Concern Based Mining of Heterogeneous Software Repositories

Imed Hammouda
Tampere University of Technology
Institute of Software Systems
P.O. Box 553
FI 33101 Tampere, Finland
imed.hammouda@tut.

Kai Koskimies
Tampere University of Technology
Institute of Software Systems
P.O. Box 553
FI 33101 Tampere, Finland
kai.koskimies@tut.

ABSTRACT

In the current trend of software engineering, software systems are viewed as clusters of overlapping structures representing various concerns, covering heterogeneous artifacts like models, code, resource files etc. In those cases, adequate search mechanisms for software repositories should be based on such fragmented nature of software systems, allowing concern-oriented queries on the system data. For this purpose, we propose a conceptual framework for a concern-oriented query language for software repositories. A pattern-based implementation scheme is discussed, exploiting existing tools. The applicability of the approach is studied in the context of an industrial case study.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Miscellaneous

General Terms

Documentation

Keywords

Heterogeneous software repositories, concern-based mining, pattern-based search structures

1. INTRODUCTION

Software repositories have traditionally been understood as collections of source artifacts that consist of either monolithic source text [15] or higher-level data structures storing the essential information contained by the source [5]. In both cases, the information in the repository is organized according to the structures and semantics of the underlying programming language. Using this kind of organization creates a gap between the repository and the needs of various stakeholders: it is often difficult to express high-level information needs in terms of the low-level concepts provided by the repository.

For example, a maintainer may be interested to identify those parts of the system that are related to the persistency issues concerning the copy-and-paste feature of the system. Mapping such information request to the source structures is virtually impossible without substantial additional knowledge about the ways copy-and-paste and persistency are implemented in the system, no matter how detailed information about the source is stored in the repository.

We argue that there should be a convenient mechanism that allows us to incorporate information about the anticipated concerns of the system stakeholders into the software repository, and that the search engine should exploit this information. In this way the high-level requests of the stakeholders can be mapped to the low-level structures of the source. The required information is inherently external: it cannot be inferred from the system with any automatic means. This is in contrast to traditional techniques such as adding special comments to code or informal notes to UML diagrams [17]. However, the results of queries may introduce new information which can be cumulated in the repository.

Another major requirement for a software repository is its ability to support heterogeneous software artifacts. Many stakeholders are not interested in source code only, but in requirements and feature models, in architectural and design models, in scripts, resource files etc. Most concerns are related to several artifact types expressed in different languages and notations. Thus, queries on the software repository should yield results that span heterogeneous artifacts.

In this paper, we propose a concern-based organization of a software repository, with an external structure representing the relevant concerns. The result of a query is always a concern as well, possibly added to the set of persistent concerns. The actual software artifacts are not touched, but only referenced in the external structure. The overall approach is depicted in Figure 1.

The technical solution for representing concern structures may vary. In this paper, we investigate the possibility to use a pattern-based approach, which has certain advantages. In particular, this approach lends itself to capturing software elements in heterogeneous artifacts, fast concern manipulation, overlapping concerns, and existing tool support. However, the basic idea could be realized using other techniques, too.

The resulting concern-based information about the software system can be displayed and exploited in various ways.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro t or commercial advantage and that copies bear this notice and the full citation on the rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior speci c permission and/or a fee.
MSR'06, May 22–23, 2006, Shanghai, China.
Copyright 2006 ACM 1 59593 085 X/06/0005 ...$5.00.

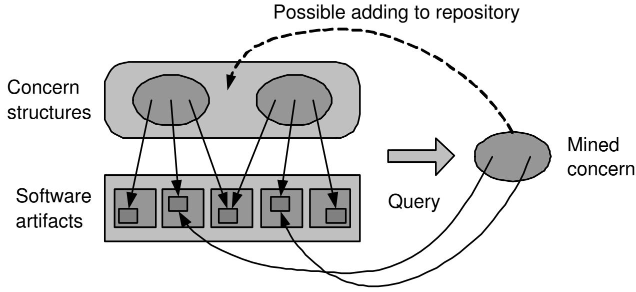

Figure 1: Concern-oriented repository mining.

