Delft University of Technology
Software Engineering Research Group
Technical Report Series

Documenting Typical Crosscutting Concerns

Marius Marin, Leon Moonen and Arie van Deursen

Report TUD-SERG-2007-010

TUD-SERG-2007-010

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 Proccedings of the 14th IEEE Conference on Reverse Engineering

Documenting Typical 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

Our analysis of crosscutting concerns in real-life software systems (totaling over 500,000 LOC) and in aspect-oriented literature, indicated a number of properties that allow for their decomposition in primitive building blocks which are atomic *crosscutting concerns. We identify these blocks as* crosscutting concern sorts*, and we use them to describe the crosscutting structure of many (well-known) designs and common mechanisms in software systems.*

In this paper, we formalize the notion of crosscutting concern sorts by means of relational queries over (object oriented) source models. Based on these queries, we present a concern management tool called SOQUET*, which can be used to document the occurrences of crosscutting concerns in object-oriented systems. We assess the sorts-based approach by using the tool to cover various crosscutting concerns in two open-source systems: JHotDraw and Java PetStore.*

1. Introduction

The typically ill-modularized, scattered and tangled implementation of crosscutting concerns in existing software systems is known to be a challenge to understanding, and hence to the maintenance and evolution of these systems. Despite significant research efforts on the design and development of aspect-oriented languages as well as on concern identification techniques, there is still little consensus on what exactly constitutes a crosscutting concern, and how such concerns can be recognized, understood, and clearly documented in source code. We believe that (enabling) consistent understanding and documentation of crosscutting concerns in existing code is the key to making such systems easier to comprehend and maintain.

Over the last three years, we have analyzed crosscutting concerns in a range of Java systems, including JBoss, TOMCAT, JHOTDRAW, and the J2EE PETSTORE, totaling over 500,000 lines of code. A detailed description of the crosscutting concerns in the latter three of these systems is provided in our earlier work and accompanying web site [12].

In our study, we found several "building blocks" for crosscutting concerns. An example is the superimposition of a new role on an existing class. An instance of such a role superimposition can be found in the drawing application JHOTDRAW, in which all classes representing figure types that are to be stored on file should implement the "Storable" interface. In AspectJ such a crosscutting concern would typically be implemented through an "introduction" mechanism.

Another building block we noted involves "consistent behavior". As an example, again from JHOTDRAW, all execute methods as occurring in the Command hierarchy consistently invoke their super method in order to check certain preconditions. Here the AspectJ equivalent is a pointcut and advice.

In our previous work, we identified a dozen of such basic building blocks and labeled them "crosscutting concern sorts" [13]. These sorts (described in Table 1) can be used on their own, but can also be composed to construct more complex designs or features. For example, the Observer pattern, often used as a typical example of crosscutting behavior, can be seen as consisting of two role superimpositions (one for the Subject and one for the Observer role), and two consistent behaviors (one for the notification and another for the observer registration). Likewise, we have seen several other well known crosscutting concerns that can be composed from our sorts, like transaction management and undo support.

Our previous work on crosscutting concern sorts essentially consists of a list of 12 sorts, each explained informally in one or two paragraphs [16, 13]. There are, however, still a range of open questions: What exactly is a "sort"? Is there a way to formalize this notion? Based on such a formalization, would there be a way to unambiguously identify the occurrence of a sort in source code? Can we offer tool support for documenting crosscutting concerns based on sorts? And, last but not least, how "typical" are the sorts we have proposed? Do these sorts indeed occur in practice? How often? How can certain well known crosscutting concerns as occurring in existing systems be captured using sorts? In this paper, we set out to provide an answer to questions such as these.

The main contributions of the paper are:

- We formalize the notion of crosscutting concern sort by characterizing each sort by a specific query over a model of the source code. We present the queries for the six most commonly encountered sorts in this paper.
- We have developed SOQUET, an Eclipse plug-in that implements the queries for each sort, which can be used to document crosscutting behavior in Java applications.
- We provide an in depth study of sorts occurring in existing Java systems. In particular, we use SOQUET to document a variety of crosscutting concerns as presently implemented in the drawing application JHOTDRAW and web application PETSTORE.

The next sections each cover one of these contributions, after which we conclude with a discussion on these results, a survey of related work, and an outlook towards future work.

2. Crosscutting Concern Sorts

In our previous work, we recognized several sorts of crosscuttingness, which recur in many well known and lesser known crosscutting concerns. The full list is shown in Table 1. We characterized each of these sorts by:

1. an *intent*, summarizing the purpose for which it is used;
2. an *implementation idiom*, explaining how the sort is typically implemented in an OO programming language;
3. an *aspect mechanism*, describing how the sort's implementation can be modularized using aspects.

In the present paper, we go one step further, and discuss how we can formalize the notion of a sort. To that end, we express the six most commonly encountered sorts as queries over a meta-model describing object oriented source code.

2.1. The query model

Our query model is aimed at providing a standard, formalized description of the relations underlying each of the crosscutting concern sorts. The model consists of a generic query definition and a set of query templates (*sort queries*) that capture the relations specific to each of the sorts.

A sort query is a binary relation between elements of two sets, the *source context* and the *target context*. The elements in these contexts are program elements, such as classes or methods. The relation between them is based on a combination of various source code relations, such as *call* or *inheritance* relations, that can be extracted using static analysis. A query can limit each context by selection clauses that impose restrictions to the elements that participate in the relation.

The elements and collaborations relevant to the sort queries are shown in Figure 1. These relations are used to pose restrictions in queries, such as "any method m that *is_of_type T*". The type of a method is considered to be its returned type.

