As far as program comprehension is concerned

Tommi Reinikainen¹, Imed Hammouda¹, Juha Laiho², Kai Koskimies¹, and Tarja Systä¹
¹Tampere University of Technology, ²Nokia Research Center
tommi.reinikainen@tut.fi, imed.hammouda@tut.fi, juha.m.laiho@nokia.com, kai.koskimies@tut.fi,
tarja.systa@tut.fi

Abstract

A concern-based approach to support software comprehension and analysis is proposed. Concerns represent various aspects of a system that are interesting for some stakeholders, like features, architectural solutions, units of work, etc. Using concern-based queries on the software artefacts, new derived information on the system can be exposed. The results of the queries are also represented as concerns and added to the concern repository. The approach thus supports the analyzer in gradually building up an understanding on the subject system to be analyzed.

The proposed approach and tool support is applied in the analysis of an industrial large-scale product platform and products built on top of this platform.

1 Introduction

Identifying, understanding, and managing evolulutional changes in large software systems is challenging. When adding or changing a particular functionality, its implications to the rest of the system should be carefully analyzed. The parts affected by the change may be scattered over the system artefacts, making the task even more challenging.

Since manual change management can be tedious and errorprone, if not impossible in some cases, proper tool support is desired. A lot of methods and tools have been proposed to support source code analysis, feature analysis and identification, software refactoring, and visualization of evolutionary changes in software. These tools have and should be built keeping in mind the fact that software maintenance and evolution are human-intensive activities and typically task-driven. Thus, the tools often represent the subject software as a graphical model and provide tools for studying the model. Such aid helps the software engineer to understand the system artifacts and their relationships. Analysis and querying the model is often supported by scripting languages, ready-made implementations of graph algorithms, methods utilizing software product metrics, and other tools for graph manipulation and visualization.

However, representations reflecting the system structure are often unnecessarily complex for extracting and combining information that is relevant for a stakeholder. We argue that a complementary approach is to express various topics of interest in a system using the simplest possible abstraction, the set. When the topics of interest – hereafter called concerns – are expressed as sets of system elements, the process of representing and extracting information about a software system can be established on well-understood set theory. Assuming that the concerns have meaningful contents from the viewpoint of a stakeholder, the concerns constitute a higher abstraction layer for the software system, providing different views for different stakeholders. Intuitively understandable queries can be constructed using such concerns, revealing new derived information on the system.

Several definitions for concerns have been presented in the literature [14,18,11]. Kande, for instance, defines a concern as an aspect of a problem that is critical or otherwise important to one or more stakeholders [11]. This definition comes close to our view in this context. From a concrete point of view, a concern is a set of software system elements that form a unit of interest for some stakeholder. Concerns can be overlapping and nested in an arbitrary way. The elements belonging to a concern may be in principle any elements included in the software artefacts, like individual requirements, UML model elements, code fragments, XML elements etc. However, in this paper we are studying concerns in the context of UML class diagrams only.

Further elaborating the idea of a concern-based software repository suggested in [9], we present an approach for concern-based program comprehension. The approach and provided tool support allows the software engineer to define the concerns she is interested in, either manually or assisted by the tool. Based on an existing set of concerns, she can query on the relationships and evolution of the concerns, e.g. to find out how a change in a certain functionality influences other functionalities. In essence, compared to existing tools supporting software comprehension, we propose the use of an additional abstraction layer, consisting of a set of concerns, on the top of the software model. The
actual analysis is then done by querying this concern layer. We demonstrate the applicability of the approach and tool support by using them to analyze the source file structures of an industrial large-scale product platform and products built on top of this platform. A detailed account of the approach can be found in [16].

One of the main benefits of the proposed approach is that it supports the analyzer to gradually build up an understanding of the software model to be analyzed. The concerns in our approach can be seen as the basic building blocks and tools to support the analysis and to build a mental model of the (parts of the) software under analysis. New concerns can be easily constructed by applying set operations to the concerns. The user can, for instance, merge two concerns, build an intersection of them, or “subtract” one concern or its parts from the other.