We assume that the concerns (including the concerns resulting from the queries) can be traced and visualized by a tool, allowing the construction of various concern-based views on the system.

We will proceed as follows. In the next section we discuss the multidimensional nature of software systems and the use of concerns as the basic unit of queries. In Section 3 we briefly explain the concept of a pattern and how it can be used to represent concerns in a software repository. Section 4 briefly discusses tool support for the pattern-based mechanism. The approach is demonstrated in the case of an industrial system in Section 5, and some concluding remarks are presented Section 6.

2. DECOMPOSITION OF SOFTWARE AR TIFACTS

We assume that the artifacts of a software system consist of sets of artifact elements. The elements of a set have the same type, e.g model or code elements, but the sets are heterogeneous. Each set is associated with a specific artifact type and addresses certain concerns in the software system. From the concern point of view, a set can address one or many concerns. For instance, given an MVC-based (Model View Controller [13]) implementation of an arbitrary software system, the code representation of the architectural style may address several functional features of the system. Thus, from the architectural point of view the MVC part of the system addresses one concern, but from the functional point of view several concerns are involved.

This decomposition scheme is discussed in Figure 2. The top cubical structure illustrates the decomposition of a system based on its artifact types. Each top cube represents a set that groups together artifact elements that are of the same type. Each of these sets can be further decomposed into smaller sets based on the concerns addressed in the artifact elements of that set.

The fragmented nature of software systems is naturally reflected in the process of mining software repositories as the goal of any repository search operation is to retrieve the elements of that repository that address a certain matter of interest. Furthermore, the search operations themselves could be based on the outcome and the combination of other search operations. To give an example, system maintainers are usually interested in those parts of the system data that are relevant to their actual maintenance needs. As the context of maintenance changes during the system evolution process, new fragments of the system, in addition to the current ones, might become relevant. The natural relationship between system fragments and the concerns they treat advocates the need for a concern-oriented query language. As the fragments represent heterogeneous sets of repository elements, we can express the elements and the operations of this language in terms of set theory [7] concepts and terminology.

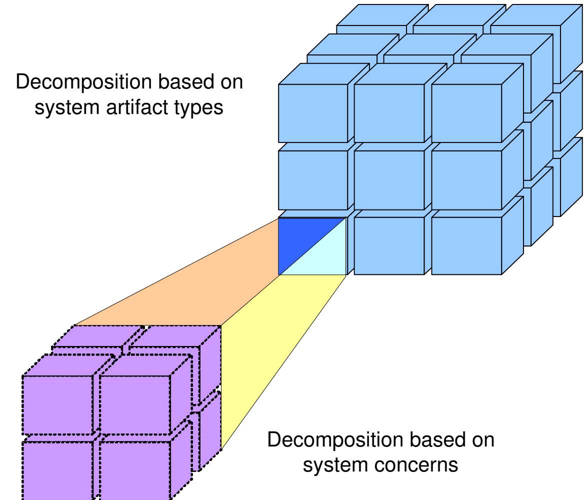

Figure 2: Decomposition of a software system.

In the context of concern-based decomposition of software repositories, we can define a set (S) as an unordered group of artifact elements (E0, E1, E2 ...) with no duplicates. It is possible, however, to decompose sets of repository elements into smaller sets (S = S0 + S1 + S2 ...). Considering the example given earlier, the MVC-based source code can be decomposed into smaller sets of code elements, each representing a certain functional feature. During the decomposition process, the same artifact element may be placed into different sets. This is the case for elements that address multiple concerns.

There are two special cases of sets: the empty ($\emptyset$) and the universal set (U). For software systems, the universal set is a heterogeneous collection of elements that represent all system artifacts. No concern-based logical system decomposition, however, can lead to sets that are empty. Empty sets can only be results of arbitrary search operations performed on the concern-based decomposition of the system.