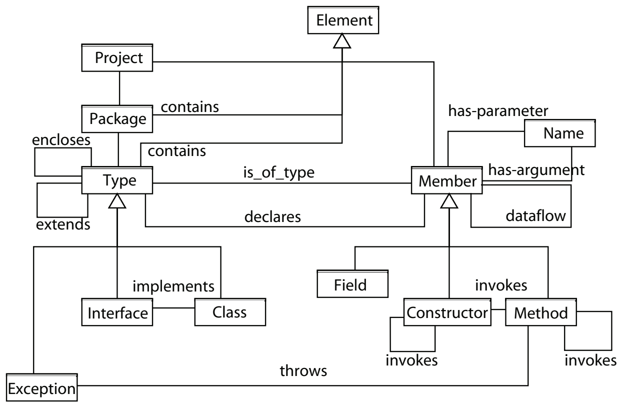


Figure 1. Meta-model relevant to sort queries

2.2. Formalization of sorts

This section covers a selection of the crosscutting concerns sorts shown in Table 1. Because of space constraints, we only discuss those six sorts most commonly encountered in practice. The remaining sorts are covered in the technical report [11].

(Method) Consistent Behavior The crosscutting relation specific to this sort occurs between a set of methods in a defined (source) context and a given action implemented by a method. The methods in the set consistently invoke the action to fulfill a requirement additional to their core functionality.

While the target context is defined through one method, namely the invoked action, the definition of the source context can cover various cases; for example, a logging instance could define the set of all methods in a Java project as source context. In this case, the seed to define the context is a project element. Other contexts could cover a type hierarchy, or just the set of methods of a class, etc. Each of these contexts requires a different (type of) seed element. Our helper function $\text{Context}_{\text{CB}}$ distills all methods from a given starting point s.

We formalize the concern sort and document its instances through a query that takes as input the invoked method and the seed element to define the source context.

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

The common idiom to implement instances of this sort in an object-oriented (particularly Java) language consist of scattered method calls (from a defined context) to the method implementing the common action to be executed consistently.

An example of consistent behavior is the notification mechanism in the Observer pattern: actions that change the state of the Subject have to consistently call the notification method to allow the observers to update their state.

Another example from transaction management is aimed at maintaining data integrity by ensuring that an operation is committed only when it is fully completed and rolled-back otherwise (e.g., in bank transfers both the debit and credit operations have to succeed to keep the data in a consistent state). Transaction management in Java is supported via JDBC transactions and the Java Transaction API (JTA) [1]. A JTA transaction requires that methods implementing the transaction logic consistently invoke dedicated methods of the *javax.transaction.UserTransaction* interface: the `begin` method at the beginning and the `commit` (or `rollback`) at the end to demarcate a JTA transaction. These invocations are instances of the *Consistent behavior* sort.

Other instances of this sort include Logging of exceptions thrown in a system; Wrapping service level exceptions

[1] `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/`

Sort	Short description
(Method) Consistent Behavior	A set of method-elements consistently invoke a specific action as a step in their execution.
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 (Pass Context)	Methods in a call chain consistently use parameter(s) to propagate 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 enforces (and makes explicit) the relation between the role of the enclosing class and that of the support class.
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.
Contract Enforcement	A set of method-elements consistently check a common condition.
Interfacing Layer	Elements implementing different roles are highly coupled, with one element (e.g., an UI component) wrapping and interfacing the other (e.g., action-controller, model) to mirror its state and action-response.
Add Variability	Clients consistently pass method-objects to server elements which invoke them via these callback methods.
Policy Enforcement	References between program elements are restricted as part of a (system) policy rule.
Design Enforcement	A set of elements are forced to comply with specific design rules, such as "all types in a context have to declare a no-argument constructor".
Dynamic Behavior Enforcement	A set of elements implement certain rules for object use, such as the use of life-cycle methods.

Table 1. Sorts of crosscuttingness. The first six sorts are discussed and formalized in this paper.

of business services into application level exceptions [12]; Checking credentials as part of authorization [10]; etc.

Redirection layer A redirection layer defines an interfacing layer to an existing object (to add functionality or change the context) which acts as a front-end that accepts calls and redirects them to dedicated methods, optionally executing additional functionality. The consistent (yet, method specific) redirection logic crosscuts this layer's methods.

The relation for this sort is between the redirecting layer and the target object, and resides in the consistent redirection of calls between method pairs. The source context is defined by the class acting as a redirector, the target context is the type whose methods receive the redirection. The implementation idiom consists of the identical logic in the methods (of a class) that redirect their calls to partner methods of a given type.

The query to document instances of this sort is parameterized with the redirecting type and a reference to the object receiving the redirection:

$$\begin{aligned}&\mathrm{RL}(\text{Class } c, \text{Member } f) := \{\ (m,m') \mid m,m' \in \text{Method} \wedge \\ &\quad c \text{ declares } m\ \wedge\ f \text{ is\_of\_type } c'\ \wedge\ c' \text{ declares } m' \wedge \\ &\quad\ \text{invokes}(m)\ \cap\ \text{methods}(c') = \{m'\} \wedge \\ &\quad\ \text{invokes}^{-1}(m')\ \cap\ \text{methods}(c) = \{m\}\ \}\end{aligned}$$

where $\text{invokes}(m) = \{m' \mid m \text{ invokes } m'\}$, $\text{invokes}^{-1}(m) = \{m' \mid m' \text{ invokes } m\}$, and $\text{methods}(c) = \{m \mid c \text{ declares } m\}$.

Implementations of the *Decorator* pattern are common examples of instances of this sort. For example, decorators are used in JHOTDRAW to allow to attach elements like borders to *Figure* objects. The decorators for figures extend *DecoratorFigure*, which defines the set of methods to consistently forward their calls to the stored reference of the decorated object. Subclasses of *DecoratorFigure*, like *Border* or *AnimationDecorator*, override its methods to dynamically extend its functionality. Consistent redirection is also common in implementation of patterns like *Adapter* and *Facade*, as well as wrapper classes [12].

Expose context (Context passing) Instances of this sort are characterized by methods that are part of a call chain that use an (additional) parameter to pass context along the chain. The caller exposes its context to a callee by passing information to each method in the call stack of that callee. The idiom specific to this sort is the crosscutting declaration of additional parameters that are used to pass context.