The paper is organized as follows. In Section 2 we introduce the concern-based approach in more detail. The tool support implemented to support the proposed approach is outlined in Section 3. Application of the approach in analyzing an industrial-scale software system is discussed in Section 4. Finally, we conclude in Section 5 with discussion on related and future work.

2 Concern-based approach

The main steps in the concern-based approach are 1) constructing the concern-enabled artefacts, 2) identifying and creating the initial set of concerns, as long as the presentation language is supported by the tool.

2.1 Constructing the UML model

We have so far used our approach to support analysis of UML models given as class diagrams. They can be manually constructed design models or e.g. reverse engineered models as in our case study. The approach itself is independent from the way these models have been constructed and from the tools used to construct the models. As an example, the models used in our case study (Section 4) have been constructed using Nokia’s reverse architecting toolset [17], which can be used to transform information included in source (and/or binary) code files into higher level models, e.g. UML.

Besides being independent from the methods and tools to construct the UML models, the approach is, in principle, also independent from particular languages in which software artefacts are expressed. Since the tool platform we exploit for presenting concerns [8] supports Java and XML as well, the required concern-based query support could be easily built for these languages, too. The approach is conservative in the sense that it makes no assumptions on the form of the artefacts containing the concerns, as long as the presentation language is supported by the tool.

2.2 Identifying and creating the initial set of concerns

As concerns themselves are abstract and disconnected from the actual software system, we need a concern mapping that binds the concerns to the subject system. This mechanism consists of identifying the software artifacts (UML models in our case) to be included in the concern and binding them to the concern.

The identification of the elements to be included in a concern typically involves manual activities, since a concern is always a viewpoint of a stakeholder. Thus, a manual selection tool for building a concern is necessarily required. In some cases, however, many of the elements belonging to a concern can be identified on the basis of some structural conditions. In such cases, a large portion of the concern can be mapped automatically using a tool that finds the elements satisfying the condition, and binds them to a draft concern that can be further edited manually (for example by adding or removing new elements). As an example, the view classes participating in an MVC pattern often have “view” as part of their names. Thus, the tool can find potential participants of a “GUI Display” concern by searching for classes whose names match with a regular expression. We have implemented this kind of simple tool support, but other (semi)automated means to identify the concerns could be developed as well. This remains as a part of our future research.

In our approach, concerns are “intelligent” in the sense that they may store information about the required elements of the concern and about their required properties. Thus, if a change is made to the concerned model, e.g. a class is removed, the concerns related to that class may become broken, informing the user about the missing element. Such a feedback mechanism, even though not commonly supported by the currently available tools, is quite important for reminding the analyzer on the content of the concerns and for alerting her on their invalidness.

2.3 Concern operations

A set of concerns provides a multi-dimensional view of the system as partially overlapping structures that cross-cut the explicit system composition. These structures reveal certain configurations of software elements that constitute units of interest for various stakeholders, but
even more importantly, they provide an interconnected high-level search structure for a software system. Assuming that the concerns are the basic topics of interest, this kind of search structure can be exploited as the interface of a software knowledge base. In particular, new views to the system can be constructed using queries that combine the information in the existing concerns in various ways. The result of a query can have transient character, in which case it is used to satisfy a temporary need of a stakeholder, or it can be more persistent, holding permanent meaning and value for future analysis.

Standard set operations have been proposed as the core of a concern-based query language in [9]. In this paper, we present a realization for basic concern operations following this idea. Of the mathematical set operations, the ones that are focused on in our work are the union, intersection and difference operations. Their concern counterparts are defined in Table 1, with the corresponding set operation in parenthesis. A fourth operation, nearest neighbourhood (later referred to as neighbourhood), was also implemented. Unlike the other operations, the neighbourhood operation is not derived from mathematical set theory, but is rather an operation that was proven to be very useful in the course of our work. A common feature for all of these operations is that they take existing concerns as their parameters and return the result as a new concern.