Based on the above discussion, let us consider two arbitrary system concerns C1 and C2 and two sets S1 and S2 containing the artifact elements addressing concerns C1 and C2, respectively. We can define different concern-based queries on C1 and C2. We assume that sensible concern-based queries can be formulated using the classical set operations: union, intersection, complement, and difference. While other set operations (like Cartesian product) could be realizable as well, it seems that they are less useful in practice for concern-based queries.

- *Concern merging.* The merging of the two concerns C1 and C2 can be expressed as the union of the two sets S1 and S2, which is the set S containing all the elements in either S1 or S2. The elements of S address either concern C1 or concern C2. This operation is denoted using the '+' symbol: C = C1 + C2.

- *Concern overlapping.* The overlapping of the two concerns C1 and C2 can be expressed as the intersection of the two sets S1 and S2, which is the set S containing all the elements that are in both S1 and S2. The elements of S address both concerns C1 and C2. This operation is denoted using the '&' symbol: $C = C1 \ \& \ C2$.

- *Concern slicing.* The slicing of the concern C1 with respect to the concern C2 can be expressed as the difference of the two sets S1 and S2, which is the set S containing all the elements that are in S1 but not in S2. The elements of S address concern C1 but not concern C2. This operation is denoted using the '-' symbol: C = C1 - C2.

- *Concern exclusion.* The exclusion of concern C1 can expressed as the complement of the set S1, which is the set S containing all the elements in the universal set U except those in S1. The elements of S address all other concerns except concern C1. This operation is denoted using the '^{c}' symbol: $C = C1^c$.

To give an example, consider a software system with a repository containing models, programs, resource files, project documentation, etc. The software system provides different security strategies and requires user authentication for many of the functionality it implements. We can, therefore, identify two system concerns: security (say Sec) and user authentication (say Auth). In the context of those two system concerns, the four above concern operations can be viewed as searching for all repository elements addressing security or user authentication (merging, Sec + Auth), elements representing user authentication security (overlapping, Sec & Auth), elements corresponding to all security aspects except those for user authentication (slicing, Sec - Auth), and elements addressing all other concerns except security strategies (exclusion, Sec^c).

3. PATTERNS AS SEARCH STRUCTURES

As a tool infrastructure for concern-based querying, we use the concept of aspectual patterns [10] to represent arbitrary concerns in software repositories. Figure 3 depicts a conceptual model in UML for aspectual patterns. A pattern is a collection of hierarchically organized roles rather than concrete repository elements. A pattern is used to collect together all repository elements that address a certain system concern. This is done by binding pattern roles to certain elements of the repository representing that concern. Each role can be associated with a set of properties that can be used for the search operations. An example property could be a textual description tag explaining the purpose of the role binding. Another example property is the type of the role. The role type determines the kind of repository elements that are bound to the role, e.g. model elements (UML elements), source code (Java elements), resource data (binary documents), project documentation (text file and file fragments), etc.

A concern-based decomposition of a software repository partitions that repository into a number of fragments. At the elementary level, each fragment corresponds to exactly one system concern. Each of these fragments is represented by a separate pattern.

The elements of a fragment may refer to or depend on each others. In many cases, the relationship between two

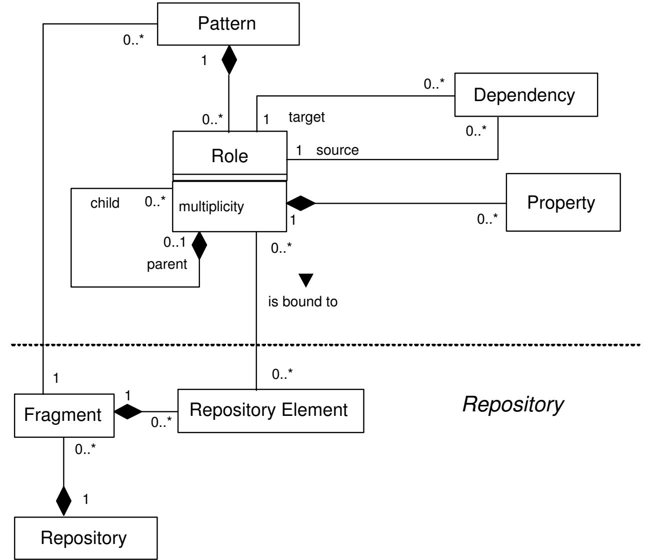

Figure 3: Conceptual model for aspectual patterns.

elements is implicit and cannot be marked in the fragment. For example, it might be hard to express that a binary file is a resource for a certain implementation class. Such a relationship can be easily made explicit in the pattern using role dependencies.