The elements related by this sort are the caller that wishes to propagate the context info and the callees to which the caller passes the argument. We use transitive closure to get a complete description of context passing along the chain.

$$\begin{aligned}&\mathrm{EC}(\text{Method } m) := \{\ (m_1,n) \mid (m_1,n) \in \text{Method} \times \text{Name} \wedge \\ &\quad m \text{ has-parameter } n\ \wedge\ m_1 =\ \text{endpoint}(\ \text{invokes+}\ (m)) \wedge \\ &\quad \forall m_2 \in\ \text{invokes+}(m,m_1)\ .\ \exists n_2 \in Name\ . \\ &\qquad m_2 \text{ has-argument } n_2\ \wedge\ n \text{ dataflow } n_2\ \wedge \\ &\qquad m_1 \neq m_2 \leftrightarrow \neg\ \text{uses}(m_2,n_2)\ \}\end{aligned}$$

where *endpoint(invokes+(m))* returns last element of the call-chain started from m, and *uses*(m_2,n_2) indicates that m_2 uses the value of n_2 for something else than propagation to its callees (i.e. n_2 occurs in another statement than a call to the next methods on the call-chain towards m_1).

An example instance of this sort is the use of *IProgressMonitor* objects to monitor progress of long-running operations in Eclipse applications. These operations are passed a reference to the monitor class through a parameter. They invoke methods of this monitor to indicate progress, like the `worked(int)` method to indicate that a given number of work units of the executing task have been completed. Any sub-operations receive a reference to the same monitor and use it to report their contribution to general progress.

Laddad discusses several other examples of concerns of this sort, part of transaction management or authorization mechanisms, and proposes an AspectJ solution to improve modularization (the Wormhole pattern) [10].

Role superimposition This sort applies to types that implement a specific secondary role or responsibility, like classes that participate in multiple collaborations and hence implement multiple roles [18]. The idiom is implementation of multiple interfaces (or methods that can be abstracted into an interface), and possibly the declaration or inheritance of attributes to support secondary role(s). The crosscuttingness resides in the entanglement of multiple roles in a single class.

In order to get those classes that implement an extra role, other than their defining hierarchy, we specify a seed element for the source context, as well as the role-element. The definition of the context is similar to that of the *Consistent behavior* sort. The role-element is specified as an interface or class, or, as we shall see in Section 3, as a *virtual interface* if the role does not have a dedicated type.

$$\begin{aligned}&\mathrm{RSI}(\text{Element } s, \text{Type } role) := \{\ (t, role) \mid\\ &\quad t \in \text{Type} \cap \text{Context}_{\text{RSI}}(s)\ \wedge\ (\ (role \in \text{Interface} \wedge\\ &\quad t \text{ implements } role) \vee (role \in \text{Class}\ \wedge\ t \text{ extends } role)\ )\ \}\end{aligned}$$

Common examples of *Role superimposition* occur in implementations of design patterns defining specific roles, like the Visitor pattern, or the Observer pattern discussed before.

The implementation of persistence is also possible through role superimposition: the *Figure* elements in JHOTDRAW implement a *Storable* interface which defines the methods for a (figure) object to write and read itself to and from a file. Each figure implements these methods in a specific way to provide persistence and recovery of drawings over work sessions.

Other examples of this sort are based on implementations of multiple interfaces with dedicated, specific roles, e.g., *Serializable*, *Cloneable*, *Undoable*, etc.

Support classes for role superimposition The mechanism of nested classes enforces and explicates the relation between the enclosing and the nested classes. This allows for superimposition of roles, as a number of elements can share a common role by enclosing support classes of a specific type.

Role superimposition through nested, support classes typically occurs for complex roles, and as an alternative to implementation of multiple interfaces: two hierarchies can interact by having common classes (that implement elements from both hierarchies) or by having elements from one hierarchy as support classes for the elements in the second hierarchy.

The relation between the enclosing and the support class (and their respective roles) can be formalized as:

$$\begin{aligned}&\mathrm{SC}(\text{Type } c, \text{Type } role) := \{\ (c_1, c_2) \mid c_1, c_2 \in \text{Type} \wedge\\ &\quad c_1 \text{ implements } c\ \wedge\ c_1 \text{ encloses } c_2\ \wedge\ c_2 \text{ implements } role\ \}\end{aligned}$$

An idiomatic implementation of the sort exhibits interaction of hierarchies through containment of nested classes.

An instance of this sort is present in JHOTDRAW as part of the support for undo functionality. Two type-hierarchies interact through support classes by having the members of one hierarchy enclosing members of the second one. The main hierarchy, *Command*, defines command elements for executing various application-specific activities like, copy and paste, or operations for setting the attributes of a figure. The second hierarchy, *Undoable*, defines operations for undo-ing and redo-ing the results of executing a command. Typically, each Command class encloses its associated Undo class.

Other examples of instances of the sort include implementations of specialized iterators for various Collection types and event dispatcher classes for managing notifications of listeners.

Exception propagation The intent of the sort is described by the consistent propagation of exceptions in a call chain when no appropriate response is available in the callers. Similar to context passing, the relation of the sort applies to a call chain. The callers implement the consistent (enforced) logic of re-throwing exceptions if they are not able to handle them.

To document such a concern, the query needs the method in which the exception originates and the type of exception. Optionally, a context seed can be provided to restrict the set of methods considered for (re-)throwing the exception:

$$\begin{aligned}&\mathrm{EP}(\text{Element } s, \text{Method } m, \text{Exception } e) := \{\ (m', e) \mid\\ &\quad m, m' \in \text{Method} \cap \text{Context}_{\text{EP}}(s)\ \wedge\ m' \text{ invokes+ } m \wedge\\ &\quad m \text{ throws } e\ \wedge\ m' \text{ throws } e\ \}\end{aligned}$$

Common examples of this sort in Java are implemented by not catching a given exception but re-throwing it through the declaration of a *throws* clause. A concrete example is the *IOException* discussed later in Section 4.1.

