A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction

Günter Kniesel
Institut für Informatik III
Universität Bonn
Bonn, Germany
gk@cs.uni-bonn.de

Jan Hannemann
Department of Graphics and Computer Science
University of Tokyo
Tokyo, Japan
jan@graco.c.u-tokyo.ac.jp

Tobias Rho
Institut für Informatik III
Universität Bonn
Bonn, Germany
rho@cs.uni-bonn.de

ABSTRACT

In this paper we evaluate logic code analysis and transformation frameworks for their suitability as basic infrastructures for fast detection and extraction of (crosscutting) concerns. Using design patterns as example concerns, we identify desirable properties that an infrastructure should fulfill. We then report our initial results of evaluating candidate systems with respect to these properties. We show how high precision design pattern detectors can be easily formulated as predicates that are evaluated in mere seconds even on the sources of large software systems, such as the Eclipse IDE. Although details still remain to be analyzed further, our current results suggest that the pair JTransformer & CTC is a good candidate for a general infrastructure, combining very good querying performance, scalability and short turnaround times with a seamless integration of querying and transformation capabilities.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement —*Restructuring, reverse engineering, and reengineering*

Keywords

Logic code analysis and transformation, concern mining, design pattern detection, JTransformer, CTC, JQuery, CodeQuest.

1. INTRODUCTION

Design pattern detection is an important problem for program understanding, design recovery, and reverse engineering of legacy systems [23, 11, 1, 25].

AOSD provides a novel motivation for design pattern detection. Unlike traditional object-oriented programming, aspects can express the structure and collaborations of a design pattern as a reusable software module [9, 6]. This ability opens the way for a new reengineering option: refactoring patterns to aspects. It comprises the behavior-preserving extraction of different instances of a design pattern's implementation scattered across an application and their replacement by a joint implementation in a single "aspect" module [13], as shown in previous work [10]. Refactoring of patterns into aspects requires a precise detection of the elements that should be refactored and identification of the roles they play in the detected pattern's collaboration.

In this paper we focus on the challenge of design pattern detection that is precise enough to serve as input to a subsequent automated refactoring to aspects. In this context, we do *not* (yet) propose novel detection techniques but are aiming, as a first step, at identifying a suitable basis for the development, joint experimentation, comparison and integration of various detection technologies. Thus we address the issue of a versatile and programmer-friendly infrastructure that we consider to be an essential prerequisite for being able to develop a comprehensive pattern detection system that can cope with the high number of relevant patterns, the complexity of individual patterns and the high number of variations that must often be captured.

The infrastructure evaluation described in this paper is motivated by our dissatisfaction with different object-oriented program representation, analysis and manipulation frameworks that we had used previously. In particular, we used BCEL [3] for the byte code analysis performed at load-time by JMangler [18, 17], Recoder [20] as the basis of the refactoring infrastructure JConditioner [19] and evaluated the Eclipse JDT's search facility [27] for its applicability as a general Java source code analysis infrastructure. However, our main critique is not specific to these systems but common to all object-oriented approaches. It addresses primarily the slow turnaround time in developing detectors, the relatively low-level programming style, and the inability to address performance issues without recoding the detectors.

An alternative to object-oriented detection methods are logic-based approaches, which have demonstrated capabilities to quickly specify, execute and refine complex queries

In Proc. of *AOSD 2007 Workshop on Linking Aspect Technology and Evolution (LATE'07)*, Updated version to appear in ACM Digital Library.

over code bases [12, 7, 22]. However, the experiments recently reported by Hajiyev et al. [7] show that significant differences exist between logic-based approaches regarding their efficiency and scalability. This motivated our extended evaluation of available systems reported in this paper. Like [7], our comparison addresses efficiency and scalability issues but additionally includes further criteria that are relevant for aspect identification and extraction, and applies them also to systems disregarded in [7]. In particular, it includes JTransformer [30], a system that combines code analysis and transformation capabilities for Java and that we found to be better suited than similar systems.