It is possible to bind the same pattern role to multiple repository elements. This is the case if multiple elements play that specific role. The possibility to bind a role to multiple elements (and the optionality of certain roles) is indicated using the multiplicity attribute of the role. For example, if a role has multiplicity [0..1], the role is optional in the pattern (and thus in the concern).

Figure 4 illustrates how patterns are bound to repository elements. There are two patterns X and Y. Each pattern represents a separate system concern and is associated with a repository fragment. Each fragment contains the elements that address the corresponding concern. The fragments do not have to be aligned with the physical or logical distribution of the repository. In Figure 4, the fragment that is associated with pattern Y contains elements cutting across different packages. Furthermore, the example fragments are overlapping, there is a repository element that addresses both concerns. Roles r2 and r3 are bound to the same element while r7 represents the role of two different elements. The repository may contain elements that are not referred by the patterns. Such elements do not correspond to any anticipated concern and do not manifest in the repository mining process. Similarly, patterns may have unbound roles. Such roles may represent possible extension points of the corresponding fragments.

Using role-based pattern structures, the concern operations we identified in the previous section can be realized in a straightforward way.

- *Merging operation.* The merging of the two patterns X and Y returns a new pattern with roles bound to all repository elements referred by either X or Y. In the example case, the resulting pattern references elements of both fragments (six elements).

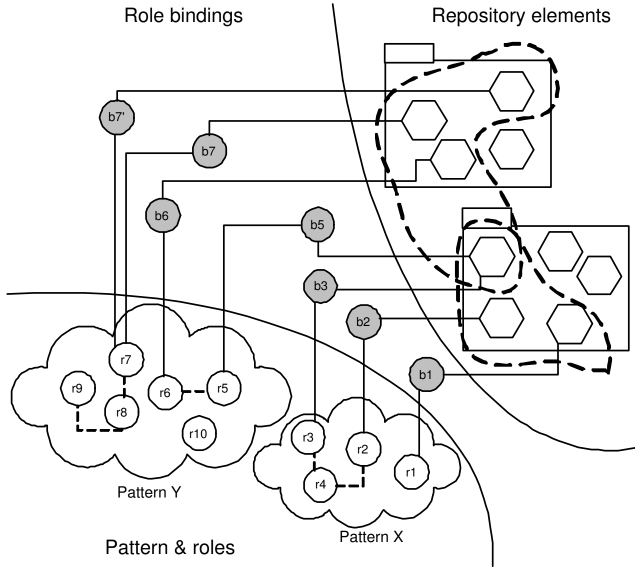

Figure 4: Aspectual patterns as search structures.

- *Overlapping operation.* The overlapping of the two patterns X and Y returns a new pattern with roles bound to all repository elements referenced by both patterns. In the example case, the resulting pattern references one element (corresponding to binding b3 and b5).

- *Slicing operation.* The slicing of pattern Y with respect to pattern X returns a new pattern with roles referring to repository elements bound to roles of Y but not to roles of X. In the example case, the resulting pattern references three elements (corresponding to bindings b6, b7 and b7').

- *Exclusion operation.* The exclusion of pattern Y returns a new pattern referring to all elements not bound to the roles of Y. In the example case, the resulting pattern references two elements (corresponding to bindings b1 and b2). Note that the universal set represents only the bound repository elements, the other unbound elements (three elements) are not considered for any concern queries.

In addition to the above search mechanisms, patterns can be exploited to provide support for other ways of mining tasks. This is achieved by using the properties attached to pattern roles or by accessing the properties of the referenced repository elements themselves. For instance, we might be interested in retrieving repository elements of certain types or simply those elements whose role properties match certain criteria.