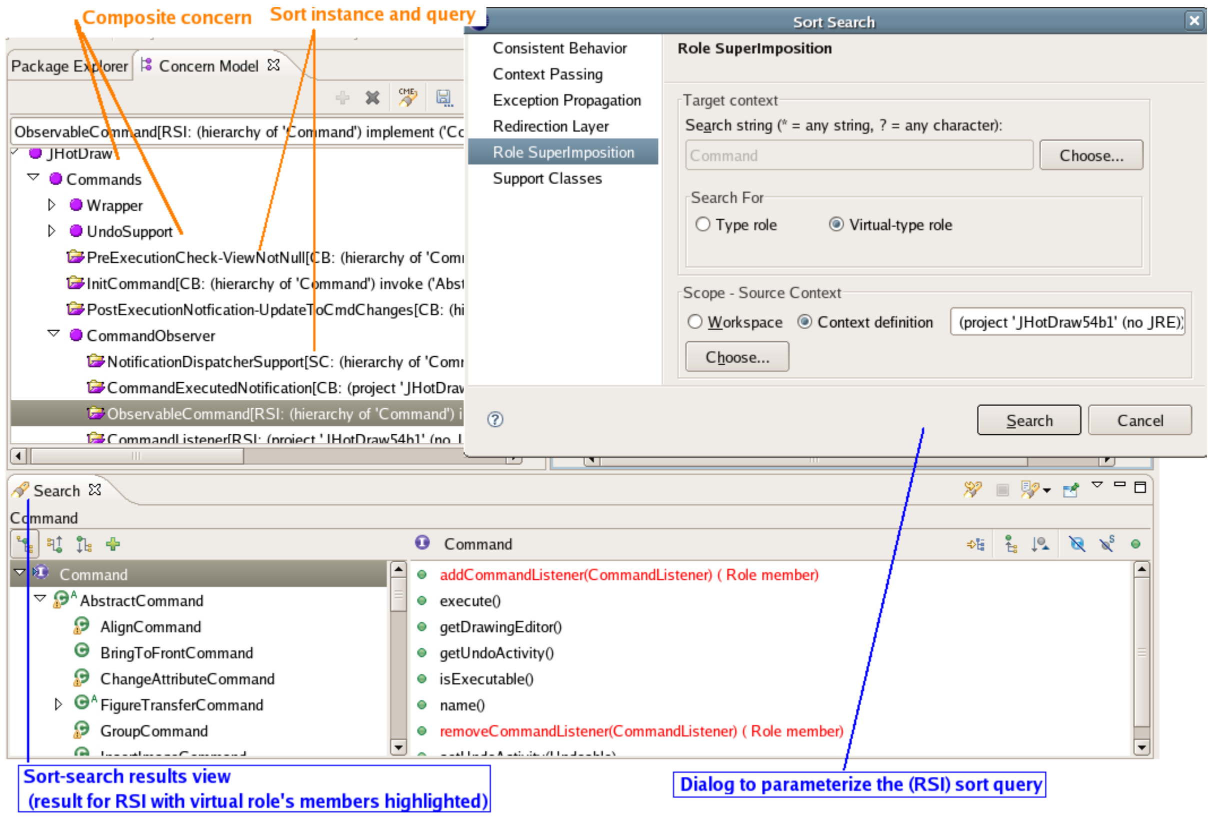
3. Sort-Based Concern Modeling

Concern modeling tools support software engineers in modeling their software systems in terms of concerns. Examples of such tools are the Concern Manipulation Environment CME [8], and the Feature Exploration and Analysis Tool FEAT [19]. A study by Robillard and Murphy shows that concern models help software maintenance [19].

Currently, concern modeling tools create rather low level models that are built from concrete source elements from the system that is documented. We propose the use of *crosscutting concern sorts* to raise the level of abstraction in concern modeling. We integrate sorts into concern models by permitting queries as elements in the concern hierarchy.

To support sort-based concern modeling, we have built an Eclipse plug-in called SOQUET (SOrts QUEry Tool) that is available from our website.[2] This tool allows one to describe crosscutting relations in a system based on querying the system's source code for instances of crosscutting concern sorts.

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



ObservableCommand[RSI: (hierarchy of 'Command') implement ('Cc
JHotDraw
Commands
Wrapper
UndoSupport
PreExecutionCheck-ViewNotNull[CB: (hierarchy of 'Comi
InitCommand[CB: (hierarchy of 'Command') invoke ('Abst
PostExecutionNotfication-UpdateToCmdChanges[CB: (hi
CommandObserver
NotificationDispatcherSupport[SC: (hierarchy of 'Comi
CommandExecutedNotification[CB: (project 'JHotDraw
ObservableCommand[RSI: (hierarchy of 'Command') i
CommandListener[RSI: (project 'JHotDraw54b1' (no

Consistent Behavior
Context Passing
Exception Propagation
Redirection Layer
Role SuperImposition
Support Classes

Role SuperImposition
Target context
Search string (* = any string, ? = any character):
Command
Search For
○ Type role ◉ Virtual-type role
Scope - Source Context
○ Workspace ◉ Context definition (project 'JHotDraw54b1' (no JRE)

Command
Command
AbstractCommand
AlignCommand
BringToFrontCommand
ChangeAttributeCommand
FigureTransferCommand
GroupCommand

addCommandListener(CommandListener) (Role member)
execute()
getDrawingEditor()
getUndoActivity()
isExecutable()
name()
removeCommandListener(CommandListener) (Role member)

Figure 2. SoQueT views and dialogs

These queries can be composed and stored to create persistent, sort-based documentation of concerns in existing code. The tool's main user interfaces are shown in Figure 2.

SoQueT assists the user in documenting and/or understanding crosscutting concerns in a system in the following way: First, the user defines a query for a specific sort based on its predefined template. The template guides the user in querying for elements that pertain to concrete sort instances and the user can restrict the query context, for example, by limiting it to a certain inheritance hierarchy.

Next, the results of the query are displayed in the *Sort-search results* view. This view provides a number of options for navigating and investigating the results, like display and organization layouts, sorting and filtering options, links from the query results for source code inspection, etc.