The list of possible operations is in no way limited to the ones we decided to implement. For example, one of the unused mathematical set operations, the Cartesian product, is sensible in the concern realm as well. However, it was left unimplemented as it did not seem be useful in the scenarios we studied.

Table 1: Concern operations

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
<th>Commutative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge (union)</td>
<td>+</td>
<td>Binary</td>
<td>Merging of concerns A and B results in a concern containing all the elements of A and B, excluding duplicates.</td>
<td>Yes</td>
</tr>
<tr>
<td>Overlap (intersection)</td>
<td>&amp;</td>
<td>Binary</td>
<td>Overlapping A and B results in a concern containing all the elements that are common to both A and B.</td>
<td>Yes</td>
</tr>
<tr>
<td>Slice (difference)</td>
<td>-</td>
<td>Binary</td>
<td>Slicing of concern A with B results in a concern that contains the elements that belong to A but do not belong to B.</td>
<td>No</td>
</tr>
<tr>
<td>Nearest neighborhood</td>
<td></td>
<td>Unary</td>
<td>The nearest neighbourhood of concern A results in a concern that contains all elements that have a relationship with an element in A and that do not belong to A, including the direct or indirect parents of these elements. There can be different versions of this operation for different kinds of relationships.</td>
<td>-</td>
</tr>
</tbody>
</table>

All of the operations may be chained to form more complex expressions. The precedence between the binary operations is not defined (i.e. the precedence is from left to right), whereas the unary operation takes natural precedence before the binary ones. The order of execution can be altered by using parentheses. It is also important to note that the laws of distributivity and associativity for mathematical set operations [7] are also valid for the concern operations.

As a simple example of what can be done with the defined operations, assume there is a model with components A and B, and concerns FeatureX (containing all elements belonging to feature X), and Tested (containing all components that have been Tested). Further, assume that Concern(e) denotes a concern consisting of the element e only. Then, the following queries can be constructed:

- What are the tested clients of component A?
  
  \[ \text{Concern(A) & Tested} \]

- What are the components belonging to feature X that are not tested?
  
  \[ \text{FeatureX - Tested} \]

- What are the clients of either A or B that are not tested?
  
  \[ (\text{Concern(A) + Concern(B)}) - \text{Tested} \]


2.4 Concern query language

The operations defined above are the basis of our concern query language. This language is meant to be a
foundation that can be later supplemented with additional operations, such as the aforementioned Cartesian product. The EBNF syntax of the language is given in Table 2.

Table 2: EBNF of query language

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;exp&gt; ::= &lt;factor&gt; {&lt;binary_op&gt; &lt;factor&gt;}</td>
<td>An expression consists of a factor and zero or more binary operations on factors.</td>
</tr>
<tr>
<td>&lt;factor&gt; ::= [ &lt;unary_op&gt; ] &lt;primary&gt;</td>
<td>A factor is a primary optionally followed by a unary operation.</td>
</tr>
<tr>
<td>&lt;primary&gt; ::= &lt;allowed_char&gt; { &lt;allowed_char&gt; }</td>
<td>A primary is a sequence of allowed characters.</td>
</tr>
<tr>
<td>&lt;binary_op&gt; ::= '+'</td>
<td>A binary operation is addition.</td>
</tr>
<tr>
<td>&lt;unary_op&gt; ::= '</td>
<td>'</td>
</tr>
<tr>
<td>&lt;allowed_char&gt; ::= [a-zA-Z0-9]</td>
<td>An allowed character is a letter or digit.</td>
</tr>
</tbody>
</table>

2.5 Tracking concern evolution

As software systems evolve, so must the concerns. Since software evolution may entail anything from addition or omission of artifacts to subtle changes in relationships between artifacts, as well as merely changing a specific artifact, tracking the evolutionary changes of concerns is indeed a complex matter [16].