As explained earlier, the approach discussed in this paper assumes that the repository comes with an initial annotation that represents an original concern-based clustering of the repository elements. Each cluster addresses a specific system concern and is therefore represented by a separate pattern. During the mining process, the repository is explored by visiting the appropriate patterns, following role bindings (to the repository elements), and investigating the properties of the pattern roles. Based on the original patterns, new clusters corresponding to other concerns (and concern combinations) can be constructed by applying the search mechanisms discussed earlier. As the search results are in fact returned as new patterns, these new patterns could be added to the original annotation and might themselves be used in future mining operations. In the case study section, we will give a concrete example of such a scenario.

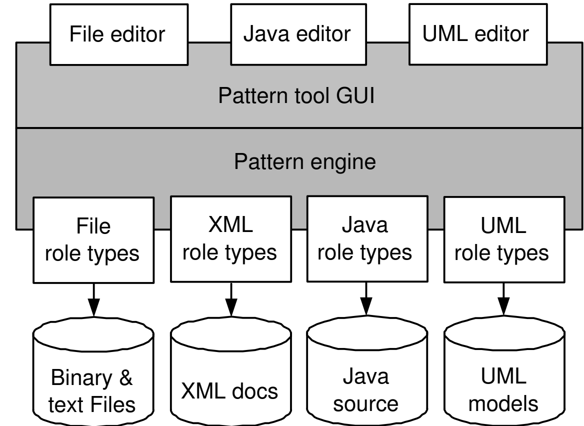

Figure 5: Architecture of the MADE environment.

4. TOOL SUPPORT MADE

In order to demonstrate the pattern-based approach for representing concern structures, we use an Eclipse-based [6] pattern-driven development environment called MADE (Modeling and Architecting Development Environment [11]). Figure 5 depicts a layered architecture of the MADE tool environment. The pattern engine represents the core component of the platform. It is used to manage the binding process and is thus independent of any artifact types. Currently, the environment provides support for binding pattern roles to UML, Java, XML, and general file (binary and text) elements.

The pattern tool GUI provides different views and wizards for creating new patterns, adding roles to patterns, binding roles to repository elements, viewing role bindings, and tracing bindings to their corresponding repository elements. When a bound element is retrieved, it can be viewed in its own editor. Currently Rational Rose is used as the UML editor. Therefore, only Rose-based UML models can be browsed. For Java, XML, and general file content, Eclipse-built-in editors are used.

Furthermore, the MADE platform comes with mechanisms for browsing and highlighting patterns (concerns) in the repository data. The biggest limitation of the tool, however, is that search operations, as presented in this paper, are not currently implemented. We are planning to add such support in future releases of the platform. Nevertheless, it is still possible and beneficial to use the tool in its current state to construct an original concern-based annotation of a repository and present that as a documentation tool for system learners. Another limitation of the tool is that MADE pattern roles are strongly typed. In order to support other repository element kinds (such as other programming language elements), the tool has to be extended with new role types.

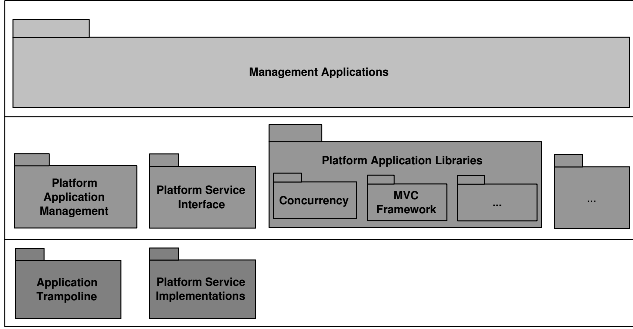

Figure 6: Architecture of Nokia GUI platform.