Finally, a *Concern model* view allows one to organize sort instances in composite concerns and describe them by user defined names. The concern model is a tree that defines a view over the system that is complementary to Eclipse's standard package explorer. The system's sort instances are leaves in this tree and intermediate nodes describe composite concerns. An element representing a sort instance element can be selected to re-run its associated query and inspect the results. Note that queries can be associated only with sort instances and not to a composite concern. A model can describe complex concerns, system features, or whole projects.

SoQueT introduces the concept of a *virtual interface* to define and describe a role whose definition is tangled within another type. This mechanism allows the user to create a virtual 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. Further implementation details are available in a formal research demonstration paper [15].

4. Sorts in Practice

In this section, we look at how sort instances occur in real systems. We describe two cases, totaling around 40,000 non-comment lines of code, from different application domains: JHotDraw[3] is an open-source drawing application and framework, and PetStore is a sample J2EE e-business application (an on-line shop) developed by SUN.[4] These applications have regularly been used as benchmarks in (collaborative) aspect mining studies, and detailed reports of earlier findings have been published [12, 2, 14, 17].

Discussion of the first case is based on the main sorts that occur in our sort-based concern documentation of the case

[3] `http://jhotdraw.org/`, version 5.4b1
[4] `http://java.sun.com/blueprints/`, Java PetStore v. 1.3.2.

using SoQueT. The concern model created for this case can be downloaded from the same web-site as the tool. We discuss a significant number of instances to show how "typical" the proposed sorts are and how they occur in practice. Table 2 shows the number of identified and documented sort instances. A number of additional sorts, not discussed in this paper, are described in a technical report which presents concerns in JHotDraw organized by design patterns [11].

The organization of the second case is aimed at showing how well-known crosscutting features and mechanisms can be decomposed and captured using crosscutting sorts.

4.1. JHotDraw

Figure 3 shows the core components of JHotDraw's architecture and their collaborations. The *Figure* type generalizes the notion of geometrical and text figures in the application. Figure elements support core operations like drawing and management of their display box. On top of these responsibilities, Figures participate in collaborations that require implementation of multiple roles, such as *observability* for changes, *composability*, and *visitability* for insertion or deletion of figures in composites. Moreover, figures and their enclosing drawings are *persistent* and implement specialized (`read` and `write`) methods defined by the *Storable* interface.

JHotDraw's user interface contains menus to execute operations like storing drawings and manipulating figures. It uses a dedicated *Command* hierarchy of around 40 elements to implement these operations. Figures can also be manipulated through a set of *Tool*s, such as a selection or copy tool. Tools access figures via a common interface realized by *Handle* elements, which act as Figure adapters.

Operations on figures notify listeners, such as drawing views, of changes and support *undo* functionality of such changes. Below we give an overview of the sorts encountered in JHotDraw together with their concrete instances.

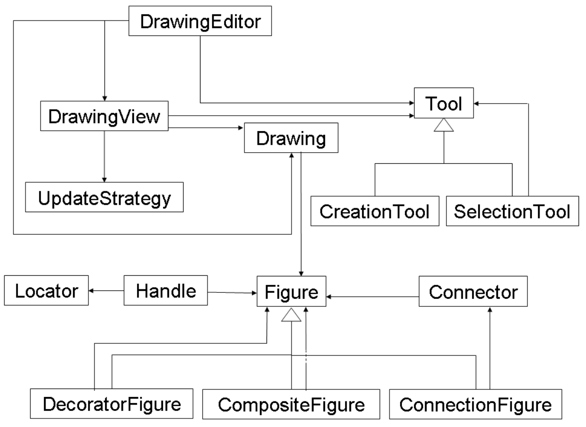


Figure 3. Collaborations in JHotDraw

Role superimposition We document the multiple responsibilities in the *Figure* elements as distinct instances of the *Role superimposition* sort. Roles like observability or manipulating parts of a composite figure require the use of virtual interfaces since their elements are tangled within the main *Figure* interface. Concerns such as persistence simply require to pass the *Storable* interface as a parameter to instantiate the sort-query. A similar case holds for drawing persistence.

Other instances of *Role superimposition* describe listeners for Command and Tool events. Listeners typically implement a secondary, dedicated interface. However, in JHotDraw the elements of the *Observable* role are defined by the same top interfaces as the core concerns of each of the two elements. These interfaces also include elements of the role for undo support, which requires another virtual interface.

The Command, Tool and Handle elements participate in various design patterns, like *Command*, *State*, and *Adapter*, respectively. We use sort instances to document each of the roles associated with these patterns in the three elements.

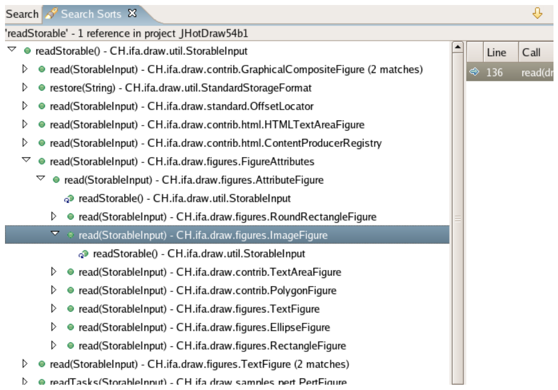
Other elements of our documentation are drawing views and editors that are common event and change listeners.

Consistent behavior and Contract enforcement A large number of concern instances documented for JHotDraw belong to the *Consistent behavior* sort. Some of these implement notification mechanisms for the various *Observer* designs, like drawing and figure changes, tool state changes, etc. Others crosscut the *Command* hierarchy, like consistently checking that a view is active before command execution, and refreshing that view after execution. The same (named) commands initialize the reference to the associated undo activity prior to their execution, and save the set of affected figures. Similar instances are present in the Tool and Handle implementations. The specific Undo activities for an activity also consistently conduct a number of checks before execution, like checking the undo/redo state of that action.

Consistent behavior is also present in several constructors, like *Command*s and *Tool*s, and in mouse or key handling actions, that implement a shared functionality by means of *super* calls. These concerns cut across specific type hierarchies.

