Chapter 1

SOrts QUery Tool (SOQUET)

This document contains the SOQUET user manual.

SOQUET \(^1\) is a query-based (crosscutting) concern modeling and documentation tool distributed as an Eclipse IDE (v.3.2.x – v.3.3) plug-in.\(^2\)

1.1 Installation

To install the tool, the user needs to download and save the “jar” distribution into the “plugins” directory of Eclipse and then (re-)start the IDE.

1.2 User manual

The documentation and modeling of concerns in SOQUET is based on a categorization of crosscutting concerns in so-called sorts. A concern sort describes elementary (atomic) crosscutting concerns that share their typical implementation idiom in an object-oriented language, like Java. For example, logging, authentication and authorization mechanisms, notification of changes in observable objects, etc., are typically implemented in Java by means of scattered calls to dedicated methods, such as org.apache.log4j.Logger.debug(message), java.security.AccessController.checkPermission(permission), or Subject.notifyObservers() (see the Observer pattern [Gamma et al., 1994]). We call the sort that describes these concerns and their common idiom, Consistent Behavior (CB), and we call each of the concerns above an instance of this sort.

SOQUET provides the user with a set of (6) query templates. Each query template describes the relation specific to the sort of concerns associated to it, such as method call relations or inheritance relations. The sort-queries can be parameterized by the

\(^1\)http://swerl.tudelft.nl/view/AMR/SoQueT

\(^2\)Some of the figures may be more difficult to read on paper. We refer the reader to the SOQUET web site for the on-line version of this manual, in which the figures are available in high resolution.
user in a SOQUET dialog window, to define concrete queries. The concrete queries are defined by the user so that their results map onto the source code elements that implement a particular, (atomic) crosscutting concern.

The parameterized sort-queries can be saved by the user in a dedicated SOQUET view to document crosscutting concerns by showing the underlying relations of these concerns as well as the program elements that implement them. Moreover, the queries can be grouped in composite, hierarchical models to show relations and associations between different concerns. Such models are aimed at assisting developers in understanding what concerns exist in a system and how these concerns are implemented.

This user-manual presents two use-case scenarios: in the first scenario, we use SOQUET to document crosscutting concerns in the JHOTDRAW drawing application, particularly in the Observer pattern for figure changes outlined in Figure 1.1.

In the second use-case scenario, we should use an existing concern model documenting various concerns in JHOTDRAW to assist us with a software change task.\(^3\)

### 1.2.1 Modeling and documenting concerns in SOQUET

Our first use-case scenario assumes the perspective of a developer that is familiar with a particular system, namely JHOTDRAW, and with the crosscutting concerns in this system. These concerns are not visible in the class decomposition of our system due to their crosscutting nature, and hence they are harder to notice and understand. Our goal here is to use SOQUET to document these concerns and make them explicit.

The main elements of the user interface in SOQUET consist of two Eclipse views and a dialog window for the sort-queries templates. One of the views is a customized extension of the default Search view in Eclipse. The results of our sort-queries will be displayed in this view.

The second view is the Concern Model view, which can be opened as shown in Figure 1.2. This view can be used to save the queries documenting concerns and to organize them in hierarchical, composite structures, similar to the ones in Eclipse’s Package Explorer view, shown on the left-side of Figure 1.2.

The sort-search dialog can be accessed from Eclipse’s Search menu, as shown in Figure 1.3. The dialog window presents the user with six query templates for the six most commonly encountered sorts of concerns. (If the option is not available in the Search menu, check that you are in the Java perspective and select some element in the Package Explorer view.)

---

\(^3\)This manual uses for exemplification JHotDraw 5.4b1, which can be downloaded from [http://sourceforge.net/projects/jhotdraw/](http://sourceforge.net/projects/jhotdraw/) or from the web-site of SOQUET. To use the concern model built for this application, the project has to be imported in Eclipse and named JHotDraw54b1. The source packages (CH.ifa.draw..) need to be placed in a ‘src’ folder – see the Package Explorer view in Figures 1.2 and 1.3.
Figure 1.1: Observer solution for figure changes in the JHOTDRAW drawing application.
Chapter 1. **SOrts QUEry Tool (SOQUET)**

Figure 1.2: Opening the Concern Model view.
Figure 1.3: Opening the sort-search dialog.
Documenting Consistent Behavior (CB)

A first step in documenting our Observer design for figure changes is to recognize those (atomic) crosscutting concerns that occur in this design. In Figure 1.1, we notice that each action (method) changing the state of a figure (for instance, moving or resizing a figure), consistently invokes the willChange method at the beginning of the execution and the changed method after the change is completed. The concerns to notify (pre-) changes in a figure follow the idioms described by the Consistent Behavior (CB) sort above, hence we will use the template for the CB query to document these concerns as CB instances.