5. CASE STUDY

Network management represents one of the core businesses of Nokia. For managing networks and network elements, the company produces a family of NMS (Network Management System) and EM (Element Manager) applications. The GUI parts of the applications are developed based on a common platform. The main purpose of the platform is to help developers to build Java-based GUIs and to make sure that NMS and EM applications share the same features. The GUI platform has been developed as an object-oriented Java framework. New management applications are constructed by specializing parts of the framework and using the common services offered by the platform.

Figure 6 depicts a logical decomposition of part of the GUI platform consisting of several independent logical blocks. The upper layer (Management Applications) stands for the NMS and EM applications. They are not part of the platform but they are built on top of it. The middle layer shows a number of platform subsystems:

- Platform Application Management. The purpose of this subsystem is to keep track of running applications and to manage the starting of new applications.
- Platform Service Interface. This component defines the abstract interfaces to the services offered by the GUI platform. Example services include logging, authentication, and online help facilities
- Platform Application Libraries. This subsystem provides common features used for building new network management applications. Example features include concurrency control and an MVC-based framework

The bottom layer shows a block representing Platform Service Implementations. The GUI platform comes with a number of default service implementations. If needed, application developers can provide their own service implementations based on the service interfaces. The Application Trampoline component represents an external interface to Application Management for allowing processes outside the virtual machine to start new applications in the same virtual machine.

The system repository for the platform comes with a wide range of artifact types including design models, source code, executable jars, property files, deployment descriptors, user manuals, technical development documents, managerial presentations, etc. In addition to platform artifacts, the repository also contains example applications built on top of the platform.

Table 1: Example platform concerns

Concern Category	Concerns	Artifact types
Component concerns	Application Management (AppMan), Application Trampoline (AppTram)	Jars, UML design models, Java source files, documents
Feature concerns	Authentication (Auth), Online help (OnlineHelp)	Jars, UML feature and design models, Java source files, XML descriptors, documents
Architectural concerns	MVC (MVC), Layered architecture (layered)	UML architectural and design models, Java source files, documents
Maintenance concerns	Adding alternative service implementation (AltServImp), Modifying service interfaces (ModServInt)	UML design models, Java source code, documents
Specialization concerns	Feature specialization (FeatSpec), GUI specialization (GUISpec)	UML design models, Java source code, XML descriptors, documents
Global concerns	Persistency (Pers), Security (Sec)	UML design models, Java source code, documents

As an original annotation of the repository, we have identified a number of system concerns of different categories. By concern categories, we mean groups of related system concerns representing similar matters of stakeholders' interests. Table 1 depicts six concern categories. For each concern category, we give two example concerns. For instance, from the viewpoint of system components, one can identify a concern standing for application management and another representing application trampoline. Each of these system concerns is represented using a separate aspectual pattern. The name of the pattern is given next to the concern definition. In addition, the table gives the artifact types where each concern is represented.

In order to illustrate our concern-based query language, Table 2 shows four example concern queries based on the concerns (patterns) identified in Table 1. The first query stands for concern slicing. It is for excluding all repository elements that address the authentication concern (Auth) and are at the same time related to system security (Sec). The search results of this query are themselves returned as a pattern (NonSecAuth). The latter pattern is then used in the second query to narrow down the search results further to those elements that are also addressing application trampoline. The third query gives an example of a concern merging operation and the fourth shows a second illustration of the merging operation.

Table 2: Example concern queries

Stakeholder interest	Concern expression
Are there any parts in authentication that are not related to security?	NonSecAuth = Auth - Sec
Are any of the parts above in the application trampoline component?	NonSecAuth & AppTram
Show me the application management component in the context of the MVC architecture!	MVC + AppMan
What are the parts of the platform that are relevant for creating online help for an application?	FeatSpec & OnlineHelp

In some situations, the user might not be interested in viewing all the artifact types corresponding to her concern query. For instance, designers may prefer to analyze system repositories based on detailed design diagrams. In the context of tool support, it is useful to allow users to refine the query results based on their desired artifact types.