Exception propagation The operations to implement persistence, like reading Figure objects from input streams, throw *IOException*s if not successful. The exception is propagated upward in the call chain to the method handling the drawing-recovery command triggered by the user actions, which catches the exception and prints an error message. We document the mechanism in SoQueT through an instance of the *Exception propagation* sort: The query starts from the method generating the exception of the given type, and displays recursively those callers re-throwing the exception.

Figure 4 shows the results of the query in the SoQueT view; The nodes for the callers re-throwing the exception can



```
'readStorable' - 1 reference in project JHotDraw54b1
readStorable() - CH.ifa.draw.util.StorableInput
    read(StorableInput) - CH.ifa.draw.contrib.GraphicalCompositeFigure (2 matches)
    restore(String) - CH.ifa.draw.util.StandardStorageFormat
    read(StorableInput) - CH.ifa.draw.standard.OffsetLocator
    read(StorableInput) - CH.ifa.draw.contrib.html.HTMLTextAreaFigure
    read(StorableInput) - CH.ifa.draw.contrib.html.ContentProducerRegistry
    read(StorableInput) - CH.ifa.draw.figures.FigureAttributes
        read(StorableInput) - CH.ifa.draw.figures.AttributeFigure
            readStorable() - CH.ifa.draw.util.StorableInput
            read(StorableInput) - CH.ifa.draw.figures.RoundRectangleFigure
            read(StorableInput) - CH.ifa.draw.figures.ImageFigure
                readStorable() - CH.ifa.draw.util.StorableInput
            read(StorableInput) - CH.ifa.draw.contrib.TextAreaFigure
            read(StorableInput) - CH.ifa.draw.contrib.PolygonFigure
            read(StorableInput) - CH.ifa.draw.figures.TextFigure
            read(StorableInput) - CH.ifa.draw.figures.EllipseFigure
            read(StorableInput) - CH.ifa.draw.figures.RectangleFigure
    read(StorableInput) - CH.ifa.draw.figures.TextFigure (2 matches)
    readTasks(StorableInput) - CH.ifa.draw.samples.pert.PertFigure

136    read(dr
```

Figure 4. SoQueT view for Exception Propagation

be expanded to display their own callers that propagate the same exception further in the call chain.

Redirection layer Besides the figure decorators that were previously discussed, JHOTDRAW contains a number of redirector instances for event handling and action delegation.

Command invokers implement the *ActionListener* interface whose only method, `actionPerformed`, consistently invokes the `execute` method of the associated command. We document this action delegation for command execution as an instance of the Redirection layer sort. Similar Redirection instances occur in key and mouse listeners which forward the handling of the captured event to associated tools.

Finally, commands that can be undone are wrapped by an *UndoableCommand*, which redirects requests to the wrapped command. We document this concern using a sort query that reports those methods that consistently redirect from the wrapper to the *Command* reference. A similar wrapper is used for *Tool* elements. Redirection layers are also used to reverse undo/redo activities, as in the *UndoRedoActivity* class.

Support classes Commands use Support classes to implement undo functionality. Similar instances are present in the undo support for Tools as well as for Handles.

Other instances of this sort document *EventDispatcher*s nested classes that implement more complex notification mechanisms for various *Observer*s, like for *AbstractTool*. The support class stores the association between a tool and its list of observers, and notifies the observers of various events. To document this concern, we pass the *EventDispatcher* and the *Tool* hierarchy as input parameters to the sort query.

4.2. Enterprise Applications

Several mechanisms commonly encountered in enterprise (J2EE) application development, such as transaction management, persistence or component lookup are well-known to be crosscutting and amenable to aspect-oriented solutions [10, 3]. To elaborate on the coverage of real-life crosscutting concerns by crosscutting concern *sorts*, we discuss a number of these mechanisms as encountered in PETSTORE.

Resource lookup: Service locator and caching, Business delegate, and exception wrapping and handling PETSTORE uses *service locators* [1] to provide single access points for resource lookup. The (singleton) implementation of the service locator in the web tier includes a caching mechanism to hold references to enterprise bean home objects or Java Message Service (JMS) resources for re-use. The locator is used by a business delegate (*AdminRequestBD*) to lookup business components. The delegate handles the distributed component lookup, decoupling the business services from their (presentation-tier) clients, and is responsible for catching exceptions thrown by the underlying implementation and converting them to application exceptions. We refer to this mechanism as exception wrapping [12].

Documenting the caching is based on two crosscutting concern sorts: First, the caching support in the service locator is a secondary role and an instance of *Role superimposition*. The role elements comprise the map structure used for caching. Second, the component lookup uses *Consistent behavior* to first check the cache for the searched component, and then insert the key in the cache if not present.

The exception wrapping mechanism, present in both the locator and the business delegate, is an instance of *Consistent behavior*, very common in enterprise applications. The concern implementation consists of invoking the constructor of a specific exception type to wrap a caught exception. The delegate class also implements *Consistent behavior* for logging the caught (and wrapped) exceptions.

Persistence strategy PETSTORE allows online purchases from a list of items whose details are stored in an external catalog that is accessed only for creating these lists. The application uses Data Access Objects (DAOs) to read the catalog and keep the data access mechanism hidden.

Client access to the catalog is done through a stateless session bean (*CatalogEJB*), which gets the data (*Item*, *Product*, or *Category*) from its wrapped DAO object (*CatalogDAO*): The data-access methods of the bean forward their invocations to the business methods of the DAO object. The DAO object manages persistence of the (*Serializable*) catalog entries by accessing the Enterprise Information System (EIS).

The serialization of the catalog entries is an instance of *Role superimposition*. To further document the wrapping of the DAO object in the stateless session bean, we use the *Redirection layer* sort. Both the DAO and the session bean elements implement exception wrapping, by catching specific type of exception thrown by their business methods, and rethrowing a different exception type; the concern is an instance of the *Consistent behavior* sort.

The DAO objects managing access to the persistent storage exhibit several other instances of *Consistent behavior*: the business methods consistently require a (JDBC) connection before executing the specific query, and after execution, the connection is closed through specific method invocation.

Similar mechanisms and concerns are present in other J2EE persistence examples described in literature, such as the use of Hibernate[5] for persistence by Colyer et al. [3].