The template for Consistent Behavior, shown in Figure 1.4, requires two parameters: the first is the method invoked consistently as part of the crosscutting concern to be documented (the target context). For our notification concern, this is the Figure.changed() method.

![Figure 1.4: The dialog to parameterize the Consistent Behavior query.](image)

The second parameter allows us to select from all the callers of the Figure.changed() method only those methods that are part of the concern that we want to document. In our case, all the calls to the changed() method from elements in the JHotDraw project are crosscutting and part of our notification concern. Therefore, we define our source context by selecting the whole JHotDraw project. Figures 1.5 and 1.6 illustrate how to do this.

After defining the two contexts and selecting the Search button in the dialog, the query will run and analyze all the elements of the JHotDraw project to identify calls to the Figure.changed method. The results of the search give us the participants in the notification concern. These are displayed in the Search view (see Figure 1.7).
Figure 1.5: Defining the source context for the CB query.
Figure 1.6: Select the whole JHotDraw project as the element that comprises all the callers of interest for our notification concern.
1.2. User manual

from where the user can navigate to the source code of these elements, organize them by various layouts, apply different filters, etc.

Figure 1.7: The results of the CB query that describes the figure-change notification concern as an instance of CB.

Saving the query documenting the concern

To save our query capturing the notification concern, we need the Concern Model view. A concern model in this view can include (1) atomic concerns, which are associated a concern sort query, as well as (2) composite concerns, which group together multiple sort instances (i.e., atomic concerns) and/or other composite concerns.

We start by creating a composite concern that will group together all the concerns in the JHotDraw project. Figure 1.8 shows how to create a new composite concern in the Concern Model view: Right click in the view, select Add New Concern, and introduce
a name for the new concern. In our case, our top composite concern is named by the same name as the JHotDraw project.

Figure 1.8: Create a composite concern for the whole JHotDraw project.

We continue with a new composite concern for our Observer pattern, in order to group together all the sort instances in this pattern’s implementation. This concern is a child of the composite for the JHotDraw project. The steps to add this new composite concern to the model are shown in Figure 1.9.

Now, we can add to our concern model the Consistent behavior sort instance documenting the notification concern, and its associated query. Following the steps in Figure 1.10, we start in the Search view, which displays the results of our sort query, and select “Add Sort Instance to Concern Model”. We then choose the parent concern of our sort instance: FigureChangeObserver.

The new concern shows up now in the Concern Model view, together with the description of its associated query (see Figure 1.11).

In case we added our sort instance to the wrong parent, we can select the Move option in the context menu of the instance (right click), and re-assign the parent. The Expand option in the same menu allows the user to re-run the query associated with the documented sort instance.
Figure 1.9: Create a composite concern for the Observer for figure changes.
Figure 1.10: Add a sort instance to the concern model for the Observer pattern.
Figure 1.11: Consistent Behavior instance in Concern Model and its context menu.
Documenting Role SuperImposition (RSI)

A second (atomic) crosscutting concern in our Observer occurs in Figure classes, which declare a number of members (on top of their main functionality) to allow listeners to register and receive notifications every time a change occurs in a figure’s state. These members define a secondary, crosscutting role implemented by Figures, namely the Subject (or Observable) role. Similarly, the listener elements, such as line connections between figures, have to implement the FigureChangeListener role that defines the methods for handling notifications from figures. Both these crosscutting roles follow a similar idiom, namely members declaration (and implementation) to support additional responsibilities. Same idiom can be observed in other well-known concerns as well, such as persistence and (special handling of) serialization in Java (java.io.Serializable).

All these concerns are instances of a different sort, namely Role Superimposition. The Listener role is already defined by a distinct interface: FigureChangeListener.

To document this crosscutting role, we select the Role Superimposition query in the Sort Search dialog, and pass as parameter the listener interface defining the role, as illustrated in Figure 1.12. The source context allows the user to specify what implementations of this interface are part of the concern to be documented by the query. Once again, we select the whole JHotDraw project, as all the implementations of the Listener interface are of interest.

![Figure 1.12: Selecting the type that defines the crosscutting role.](image)

The results of the query are displayed in the Search view, as shown in Figure 1.13. The view shows the FigureChangeListener hierarchy and highlights the members of the crosscutting role for each element in the hierarchy selected in the view.

We can add our sort instance for the listener role to the concern model for the FigureChangeListener Observer following the same steps as for the notification concern discussed...
Figure 1.13: Results of the query for the **Role SuperImposition** of listeners for figure changes.
Role SuperImposition (RSI) – virtual roles

The second crosscutting role we would like to document is implemented by Figure elements to allow listeners to register for and receive notification of changes. This is the Observable or Subject role in the Observer pattern [Gamma et al., 1994].