With a set of concerns bound to the artifacts, the problem becomes even harder, because the concerns must be revised according to a new version of the system. Obviously, such updating of the concerns cannot be fully automated in the presence of arbitrary changes. However, adequate concern tracking support can significantly assist in the updating process.

Our tracking tool works in the following way: given a model annotated with a set of concerns and another model (the new version of the model), the tool tries to find the matching elements in the latter model for each element in the original model that is part of a concern. Currently the matching is based purely on a simple name matching.

3 Implementing the concern toolset

3.1 Tool platform

The concern toolset was implemented as an extension of an existing architecting environment, INARI [10]. INARI is a prototype architecting environment supporting the representation of various structures that are not explicit in a software system. INARI is built on top of Eclipse platform and it uses Rational Software Architect [3] and its UML diagram capabilities extensively.

The concern abstraction was implemented using a structure already existing in INARI, the aspectual pattern [8]. An aspectual pattern (or pattern in short) consists of roles that can be bound to various software elements, like UML elements, Java source code elements, XML elements and so on. A pattern may introduce relationships and constraints for the elements that are bound to the roles, but in this context we can view a pattern instance (that is, a pattern with bound roles) simply as a configuration of software elements. The elements will be here model elements in a UML class diagram.

The INARI tool allows the user to build a pattern instance (that is, a concern) simply by selecting the included model elements in the class diagram editor. The resulting patterns (concerns) are shown in a separate view. In Figure 1, the user interface of INARI is shown when applied to a model of an electronic home system. In this case the concerns represent various features of the system, as well as organizational issues (like CompanyOwned) and process issues (like Tested). The pattern (concern) library is shown in the left, with the Tested concern selected (shown in the lower view) and highlighted in the RSA model editor (right).

The concern toolset was implemented as a suite of plugins, with separate plugins for creating new concerns, for running queries on existing concerns and for tracking evolutionary changes in concerns. The concern creation plugin was necessary to make it easier to create a new set of concerns from which we can refine a new set of concerns using queries.

The plugin for running concern queries is the main tool in the toolset, consisting of a lexer/parser pair for reading the query input and an engine for executing the queries. The lexer and parser were generated from an EBNF-like specification, using the ANTLR tool [1]. The engine, on the other hand, was written completely by hand. As extending the EBNF is fairly easy, it was also important to make the engine as easily extensible as possible.

The tracking plugin is used mainly to streamline the transition between two versions of a model. It tries to find the same elements from a new version of the model to which it was bound in the original. This remapping is done purely based on semantical information, in particular the concern element name. Looking for structural similarities has been discussed as an option for future work.
Every evolitional change of the subject system may cause changes in the concerns. A change in a concern, in turn, can be considered as a sequence of the following three scenarios: the concern may gain elements, it may lose elements, or it can remain as it is. We will discuss the former two cases in more detail in the sequel.

### 3.2 Concern evolves by losing elements

In this scenario, the evolved concern is a subset of the original concern, so the evolved concern does not contain anything that was not already contained by the original concern.

When this is translated to patterns, we see that the pattern that depicts the evolved concern (later referred to as the evolved pattern) consists of roles and role bindings that can all be found from the pattern that depicts the original concern (later referred to as the original pattern). In fact, the evolved pattern actually consists of the exact same set of roles as the original pattern did. The real difference is that some of the roles in the evolved pattern are not bound to any software element, simply because these elements do not exist in the evolved software artefact repository (e.g. a UML model). This is illustrated in Figure 2.

![Figure 1. INARI user interface with concerns](image)

![Figure 2. (a) The original pattern with all roles bound and (b) the evolved pattern with unbound roles](image)

From the set of roles in the evolved pattern, we can separate those that are bound to a software element and those that are not. The set of bound roles is actually the evolved concern, i.e. what is left of the original concern after the software has evolved. The set of unbound roles is the change in the concern. Both pieces of information can be useful, though usually in different situations.
3.3 Concern evolves by gaining elements

If a concern gains new elements, the evolved pattern has actually expanded its set of roles and role bindings. Finding these new roles and bindings cannot be fully automated. If the original pattern was formed by some kind of systematic rule (e.g. pattern consists of model elements with a given prefix in their names), the rule could be reapplied to the evolved software, but still the result would be uncertain.