Transaction management Programmatic transaction management is often acknowledged as crosscutting due to the consistent calls to Java Transaction API (JTA) for demarcation of the transaction, which has to execute completely or not at all [10]. The transaction mechanism in PETSTORE is similar to the one discussed by Laddad in [10]: the application's web-tier does not benefit from automatic (declarative) transaction management and hence implements it through JTA calls (e.g., the *TemplateServlet* servlet).

Transaction control implies several instances of *Consistent behavior*: first, a lookup action provides a *UserTransaction* object that can be used to begin the transaction by invoking the object's `begin` method. Next, the execution of the operation is followed by a commit if no exception occurred, or by a rollback otherwise. These actions invoke specialized methods on the transaction object.

PETSTORE also uses instances of *Consistent behavior* for (simple) exception logging. Laddad's example exhibits an instance of the same sort to wrap the exception when the lookup operation for the transaction object fails.

The *ServletTemplate* class, which implements a templating service for composing multiple views in one page, dispatches specific requests to an appropriate template component by passing its `request` and `response` parameters as arguments. We document this mechanism as an instance of the *Expose context* sort.

Order processing center: Transition delegates Transition delegates are part of an asynchronous messaging system implemented by the application for processing customer orders. After an order is received, a number of activities execute specific operations in a defined succession; these include sending emails to customers to acknowledge orders, sending order documents to suppliers, or updating orders based on invoice information. At the end of its execution, an activity passes a message asynchronously to the next activity by using a dedicated transition delegate. The delegate knows the successive activity in the workflow to be notified.

The activities are implemented as message-driven beans and trigger the notification by first setting-up their related transition delegate and then invoking the delegate's `doTransition` method at the end of their work. This notification occurs as a distinct concern from the main logic of the activity sending it, and hence we document it as a (*Consistent behavior*) sort instance.

[5]hibernate.org

5. Discussion

Coverage of the crosscutting concerns by sorts The list of sorts is open-ended, and new sorts can be added, following the rules of the proposed catalog for formalizing concerns, if their relations cannot be covered by the existing sorts. With the present catalog we cover all concerns that we are aware of in complex cases, like PETSTORE and JHOTDRAW, as discussed in this paper. The analysis carried out and the examples accompanying the description of sorts also show that these sorts cover well many of the crosscutting concerns described in the aspect-oriented literature.

It is important to notice that while crosscutting concerns can be discussed or even recognized at a higher level of granularity, like an Observer design or a transaction management mechanism, the concerns described by sorts are meaningful on their own. In fact, common refactoring solutions typically address concern sorts, like introduction of roles or advice for consistent behavior, which are only presented in a larger context of a specific feature or design [10, 7]. The classification in sorts helps us to describe those crosscutting concerns at a consistent granularity level.

Crosscutting concerns as relations The sort-based approach to crosscutting concerns proposed in this paper looks at such concerns as non-explicit relations between two sets of elements of variable cardinality (i.e., the source and the target contexts respectively). Each sort is described by a distinctive relation captured by the sort's specific query. While the relation is common to all instances of the sort, the definition of the context can vary from instance to instance: for example, a `log` method (the target context) can be invoked by, and hence be crosscutting for, all the methods in a system (the source context), while a notification action (target) is invoked by the set of methods changing the state of an Observable object (source). Each of these instances of the *Consistent behavior* sort has particular contexts.

The definition of contexts is a research topic on its own. Current aspect languages cover definitions based on naming conventions or some structural relations (e.g., type hierarchies), but do not support specification of a context like "all elements changing the state of a Figure object". SOQUET allows for a number of context definitions, such as class, project, package, or hierarchy, as well as for simple collections of elements. Although the last option is very flexible, permitting any selection of elements, a concise definition based on common properties of the context set's elements is more relevant for the intent of the crosscutting concern.

Tool usage SOQUET can typically be 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.

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 locate and better understand certain crosscutting concerns in the implementation. This documentation highlights policies and contracts in the code that are relevant for software evolution tasks and migration towards aspect solutions.

The main problems with describing and documenting crosscutting concerns stem from to the flexibility of the tool for defining contexts, as discussed above. SOQUET could be improved by allowing set theoretic operations, such as the union of type hierarchies. Furthermore, defining contexts using pattern matching on names (e.g., all set* methods) is not implemented at the moment.

Adding new sort queries in the current version of the tool is also fairly complex and relies on the "extension points" mechanism in Eclipse. We are exploring how we can prototype our queries using tools that support more direct source code queries, as discussed in the next section. This support is, however, still limited at the moment.

Using sorts in aspect mining and refactoring We have used crosscutting concern sorts to define a common framework for aspect mining [14]. The framework uses the sort specific idioms as a starting point for the design of aspect mining techniques and as a reference for defining mining results representation that can ensure consistent comparison of techniques and results. The definition and description of sorts further allow us to conduct *idiom-driven aspect mining*, and to engineer techniques that target specific idioms and hence sort instances.

Sort instances also allow us to group elements participating in relevant crosscutting relations, which are not explicit in source code. In this respect, sorts are modular units comparable with aspects. Sorts are mainly aimed at supporting crosscutting concern comprehension by describing atomic elements in a standard, consistent way. However, sorts can be associated with template refactorings and sort queries can help instantiate such refactorings by selecting the right program elements. This can help in refactoring concerns to aspect solutions.

The number of sort instances One observation about the data shown in Table 2 regards the number of sort instances in JHOTDRAW case compared to the PETSTORE one. A reason for this difference is the nature of the two applications: PETSTORE is a J2EE applications and a number of (potential) crosscutting concerns can be dealt with by the container, such as declarative transaction management. Therefore, these concerns do not occur in the source code.

Sort	**JHotDraw**	**PetStore**
(Method) Consistent behavior	34	15
Contract enforcement	5	1
Redirection layer	15	2
Expose context (*Context passing*)	1	1
Role superimposition	32	2
Support classes for role superimposition	5	0
Exception propagation (Declare *throws* clause)	11	5
TOTAL	103	26

Table 2. Number of sort instances.