6. DISCUSSION

In this paper, we have argued that in the current trend of software engineering, mining mechanisms are driven by the heterogeneous and fragmented nature of software systems. Such a viewpoint has already been taken in several research works in the field of mining software repositories [4, 16]. Supporting such a viewpoint, we have presented a conceptual framework for a concern-oriented query language. Our approach differs from other querying models, such as the one presented in [12], by treating system concerns as first class citizens. The concerns, grouped according to various concern categories, address different matters of stakeholders' interests.

Concern-oriented mining has been widely discussed in the field of aspect-oriented software development (AOSD [9]) and is generally referred to as concern elaboration. There are many tools that can be used for concern elaboration. These include standalone search tools, browsers integrated in IDEs, and compilers. A typical scenario, using these tools, is to run a query over a model to retrieve the model elements addressing a certain concern. Similar to our approach, the queries could be refined once the results are evaluated.

In [14], three concern elaboration tools have been evaluated namely, AspectBrowser [1], AMT [2], and FEAT [8]. These tools differ in two major ways: how programs models are represented before being elaborated and how search results are presented to the user. Instead of relying solely on repository data for extracting the required information, our approach is based on mining a search space that is external to the repository. In our methodology, the search space is not deduced from the repository but rather superimposed on the repository data. This is in contrast to other approaches (i.e. reverse engineering [3]) that are based on building high level representations of low level system data [5]. It can be argued that the two approaches are in fact complementary.

As an implementation scheme, we have used a role-based pattern mechanism for representing system concerns and formulating the queries. Using patterns we could successfully annotate a repository taken from the industry and we could conveniently express our search queries. As an experimental environment for concern-based mining, we have exploited an existing pattern-driven development environment known as MADE. The MADE environment comes with various capabilities for concern-based annotation of software repositories. However, search mechanisms need still to be implemented in the tool.

7. ACKNOWLEDGMENTS

This research has been financially supported by the National Technology Agency of Finland (project Inari), Nokia, Plenware Group, TietoEnator, and John Deere.

8. REFERENCES

[1] Aspect Browser WWW site. Available at `http://www-cse.ucsd.edu/users/wgg/Software/AB/`, 2006.

[2] Aspect Mining Tool (AMT) WWW site. Available at `http://www.cs.ubc.ca/~jan/amt/`, 2006.

[3] E. Chikofsky and J. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[4] A. Dekhtyar, J. H. Hayes, and T. Menzies. Text is software too. In *Proc. MSR 2004*, pages 22–26, Edinburgh, Scotland, UK, 2004.

[5] S. Demeyer, S. Tichelaar, and S. Ducasse. Famix 2.1 - the famoos information exchange model. Technical report, University of Berne, 2001.

[6] Eclipse WWW site. Available at `http://www.eclipse.org`, 2006.

[7] H. B. Enderton. *Elements of Set Theory.* Academic Press, 1977.

[8] Feature Exploration Tool (FEAT) WWW site. Available at `http://www.cs.ubc.ca/labs/spl/projects/feat/`, 2006.

[9] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, 2004.

[10] I. Hammouda. A tool infrastructure for model-driven development using aspectual patterns. In S. Beydeda, M. Book, and V. Gruhn, editors, *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, pages 139–178. Springer, 2005.

[11] I. Hammouda, J. Koskinen, M. Pussinen, M. Katara, and T. Mikkonen. Adaptable concern-based framework specialization in UML. In *Proc. ASE 2004*, pages 78–87, Linz, Austria, 2004.

[12] A. Hindle and D. M. German. SCQL: a formal model and a query language for source control repositories. In *Proc. MSR 2005*, pages 100–104, Saint Louis, Missouri, USA, 2005.

[13] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in the

smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[14] G. Murphy, W. Griswold, M. Robillard, J. Hannemann, and W. Wesley Leong. Design recommendations for concern elaboration tools. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 507–530. Addison-Wesley, 2004.

[15] Rational ClearCase WWW site. Available at `http://www-306.ibm.com/software/awdtools/clearcase/`, 2006.

[16] G. Robles and J. M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. In *Proc. MSR 2005*, pages 106–110, Saint Louis, Missouri, USA, 2005.

[17] Unified Modeling Language WWW site. Available at `http://www.uml.org/`, 2006.