marius Marin, Leon Moonen and Arie van Deursen

Report TUD-SERG-2007-005

TUD-SERG-2007-005

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the International Conference on Software Engineering (ICSE), 2007

SoQueT: Query-Based Documentation of Crosscutting Concerns

Marius Marin
Delft University of Technology
The Netherlands
A.M.Marin@tudelft.nl

Leon Moonen
Delft University of Technology
The Netherlands
Leon.Moonen@computer.org

Arie van Deursen
Delft Univ. of Technology & CWI
The Netherlands
Arie.vanDeursen@tudelft.nl

Abstract

Understanding crosscutting concerns is difficult because their underlying relations remain hidden in a class-based decomposition of a system. Based on an extensive investigation of crosscutting concerns in existing systems and literature, we identified a number of typical implementation idioms and relations that allow us to group such concerns around so-called "sorts". In this paper, we present SoQueT*, a tool that uses sorts to support the consistent description and documentation of crosscutting relations using pre-defined, sort-specific query templates.*

1. Introduction

It is generally accepted that crosscutting concerns hinder program comprehension due to their tangled and scattered nature. Understanding these concerns requires one to observe relations spanning multiple modules and to identify relevant code elements in each of these modules. For example, a simple mechanism for notifying the listeners to specific events requires a number of methods in a given context to consistently invoke a given action after executing their core functionality. The comprehension challenge lies in the fact that all invocations of this notification method are part of a common task which itself is implicit in the code.

Having consistent documentation of the crosscutting concerns in a system can help developers to better understand and evolve the system by providing starting points for exploration and refactoring. In addition, such documentation supports aspect mining researchers with a much desired "golden standard" for validating their results.

We propose to use *queries* to document crosscutting relations in source code. Queries have been proposed by others to navigate and understand code [7, 3, 8]. However, these previous approaches do not particularly focus on crosscutting concerns, and hence neither on identifying common properties in such concerns, which would be needed for their consistent formalization and documentation using queries.

In our previous work we have identified and examined crosscutting concerns in more than 500,000 lines of Java code (including Tomcat, JBoss, JHotDraw, and Java PetStore) as well as in literature. Starting from this investigation we have developed a classification of these concerns based on a number of identified common properties. Such properties, which include typical implementation idioms and relations, allow to distinguish several classes of concerns that we call *sorts* [5]. Sorts are aimed at ensuring a consistent granularity level in describing and referring to crosscutting functionality. Moreover, we use queries to formalize the crosscutting concern sorts and their underlying relations, and, thus, document their concrete instances in source code.

This paper presents SoQueT, a tool for documenting crosscutting concerns in source code using sort-specific queries. SoQueT is typically used from two perspectives, namely, (1) as a tool for consistently *creating* crosscutting concern documentation for a system, and (2) as a tool for *exploring* query-based crosscutting concern documentation that was defined earlier for the system under investigation.

In the first scenario, the user is assumed to be acquainted with the concerns to be documented. An example is a developer that wants to make explicit some relations that are otherwise "hidden" by the object-based decomposition of a given system. The tool assists the developer with documenting these relations through a number of query templates defined for each concern sort. These templates can be instantiated using source entities from the target system.

In the second scenario, a user can explore a given system by loading the (pre-existing) sort-based documentation of an application into SoQueT in order to understand a number of crosscutting concerns and policies present in the implementation. In this case, the documented concerns can be browsed and their associated queries can be executed to find the corresponding source code.

2. Crosscutting Concern Sorts

A *crosscutting concern sort* is a class of concerns that share a number of properties: an *intent* (behavioral, design or policy requirements), a specific *implementation idiom* in an (object-oriented) language, and a (desired) *aspect-language mechanism* that supports the modularization of the sort's concrete instances. Examples of aspect mechanisms include *pointcut and advice* or *introduction*, as in AspectJ [1].