Another observation is that some sorts are (typically) more common than others, like *Consistent behavior* or *Role superimposition*. This suggests that aspect language mechanisms aimed at refactoring instances of these sorts could address most of the encountered crosscuttingness. However, as previously mentioned, a major difficulty in dealing with these concerns resides in the ability to define contexts for the sorts relations. This is similar to the challenge of having a flexible and expressive pointcut definition in an aspect-oriented language.

6. Related Work

Our approach differs from related work by identifying typical implementation idioms of crosscutting concerns which we formalize as sorts. The sorts define a system to consistently describe crosscutting implementation of concerns, which helps us to recognize and document such concerns in source code. Furthermore, the approach emphasizes relations rather than program elements as crosscutting.

A number of tools support source code querying and exploration for concern understanding. FEAT organizes program elements that implement a concern in *concern graphs* [19]. The user can add elements to a concern graph by investigating the incoming and outgoing relations to and from an element that is part of the concern implementation. The elements in a concern graph are classes, methods or fields connected by a *call, read, write, check, create, declare,* or *superclass* relation.

Although the tool allows one to add relations to the graph describing a concern, the focus is on the elements participating in the implementation of the concern. The navigation for understanding a concern and incrementally building its graph representation is from a root (class) element to other elements in the relation chain. That is, a concern is described by its elements, and an element is described by its relations. Unlike FEAT, the sorts-based approach uses relations as the main representation of a concern and builds concern models based on these relations. Moreover, FEAT has no built-in support for describing typical crosscutting relations, focusing on code browsing and organization instead.

The Concern Manipulation Environment (CME) [8] also allows for code querying, and, furthermore, for restricting the query domain similar to context definitions in SoQueT. The CME concern model is persistent and the queries, written in its own (pattern-matching) language Panther [20], can be saved over work sessions. However, neither CME nor FEAT allow for complex queries like the ones we used to describe redirections, (exception) propagations or support classes.

We analyzed the possibility to use CME's query language and its internal code representation for implementing our sort queries. As yet the syntax of the language is not completely defined and is not fully implemented. Informal communication with CME developers revealed that they see queries as available in SoQueT, as a desired extension for CME.

Alike CME, JQuery is a code browser developed as an Eclipse plugin [9]. JQuery uses a logic query language (TyRuBa) similar to Prolog [4]. The TyRuBa predicates supported by JQuery cover all relationships defined by FEAT and include additional ones, such as checking the type of an argument. It also supports additional source relations with respect to FEAT and Panther, such as thrown exceptions.

Despite being more flexible than CME for querying code, JQuery 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.

Such performance improvements have motivated the work on CodeQuest, a software query tool using Datalog as a query language, implemented on top of a relational database system [6]. We plan to experiment with our template queries in CodeQuest's Datalog once the announced release of the tool becomes available.

Sextant is another tool similar to JQuery, which allows querying different kinds of system artifacts [5]. The tool represents these artifacts as XML and uses the XQuery [6] language to query this representation. Our focus so far has been on describing crosscutting relations in source code.

7. Conclusions

This paper proposes a system for addressing crosscutting functionality in source code based on crosscutting concern sorts. Such a system can provide consistency and coherence for referring to and describing crosscutting concerns. As a result, sorts are useful in program comprehension and areas like aspect mining and refactoring.

We have described crosscutting concern sorts as relations between sets of program elements and formalized these relations as queries over source representations. We have discussed a selection of sorts in detail and presented the SoQueT tool for documenting sort instances using queries. Last but not least, we have used sorts to analyze crosscutting relations present in systems from two different application domains.

[6] `www.w3.org/TR/xquery`

References

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns*. Sun Microsystems, Inc., 2003.

[2] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3), 2006.

[3] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ*. Pearson Education, Inc., 2005.

[4] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog, The Standard : Reference Manual*. Springer Verlag, 1996.

[5] M. Eichberg, M. Haupt, M. Mezini, and T. Schäfer. Comprehensive software understanding with Sextant. In *Proc. 21st IEEE Intl. Conf. on Softw. Maintenance*, pages 315–324. IEEE, Sept. 2005.

[6] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: Scalable source code queries with Datalog. In D. Thomas, editor, *Proc. 20th European Conf. on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.

[7] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. 17th ACM Conf. on OO Prog. Syst. Lang. & Appl.*, pages 161–173. ACM, 2002.

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

[9] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. 2nd Intl. Conf. on Aspect-Oriented Softw. Dev.*, pages 178–187. ACM, Mar. 2003.

[10] R. Laddad. *AspectJ in Action - Practical Aspect Oriented Programming*. Manning Publications Co., 2003.

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

[12] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. on Softw. Eng. and Meth.*, 2007. Accepted for publication, preprint available as Tech. Report TUD-SERG-2006-013, Delft Univ. of Technology.

[13] M. Marin, L.Moonen, and A. van Deursen. A classification of crosscutting concerns. In *Proc. Intl. Conf. on Softw. Maintenance*, pages 673–677. IEEE, 2005.

[14] 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 Engineering*. IEEE, 2006.

[15] M. Marin, L. Moonen, and A. van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *Proc. 29th Intl. Conf. on Softw. Eng.* IEEE, 2007. Formal research demonstration, preprint available as TUD-SERG-2007-005.

[16] M. Marin, L. Moonen, and A. van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *MACS '05: Proc. 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5. ACM, 2005. Softw. Eng. Notes 30(4).

[17] A. Mesbah and A. van Deursen. Crosscutting concerns in J2EE applications. In *Proc. 7th Intl. Symp. on Web Site Evolution*. IEEE, 2005.

[18] D. Riehle and T. Gross. Role model based framework design and integration. In *Proc. 13th ACM SIGPLAN Conf. on OO Prog. Syst. Lang. & Appl.*, pages 117–133. ACM, 1998.

[19] M. Robillard and G. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. 24th Intl. Conf. on Softw. Eng.*, pages 406–416. ACM, 2002.

[20] 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.

TUD-SERG-2007-010
ISSN 1872-5392