The remainder of the paper is organized as follows. Section 2 identifies desirable properties of an infrastructure for concern identification and extraction. Section 3 introduces the JTransformer system. Section 5 describes its use for design pattern detection. Section 6 evaluates its suitability with respect to the stated challenges. In section 7 the individual approaches are compared. Section 8 concludes.

2. DESIRABLE PROPERTIES

In this section we summarize desirable properties for a practically useful infrastructure for concern mining and extraction that guided us in our evaluation:

Expressiveness A generally useful infrastructure should not limit the analyzes and transformations that can be expressed for a particular language. Whereas individual tools built on top of the infrastructure might not need the full expressiveness, the infrastructure should support implementation of any kind of detection or extraction functionality.

Turnaround Finding the best techniques for concern mining and extraction is still a wide open problem that will remain an area of active research for many years. Thus it is essential that developers can quickly implement many different detection techniques, run the corresponding detectors efficiently on the code to be analyzed, and seamlessly use the analysis results for subsequent extraction. The length of this cycle is typically dominated by the time needed to develop detectors and transformers. Thus the infrastructure should support an abstraction level that fosters rapid development without limiting expressiveness.

Integration Concern detection is typically followed by concern extraction. Ideally, developers should not need to pass analysis results for extraction to another system since implementing the respective converters slows down the development and executing them at run-time reduces overall performance. Thus the ideal infrastructure should support a seamless integration of analysis and transformation capabilities as a crucial prerequisite for fast turnaround and productivity.

Performance Concern mining and extraction is typically an interactive process, driven by the programmer's semantic knowledge that is beyond the reach of automated tools. In such settings, individual analyzes must be very fast (milliseconds to seconds) in order to be useful. Longer waiting times disrupt the work-flow and hinder productivity.

Scalability Scalability means that performance in the order of seconds must be achieved for individual analyzes even on software systems consisting of tens of thousands of classes, such as the Eclipse platform and its Java Development Tool [28].

Multi-Project support Projects of the size of Eclipse are typically not a single monolithic system but split into many subprojects. The exploration and mining infrastructure must support such settings, including analyzes and transformations that span multiple projects.

Availability We concentrated our survey on generally available, supported tools. We deliberately disregarded commercial tools and tools that are only described in papers but are not available or whose development and support has been discontinued. Availability includes the availability of downloadable software versions and of related documentation.

As explained in the introduction, the fast turnaround time and the option to tackle performance issues by automated optimizations of declarative code motivated our investigation of logic-based approaches. With respect to the availability requirement, only three logic-based tools entered our candidate list: JQuery[12], JTransformer [30] and SOUL [22]. In addition, we included CodeQuest [7] as a reference for performance and scalability, although the tool is not publicly available since it is currently being developed into a commercial product.

With respect to the integration requirement the three prime candidates support subsequent transformation of query results (CodeQuest does not). However, the transformation support offered by JQuery and SOUL is limited (see Section 6.3).

JQuery and JTransformer are Eclipse plug-ins, neatly integrated with the Eclipse Java development Tool. SOUL is implemented in a Smalltalk environment. Due to the "Irish" Eclipse plugin, developed by Johan Fabry and Oscar Andrés López P. (see http://prog.vub.ac.be/˜jfabry/irish/) it is possible to pipe information from a Java 1.3 project in Eclipse to the Smalltalk environment and back. Still, for using SOUL, Eclipse programmers would be forced to perform coding in one environment and analysis in another. In order to work effectively, most Java programmers would additionally need to learn a fair bit about Smalltalk and its IDE.

Since, for a start, we were mostly interested in environments that can readily be used by Java programmers, our comparison has focused on JQuery, JTransformer and CodeQuest, the tools integrating Java programming with logic-based analysis of Java programs within Eclipse. The next section introduces the logic-based representation employed by these tools.

3. LOGIC-BASED REPRESENTATION OF JAVA PROGRAMS

JTransformer, JQuery and CodeQuest implement Eclipse plugins that generate a representation of a Java program as logic facts. JTransformer uses the platform-independent, free SWI-Prolog implementation for storing, analyzing and transforming factbases. JQuery uses the TyRuBa logic engine developed by Kris de Volder. CodeQuest stores the facts in a relational database and includes its own query engine that is able to perform a bottom up evaluation of Datalog rules based on the database contents.