Table 1 gives a short description of the most commonly encountered sorts from the catalog of 12 sorts that we have compiled in previous work [5].

Query-based documentation of concerns As an example, consider the *Consistent Behavior* sort. Instances of this sort

Sort	Short description
(Method) Consistent Behavior	A set of method-elements consistently invoke a specific action as a step in their execution.
Contract Enforcement	A set of method-elements consistently check a common condition.
Redirection Layer	A type-element acts as a front-end interface having its methods responsible for receiving calls and redirecting them to dedicated methods of a specific reference, optionally executing additional functionality.
Expose Context (*Context Passing*)	Method-elements part of a call chain declare additional parameter(s) and pass it as argument to their callees for propagating context information along the chain.
Role Superimposition	Type-elements extend their core functionality through the implementation of a secondary role.
Support Classes for Role Superimposition	Type-elements implement secondary roles by enclosing nested, support classes. The nesting mechanism enforces and makes explicit the relationship between the role of the enclosing class and the one implemented by the support class.
Policy Enforcement	References between program elements are restricted as part of a (system) policy rule.
Exception Propagation (Declare *throws* clause)	Method-elements in a call chain consistently (re-)throw exceptions from their callees in the absence of an appropriate answer.

Table 1. A selection of Sorts of crosscuttingness.

include concerns like tracing the execution of all methods of a system, authentication and authorization calls from methods whose execution requires credentials checks, or consistent notification of observers of changes in the observed object. In all these cases, the implementation of the concern consists of scattered method calls to a specific action, implemented by a single method. Therefore, the relation specific to this sort is an *invoke* relation between the elements of a (defined) set of methods, the *source context*, and the set of the specifically invoked action, the *target context*. This relation can be formalized as follows:

$$\begin{aligned}&\text{CB}(\text{Element } s, \text{Method } m) := \{\, (m', m) \mid \\ &\quad m' \in \text{Method} \cap \text{Context}_{\text{CB}}(s) \;\wedge\; m' \text{ invokes } m \,\}\end{aligned}$$

To document a concrete concern of this sort, the user provides *seed*-elements to the query for the two contexts. The source context is a subset of all possible invokers of the method-action in the target context. This can correspond to the set of methods in a Java project, or the methods in a type hierarchy, or a class, etc. The helper function $\text{Context}_{\text{CB}}$ extracts all methods used from such a seed s.

We have used similar queries to formalize the relations for the other sorts as well. For instance, a *Role Superimposition* consists of an *implements* relation between a set of types and an explicitly (in case of an interface) or implicitly (type members) defined role. Such instances include implementation of design pattern roles, or concerns like Java's standard serialization mechanism.

A *Redirection Layer*, as another example, relates a set of methods in a class to methods of another class through an one-to-one *forwards* relation. Decorators and wrapper classes are typical examples of redirection layers.

Sorts queries such as the one above allow for the description of basic crosscutting relations. To express more complex relations that involve multiple sort instances, we organize these instances in composite *concern models*. Concern modeling has been used by tools like FEAT [7] or CME [2]. A concern model organizes concerns in a hierarchical structure. We integrate sorts into concern models by permitting queries as leaf elements in the concern hierarchy.

3. SoQueT Overview

To support sort-based concern modeling, we have built an Eclipse plugin called SoQueT (SOrts QUEry Tool).[1] This tool makes it possible to describe crosscutting relations and allows for persistent, sort-based documentation of concerns in existing code. SoQueT enables an engineer to query source code using query templates pre-defined for each crosscutting concern sort, as illustrated in Figure 1. In addition, the tool allows an engineer to organize sort instances into composite concern models to document complex crosscutting design decisions and features.