A relatively common scenario is the changing of an element that is bound to a pattern. In practice, this can be understood as a concern losing one element and gaining another. On the pattern level, this situation is problematic only if the change occurs in the element somehow changes the way the element is perceived from the concern level. An example of this could be the changing of the element's name: depending on the final implementation, this may cause the tool to interpret that the element was completely removed. However, as long as the element seems unmodified when viewed from the concern level, any changes inside the element are allowed.

4 Case study

A case study was undertaken to test the applicability of the concern manipulation toolset in a practical environment. The toolset was to be tested on a reverse engineered model of Nokia’s ISA software platform. ISA, the software platform of a major mobile phone product family of Nokia, is a proprietary software platform created and maintained by Nokia.

ISA contains a wide array of functionalities, ranging from user interface driven applications, such as the phone book or the calendar, to complex, hardware-oriented functionalities such as telephony or camera operation. This all dictates that, on the source code level, ISA is a fairly complex system and dispersed in nature. This is not aided by the fact that ISA is in a continuous state of evolution.

Another feature underlining the dispersed nature of ISA is that for each product (e.g. mobile phone) that is running ISA, there exists a configuration specific for that product, defining which of the features of ISA are to be included in that product. No one mobile phone contains all the possible features that ISA contains, so what is actually installed on a product is always a subset of the entire ISA feature set. These characteristics of ISA provide a nice test environment for the concepts and tools presented in this paper.

4.1 Goals

ISA can be understood as a collection of compilation units containing modules, each of which is dependent on a number of header files. At some point during the development, a design decision was made to place the different header files in a single, global directory to streamline the compilation. However, as the platform evolved and grew, the number of header files placed in the global directory multiplied and reached a size that started to cause increasingly amount of problems.

The primary objective of the case study was to help solve this problem: to reduce the amount of dependencies between ISA's global header file repository and the different modules. To achieve this, Nokia Research Center (NRC) created a UML model that focused on presenting the hierarchy between different compilation units and modules and the dependencies each of those modules have with the header files. This was done using reverse engineering tools developed at Nokia[17]. One of the goals of this case study was to identify header files that were endpoints for only a small amount of modules (0-3 dependencies). Another goal was to find the specific compilation unit, under which all modules dependant on a specific header file resided. The results could then be used to find an optimal location for each header file in the hierarchy of compilation units.

The data on the modules and their dependencies was presented as a Rational Rose UML 1.X model. An example screen capture of this model can be seen in Figure 3. In the model, the modules that are dependent on the header files are presented as UML classes (and tagged with the stereotype <<Library>>), while the header files are depicted as UML interfaces. Compilation units are depicted as UML packages (with stereotype <<ISA Package>>) and the dependencies as UML dependencies. The model consists of the (global) repository of header files and a hierarchy of compilation units, each containing one or more modules. To show the dependencies for a single module more clearly, the model also contains a view for every compilation unit showing all the modules and the header files those modules depend on. Though the view is associated to a specific compilation unit, that does not mean that the header files presented in that view actually reside in that compilation unit in the hierarchy, nor that they should reside there. The view can be understood as a cross-cutting view of the system from the perspective of that compilation unit.
4.2 Problem solution

The key in solving this particular problem is the neighbourhood operation, defined in Section 2. Running the neighbourhood operation on a concern that consists of a header file returns a concern that contains every module that has a dependency relationship with that header file. The result concern also contains information of the structural nested location of the found modules. The pattern view of the INARI tool thus displays the result for a single header file in a convenient form, exhibiting the users of a header file together with their position in the system hierarchy. A typical neighbourhood concern view for a header file (HeaderB) is shown (with abstracted names) in Figure 6, as displayed by INARI. After getting this result concern, it is straightforward to deduce the number of dependencies for that header file, as well as the topmost compilation unit containing those models, and the location of the dependent modules in the compile unit hierarchy. This XML-data was then transformed into HTML using XSL transformation to allow the analysis of the results using any Web browser. In HTML-form, and with the help of some JavaScript, the result data could be ordered by the different features or filtered according to the amount of dependencies.