Unlike the listener role above, the Subject role in our Observer pattern implementation is not defined by a distinct interface. To document this role in SOQUET, we choose the Figure interface for the target context, and then select in the dialog for the Role SuperImposition query the Virtual-type role option. The selection opens a window, as shown in Figure 1.14, that allows us to select the members of the Figure classes that belong to the crosscutting role. The rest of steps are then similar to the documentation of the listener concern previously discussed.

The results of the search and all the concerns we documented so far are shown in Figure 1.15. We can now save our model by selecting the Save command in the Concern Model view.
Figure 1.15: The concerns documented in the FigureChange Observer.
1.2.2 Using SOQUET to aid program comprehension and software change tasks

A second use-case scenario we shall look at consists of using SOQUET and an existing concern model to support us in a software change task. In this case, we assume that JHOTDRAW is a new system into which we have little or no insight. We would like to use a provided concern model in SOQUET to be able to extend JHOTDRAW consistently with concerns already present in this system.

The change we would like to make consists of an extension of the Command support in JHOTDRAW, namely adding a Command for mirroring a selected figure in the drawing view.

Command classes in JHOTDRAW implement actions to be run from the application’s menus, such as copying and pasting figures in a drawing view, changing the color of a geometrical figure, etc. Each command implements the Command interface and implements the core logic of its action in the execute method. As an example, Figure 1.16 shows the command for deleting selected figures from a drawing (view).

(Most of the) Commands can also be undone. The logic of undoing a command is implemented by a nested UndoActivity class, for each command class. The UndoActivity for DeleteCommand is also partially visible in Figure 1.16.

![Figure 1.16: DeleteCommand in JHotDraw.](image)
We start our change task by loading in SOQUET an existing concern model that documents various concerns in JHotDraw, including concerns in the Command support. Figure 1.17 shows how to load a concern model in the tool.

![Image: Loading an existing concern model into SOQUET.]

For a first insight into our drawing system, we can simply explore the concern hierarchy in the Concern Model view and use the Expand option to run some of the queries documenting atomic concerns, and navigate the results, as shown in Figure 1.18.

For significantly large systems and concern models, SOQUET provides (partial) support to search the concern model for queries documenting concerns that cover a specific program element. Following the example in Figure 1.19, we select from the concern model only those concerns whose queries have as one of their end points (context elements) a Command element.

After selecting the Command interface (CH.ifa.draw.util.Command) from all possible name matches, the view will show us only the sort instances of interest (see Figure 1.20).

By examining the concern in the root of the JHotDraw concern model and expanding its query (Figure 1.21), we learn that our commands are grouped in the Command hierarchy. This hierarchy is rooted in the Command interface and a default abstract command implementation, AbstractCommand.

The other sort instances in the view, shown in Figure 1.22, document a command as a multi-role element, which implements members to support a Listener role as well as Undoable functionality.

Based on the knowledge gained from exploring the concern model, we can create a stub class for our MirrorCommand by extending AbstractCommand, and then
Figure 1.18: Exploring the implementations of concerns using the queries in the concern model.
1.2. User manual

Figure 1.19: Searching concerns for a program element in the concern model.

Figure 1.20: Filtered concern model.
Figure 1.21: Exploring the concern model by running the queries documenting concerns/sort instances.
distinguishing in this class between the different roles. The new class is shown in Figure 1.23.

After creating the stub, we can return to the full concern model and show all the concerns, as illustrated in Figure 1.24.

We continue with examining some other of the documented concerns in the root of the JHotDraw model to learn more about the implementation of Commands. One such concern is shown in Figure 1.25 – its description indicates a common pre-condition check for commands. We see by running the query for this concern and examining the results in the Search view that the concern’s implementation consists of method calls from execute methods in various Command classes. The documented call is the invocation of the super’s method, as shown in Figures 1.25 and 1.26. This call is aimed at checking a common condition in all commands, so we add it to the execute method of our new MirrorCommand as well.

Similarly, we can investigate all the other concerns in the model and ensure that the implementation of our MirrorCommand is consistent with all existing policies and rules for Command classes. This exploration of concerns leads us to the stub implementation of the execute() method for the MirrorCommand shown in Figure 1.27. The stub shows the crosscutting concerns in this method and makes it consistent with the other existing Command implementations. Now, we can go ahead with implementing the core logic of our class. This is not a crosscutting concerns so it is up to the user
Figure 1.23: The MirrorCommand stub distinguishes the multiple roles based on the documentation of concerns in SOQUET.
Figure 1.24: Filters settings for the concerns to be displayed in the Concern Model view.
Figure 1.25: The concern for pre-condition check before Command executions.

Figure 1.26: The execute() method in AbstractCommand.
1.2. User manual

of this manual to do it.

Figure 1.27: The crosscutting concerns in the MirrorCommand.execute() method.
1.3 Contact

Questions, comments, bug reports, etc. are welcome at Marius Marin: a.m.marin@tudelft.nl
Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA [1994].