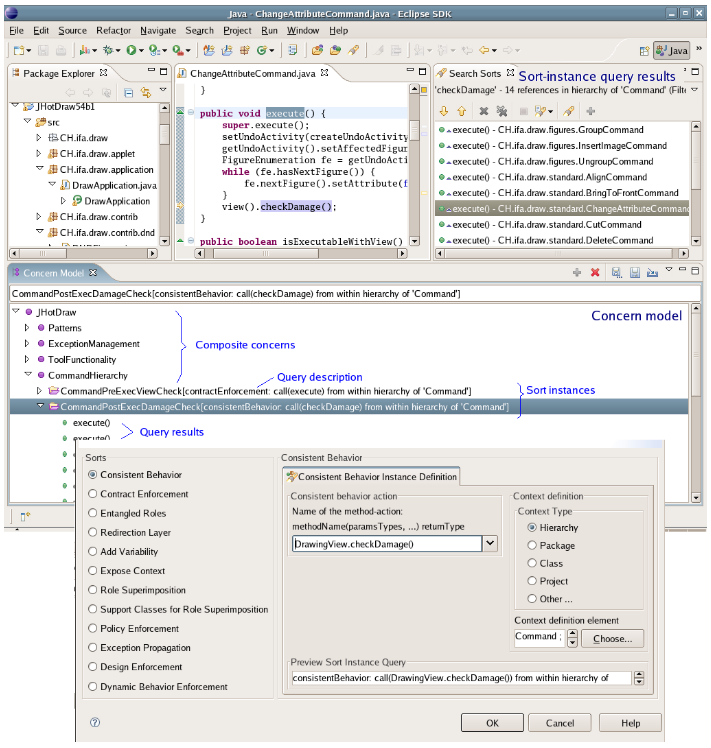
SoQueT provides three main user interface components: The interface to define a query for a specific sort based on its template is shown at the bottom of Figure 1. The template guides the user in querying for elements that pertain to a concrete sort-instance. The user just needs to introduce seed elements for defining the queried relation's contexts like, for instance, the top-level interface of a hierarchy context-type.

The results of the query are displayed in the *Search Sorts* view, shown at the top right in Figure 1. The view provides a number of options for navigating and investigating the results, such as display and organization layouts, sorting and filtering options, source code inspection, etc.

The *Concern Model* view allows to organize sort instances in composite concerns described by their user-defined names. The concern model is a connected graph, defining a view over the system that is complementary to Eclipse's standard package explorer. The graph is displayed as a tree hierarchy, with sort instances as leaf elements. A sort instance element can be expanded to re-run its associated query and display the results. A node representing a sort instance is labeled with a user defined name and the description of the associated query. Queries are associated only with sort instances and not with a composite concern.

To define and describe a role whose definition is tangled within another type, the tool introduces the concept of *virtual interfaces*. This mechanism allows the user to create a virtual

[1] Available at `http://swerl.tudelft.nl/view/AMR/SoQueT`


Java - ChangeAttributeCommand.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
ChangeAttributeCommand.java
Search Sorts
Sort-instance query results
Concern Model
Concern model
Composite concerns
Query description
Sort instances
Query results
OK
Cancel
Help

JHotDraw54b1
src
CH.ifa.draw
CH.ifa.draw.applet
CH.ifa.draw.application
DrawApplication.java
DrawApplication
CH.ifa.draw.contrib
CH.ifa.draw.contrib.dnd

```
    }

    public void execute() {
        super.execute();
        setUndoActivity(createUndoActivity
        getUndoActivity().setAffectedFigur
        FigureEnumeration fe = getUndoActi
        while (fe.hasNextFigure()) {
            fe.nextFigure().setAttribute(f
        }
        view().checkDamage();
    }

    public boolean isExecutableWithView()
```

'checkDamage' - 14 references in hierarchy of 'Command' (Filt

execute() - CH.ifa.draw.figures.GroupCommand
execute() - CH.ifa.draw.figures.InsertImageCommand
execute() - CH.ifa.draw.figures.UngroupCommand
execute() - CH.ifa.draw.standard.AlignCommand
execute() - CH.ifa.draw.standard.BringToFrontCommand
execute() - CH.ifa.draw.standard.ChangeAttributeCommand
execute() - CH.ifa.draw.standard.CutCommand
execute() - CH.ifa.draw.standard.DeleteCommand