4.3 Other uses for the toolset

We also studied the use of other concern operations in the context of this case study. The studied scenarios include tracking the evolution of the concerns in the ISA model, as well as studying how different product configurations relate to one another.

4.3.1 Tracking evolution of ISA models

The dependency model received from Nokia was based on a specific weekly build of ISA. To elaborate the case study, we were able to get access to another model based on newer build of ISA. While it is outside the scope of the original boundaries of the case study, this offered an opportunity to test the version tracking aspect of the toolset. Unfortunately, due to the fact that the complete result data of the original case study was presented only in XML/HTML-format, the version tracking could only be done to single header files at a time instead of the entire studied data. A similar, customized batch operation could have been implemented for studying the evolution of the different dependencies, as was done with the original case study. However, it was decided that this did not fit into the scope of our study and thus the idea was discarded.

What was learned was that, indeed, transition from an earlier version of the model to a newer one was easier by using the version tracking tool. After the individual concerns, bound earlier to the old version of the model, were remapped to the new version, the user only needed to go through the concerns one by one, delete the elements that had not been bound and add those thought to belong to this concern in the new version of the model. Of course, concerns that had been created as a result of thousands of resulting concerns, the results of the batch run would be presented in a more convenient format instead of the result patterns in the INARI tool. It was decided that browsing the results in INARI would be too slow, both usability and performance-wise.

Instead, an XML-based result format was developed, allowing only the essential parts of the resulting data to be presented. For each header file, we would only present information on the name of the file, the amount of modules dependent on it, the topmost compilation unit containing those models, and the location of the dependent modules in the compile unit hierarchy. This XML-data was then transformed into HTML using XSL transformation to allow the analysis of the results using any Web browser. In HTML-form, and with the help of some JavaScript, the result data could be ordered by the different features or filtered according to the amount of dependencies.
queries can usually not be remapped so easily, because the original query result was for the older version of the model. In such a case, to get a valid concern one must first remap the original, fundamental concerns that had been used as parameters for the query and only then re-execute the query. This is the only way to be absolutely certain that the contents are in accordance to what was expected.

One specific evolutionary feature was changes in the global header file repository. Since the dependency analysis was done for both of the versions of the dependency model, we already had a concern mapping containing all the header file elements from both of the models. To examine the evolutionary change in the model, the version tracking operation was run to the header concern in the later version in regards to the older version’s model (i.e. the header elements of the later version were remapped to the older version). From the result it was possible to determine what header files had been added to the model between the two versions, as those elements understandably could not be bound to the elements found in the older version. The operation could have also been executed in the opposite direction, in which case we could have learned which header files had been dropped from the system between the two versions.

4.3.2 Mapping product configurations and features

Another place where concerns can be used is in the mapping of the different product configurations in the model. As was previously mentioned, no single product contains the entire set of modules ISA has to offer. Defining each configuration as its own concern would allow us to study how the different configurations overlap and relate to one another. Also, by using the version tracking tool, we could very easily see if there is a problem with one of the product configurations due to the evolution of the system: elements of a certain product configuration concern could not be entirely mapped to the new version if some of the elements were missing.

As a practical example, let us assume there exists a concern mapping that maps all the required modules to build a certain product, hereafter denoted as product A. After the underlying platform evolves, we run the version tracking tool for product A in respect to the new version of the model. The resulting concern mapping shows us whether or not the artefacts that were earlier needed to build a working product A are still present in the model. Missing artefacts may mean that a foul-up was made, however, they may also simply mean that some artefacts have been combined, but the contents still exist. Nevertheless, the original product configuration is clearly no longer valid.