JQuery and the version of CodeQuest described in [7] confine themselves to the representation of interface elements (types, methods, fields) and their relationships (contains, inherits, calls, accesses). In contrast, JTransformer represents the complete program, including the complete abstract syntax tree (AST) of method bodies.

Because such a complete program representation is required in general, as explained in Section 5.1, we use the JTransformer representation for our design pattern detectors. We briefly introduce as much of this representation as necessary for the examples presented in this paper. For readers who are not familiar with logic programming terms we also mention their relational database counterparts (in braces).

JTransformer represents each AST node as a logic fact (i.e., database tuple). The predicate symbol (relation name) represents the node type. The first parameter (attribute) is a unique identity (key) of the respective node. The other parameters are either primitive values (names, etc) or identities of other nodes, representing references. For instance, each node's second argument is a reference to its parent node.

In contrast to relational databases, logic programs do not store facts for every relation but can define predicates via derivation rules. In a database sense these are similar to 'views'. However, unlike views, logic predicates can be defined recursively. In Figure 1 shows some of the basic JTransformer predicates (all those ending with a capital T – for 'tree'). It also illustrates some simple rules that project away details of the basic JTransformer predicates that are not relevant to our later examples. Capital spelling indicates logic variables. The underscore is a pseudo-variable indicating attributes whose value is irrelevant.

```
type(Type,Name) :-
   classDefT(Type,_,Name,_).

field(Field, InClass, FType, FName) :-
   T = type(_,FType,_),
   fieldDefT(Field,InClass,T,FName,_).

method(Meth,InClass,MName,Args,RetType) :-
   T = type(_,RetType,_),
   methodDefT(Meth,InClass,MName,Args,T,_,_).

instanceMethod(Method,InClass,Name,Args,Type) :-
   method(Method,InClass,Name,Args,Type),
   not(Name = '<init>'),
   not(Name = '<cinit>').

accesses(AccessingM,InBlock,OnRecv,AccessedField) :-
   getFieldT(_,InBlock,AccessingM,OnRecv,_,AccessedField).

calls(CallingM,InBlock,CalledM,Args) :-  // Inst. method
    execT(Exec,InBlock,CallingM, Call),
    applyT(Call,Exec,CallingM,_,_,Args,CalledM).
calls(CallingM,InBlock,CalledM,Args) :-  // Constructor
    newClassT(_,InBlock,CallingM,CalledM,Args,_,_,_).

param(Param,InMethod,Type) :-
   paramDefT(Param,InMethod,type(_,Type,_),_).

forLoopBody(For,InMethod, LoopBody) :-
   forLoopT(For,_,InMethod,_,_,_,LoopBody).
```

Figure 1: Simple analysis predicates defined on the basis of the AST representation of JTransfomer

```
1  direct_fanin(Meth,N) :-
2     setof(Caller, external_call(Caller,Meth), CallersOfMeth),
3     length(CallersOfMeth,N).

4  external_call(A,B) :-
5     calls(A,_,_,B,_),       % A calls B
6     method(A,AClass,_,_,_), % A's class is AClass
7     method(B,BClass,_,_,_), % B's class is BClass
8     not(AClass = BClass).
```

Figure 2: Direct fan-in analysis in Prolog.

4. SUITABLE BENCHMARKS

In order to compare the logic-based infrastructures with respect to their efficiency and scalability it is neither feasible nor relevant to implement all known concern mining techniques. It suffices to analyze examples that are suitable as benchmarks. Indeed, we found that some techniques are not suitable as benchmarks because the logic-based implementation approach yields programs that are so simple that they do not illustrate relevant aspects of the problems we want to address.