CommandPostExecDamageCheck[consistentBehavior: call(checkDamage) from within hierarchy of 'Command']

JHotDraw
Patterns
ExceptionManagement
ToolFunctionality
CommandHierarchy
CommandPreExecViewCheck[contractEnforcement: call(execute) from within hierarchy of 'Command']
CommandPostExecDamageCheck[consistentBehavior: call(checkDamage) from within hierarchy of 'Command']
execute()
execute()

Sorts
[✓ Consistent Behavior]
Contract Enforcement
Entangled Roles
Redirection Layer
Add Variability
Expose Context
Role Superimposition
Support Classes for Role Superimposition
Policy Enforcement
Exception Propagation
Design Enforcement
Dynamic Behavior Enforcement

Consistent Behavior
Consistent Behavior Instance Definition
Consistent behavior action
Name of the method-action:
methodName(paramsTypes, ...) returnType
DrawingView.checkDamage()

Context definition
Context Type
[✓ Hierarchy] Package Class Project Other ...
Context definition element
Command ;
Choose...

Preview Sort Instance Query
consistentBehavior: call(DrawingView.checkDamage()) from within hierarchy of

Figure 1. SoQueT views and dialogs

type by selecting in a graphical interface those members of the multi-role type, such as methods or fields, that are part of the role of interest.

Documentation of Command support The concern model depicted in Figure 1 shows several concerns in JHOTDRAW's use of the *Command* design pattern. JHOTDRAW is a drawing application that was developed as an open-source show-case for good use of design patterns.[1]

Commands are responsible for executing actions triggered by, for example, menu items selected in the application's user interface. The (core of the) command and undo functionality is modeled in two dedicated (but interacting) hierarchies. The following concern model shows a (partial) documentation in SOQUET of the concerns that cut across these hierarchies:

ConcernModel(CommandHierarchy) = {
CE(Command, AbstractCommand.execute()) [PreExecutionCheck] ∪
CB(Command, DrawingView.checkDamage()) [PostExecNotification] ∪
CB(Command, setUndoActivity())[UndoSetup] ∪
SC(Command, Undoable)[CmdUndoSupport] ∪
RL(UndoRedoActivity, myReversedActivity) [ReverseActivity] ∪
...}

Each concern instance is represented by a two-letter function with the source and target contexts as arguments, as well as a symbolic name in square brackets explaining the intent of this concern instance.

The first concern is an instance of the *Contract Enforcement* (CE) sort; Before executing, commands check a reference to the drawing editor's active view by invoking the `execute` method of the default Command implementation (*AbstractCommand*).

The execution of commands is concluded with notifying the views of changes that occurred due to the command's execution. This notification is an instance of the *Consistent Behavior* (CB) sort, where the action to be executed consistently is implemented by the `checkDamage` method. Similarly, the commands create and store a reference to an associated Undo-support object. The queries that document these *Consistent Behavior* concerns are parameterized with the action that is (and needs to be) consistently executed. This action identifies the target context. The source context seed consists of the *Command* type, and is defined as the hierarchy of this type. The instantiation of the sort-query for the second concern is also shown in Figure 1.

The fourth element (SC) in the CommandHierarchy model documents an instance of a different sort: *Support Classes* are used to make explicit the association between Commands and their support Undo activities. The latter ones are implemented by *Undoable* classes nested in the concrete *Command* implementations. The query returns all the classes

[1] jhotdraw.org

of type *Undoable* nested within *Command* classes.

The last concern in the model belongs to the *Redirection Layer* sort. The *UndoRedoActivity* reverses an undo/redo action. To this end, it stores a reference to the reversed activity and has its methods consistently forwarding calls to that reference. The query receives as inputs the redirector type and the reference used for redirection, and reports those methods that consistently redirect calls.