This particular scenario somewhat overlaps the one presented in the previous section. The premise is that, instead of products, we identify functional features (e.g. the calendar) and create concern mappings for those. After we have done this, we can create any combination of features to form a new product configuration. By using the union operation we simply combine all the different feature concerns into a composition concern that is bound to all the necessary artefacts for that configuration.

The usefulness of this approach is limited by the fact that most of the features may already be split into their separate modules and are therefore already mapped. However, whether this is the case is indistinguishable from the models. The models are clearly heavily modularized, but whether a single module equates to a single feature or service, we do not know. If it were commonplace that the features would cross-cut different modules, this approach could be seen useful, as it would define a clear boundary for a specific feature.

5 Related work and discussion

Reverse and re-engineering needs and projects have motivated the development of various tools that support managing program comprehension tasks. Our approach relates to those supporting feature identification and management, software querying, and understanding evolutionary changes in software.

In [15] Poshycanyk et al. propose an approach to support feature location, combining two existing techniques: one based on Latent Semantic Indexing (LSI) of the source code[13] and the other based on scenario-based probabilistic ranking of events appearing in an execution trace[2]. In their approach the feature location problem is considered as a decision-making problem in the presence of uncertainty[15]. In [5] another feature identification approach is proposed. While these approaches focus feature identification, we focus on feature analysis and maintenance. Therefore, the sources for queries also differ: in [15,5] features are located from the source code and run-time execution trace, while in our approach the user can manually construct the concerns or by querying the reverse engineered UML model. Therefore, our concern query approach combines bottom-up and top-down software analysis principles. One of our future research directions is to combine our approach with pattern recognition techniques. That would allow us to increase the degree of automation, if desired, in the construction of the initial concerns.

Tool support for detecting evolutionary changes in software is supported e.g. by CodeCrawler[12]. CodeCrawler is a software visualization tool that has been used e.g. to visualize the evolutionary changes of class hierarchies[6]. While CodeCrawler is rather a general purpose visualization tool, we focus on querying
and analysis of specific concerns. Further, in CodeCrawler the visualization is constructed based on calculations of certain product metrics from the source code, the visualization thus representing the output of the calculations or queries. In our approach, the model constructed is rather considered as a source for concern queries. Currently, the concerns are defined in a tool-assisted way either manually or semi-automatically e.g. based on naming conventions. Support for metrics could also be added to our approach to e.g. support the analysis of the concerns. Also, metrics could be combined with pattern recognition approaches, which often calculate probabilities to estimate the level of confidence, to enhance the concern location capabilities.

Support for visualizing software evolution and especially file evolution, is proposed by Voinea and Telea in [120]. The approach aims at mining evolution histories in large-scale software projects. As our approach, the approach proposed by Voinea and Telea is interactive for directing the visualization towards selected goals. While their approach visualizes the projects under analysis at the granularity level of files, comparably to CVS systems, with our approach the software engineer can analyze the evolitional changes of a software project at the level of reverse engineered UML models and at the level of concerns.

SEXTANT[4], in turn, is a query-based tool to explore and visualize software systems. SEXTANT combines top-down querying and bottom-up navigation strategies that work across different kinds of software artifacts. In our approach, too, top-down querying is combined with bottom-up reverse engineering techniques. Currently, we only support for UML models as subjects for queries. However, one of our future research directions is to support querying on heterogenous concerns. In [8] the concept and tool support for heterogenous patterns is proposed. A heterogenous pattern can contain artifacts from different sources, such as code, UML, and configuration or documentation files given in XML. Combining the concern querying approach with the support for heterogenous patterns, more covering queries can be made. For instance, one can find out cases where a code change is made to a concern, but the corresponding documentation has not been updated.

Acknowledgements

This research has been financially supported by Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, TietoEnator, Plenware, and John Deere.

References