4. Evaluation

We have documented well over a hundred crosscutting concerns using SoQueT, in significantly complex Java systems, such as JHotDraw and J2EE PetStore.[2] The documentation also covers a large number of sort instances present in design patterns solutions and complex features, like Undo support or transaction management. The experiments show that common crosscutting relations can be described by sorts, and documented using sort queries.

A concern model for JHotDraw comprising 103 sort instances is made available on the SoQueT web-site. More details about the documented concerns are provided in a technical report [4].

Documentation of crosscutting concerns with SoQueT helps in establishing benchmarks for aspect mining and refactoring. JHotDraw is evolving into such a benchmark for aspect mining that is currently being used by several research groups. The documentation of the crosscutting concerns by sorts and, hence, specific idioms allows for the comparison and combination of mining techniques [6].

5. Related Work

A number of tools allow to query and explore source code for concern understanding. Like SoQueT, most of these are implemented as Eclipse plug-ins.

FEAT uses a graph-based representation of concerns that originates in a seed, root (class) element [7]. The tool incorporates browsing capabilities to investigate incoming and ongoing relations from the program elements in the concern graph, and based on these relations to add new elements to the concern.

In contrast to SoQueT, the primary representation of a concern in FEAT is based on *elements* participating in the concern implementation. These participants can further be described by their relations with other elements in the graph. SoQueT on the other hand, emphasizes *relations* specific to crosscutting concerns implementation.

The Concern Manipulation Environment (CME) supports its own (pattern-matching) language for code querying. Furthermore, it allows for restriction of the query domain, similar to context definitions in SoQueT [8]. Although viable in theory, practical implementation of SoQueT's query templates using CME's infrastructure was hindered by the lack of a precisely defined syntax and a complete implementation of CME's query language. Private communication with the CME developers identified the template queries, as available in SoQueT, as a desired extension for CME.

[2] java.sun.com/developer/releases/petstore

JQuery is a code browser that uses a logic query language (TyRuBa) similar to Prolog. It is more flexible than CME for querying code and it covers more relations than the previous tools. One of JQuery's limitations is that it does not allow to save and then re-load a concern model of choice for a given project. The tool is also not suitable for large systems due to performance issues.

The focus in SoQueT is less on code browsing capabilities, as for the most of the aforementioned tools, but mainly on a consistent description of typical relations in crosscutting concerns implementation and on guiding the user in documenting such concerns.

6. Conclusions

This paper describes SoQueT, a tool supporting persistent documentation of crosscutting concerns using queries. The tool allows users to instantiate sort-specific query templates to reveal the crosscutting relations and their participant elements in the code under investigation. SoQueT has been used to document crosscutting concerns in various Java systems, such as JHotDraw and the PetStore J2EE application.

References

[1] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, 2003. Version 1.2.

[2] W. Harrison, H. Ossher, S. M. S. Jr., and P. Tarr. Concern modeling in the concern manipulation environment. Technical Report RC23344, IBM T.J. Watson Research Center, 2004.

[3] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. 2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, 2003.

[4] M. Marin. Formalizing typical crosscutting concerns. Technical Report TUD-SERG-2006-010, Delft Univ. of Technology, 2006.

[5] M. Marin, L.Moonen, and A. van Deursen. A classification of crosscutting concerns. In *Proc. 21st Intl. Conf. on Software Maintenance (ICSM)*, 2005.

[6] M. Marin, L. Moonen, and A. van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *Proc. 13th Working Conf. on Reverse Eng. (WCRE)*, 2006.

[7] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. 24th Intl. Conf. on Softw. Eng. (ICSE)*, 2002.

[8] P. Tarr, W. Harrison, and H. Ossher. Pervasive query support in the concern manipulation environment. Technical Report RC23343, IBM T.J. Watson Research Center, 2004.

TUD-SERG-2007-005
ISSN 1872-5392