For instance, Figure 2 shows how the core of a fan-in analysis framework can be implemented. It corresponds to the computation of direct fan-in the FINT system of Marin et al. [21][1]. The `external_call` predicate in line 4-8 uses the predicates from Figure 1 to capture external calls, that is calls of a method from a method in a different class. The entire direct fan-in computation is expressed in lines 1-3 using the predefined Prolog predicate setof/3. Its invocation in line 2 ensures that the variable CallersOfMeth is bound to a list containing all different external callers of the method Meth. The fan-in is the number of elements in this list. Due to the extremely simple nature of this problem, not just the code size is minimal but also the run-time. For JHotDraw 6.01b this trivial implementation finds all results in 78 milliseconds.

[1]FINT [21] distinguishes the computation of direct fan-in (to static types), its propagation up and down the type hierarchy, and the filtering and interpretation of the results.

We chose to focus our evaluation on the problem of design pattern detection. Design patters often have crosscutting implementations that can be captured cleanly in aspects [9]. Furthermore, they are well-understood and widely used in modern software development. Last but not least, we found that design patterns are sufficiently complex to illustrate all relevant issues we wanted to analyze.

5. DESIGN PATTERN DETECTION

In this section we explain how we formulate logic queries based on design pattern structures, and illustrate this on a concrete example. The queries are then used to evaluate the investigated infrastructures with respect to performance and scalability in section 7.

5.1 What do we Need to Know?

Design pattern descriptions typically use roles at the interface level, e.g. the *Subject* class, the *observers* field, the *notify()* method, the *update()* Method.

In the course of our experiments we found out that for most non-trivial examples program elements at much finer levels of granularity and much more complex relations are essential to describe a pattern. Precise detection of pattern participants often requires to identify fine-grained roles first, since they determine the characteristic properties of enclosing interface-level elements.

For instance, the interface level description of the *notify()* Method of a *Subject* Class (being a void method without any parameters) is an utterly insufficient detection criterion. There are usually too many methods in a software system with this property and in addition a programmer might decide to implement a variant that passes parameters or results.

The essential characteristic of the *notify()* method is that it contains an iteration that invokes the *update()* method on all elements of the *observers* field. Thus, detection of the *notify()* role method requires detecting loop statements (for, while, iterators) and being able to perform a static dataflow analysis that determines that the *update()* method is invoked on elements of the *observers* field.

Similarly, the essential property of a Composite pattern, is that each method in the *Composite*'s interface forwards its own invocation to every *Component*. More precisely, every method invokes a method with the same signature on each of the elements of the *children* field.

Thus, one of the first findings of our experiments was that it is not possible to limit the degree of information represented about a program. Each detection task has its own peculiarities and requires different details. In general the full representation of the program must be available and arbitrarily complex relations (e.g. points-to information, control and data flow between elements) must be inferable or represented explicitly. This result is consistent with the observation of Verbaere et al. [**?**] about the need to perform data flow analysis for ensuring correct refactorings.

5.2 Formulating Patterns and their Role Characteristics

A pattern's class diagram can provide all required information about role elements and their relationships, provided that it includes notes showing the typical code snippets in certain methods. Alternatively, the relevant behavioral information can be represented by dynamic diagrams. For the particular problem of design pattern detection, the characteristics of each role in the pattern description are identified form the UML diagram and translated into a logic predicate.

Typically, each predicate is used to identify candidates for a specific role that program elements play in a pattern implementation (such as *Observer*, *attach()*, or *notify()*). It defines a relation between this role and other roles that interact with it. Accordingly, the predicate is parameterized by logic variables, each representing an element with a specific role. This has the advantage that a query match provides us not only with the program elements found, but also with the role(s) they each play in that particular pattern occurrence.

The detector of a pattern is a predicate that orchestrates the interplay of all the different roles of the pattern. It invokes all the predicates defining role characteristics and defines their interrelation.

```
1   mineObserver(Subj,ObsF,AttachM,DetachM,
2                NotifyM,Obs,UpdateM) :-
3      notifyMethod(NotifyM,ObsF,UpdateM),
4      registryMethod(AttachM,Subj,Obs,ObsF),
5      registryMethod(DetachM,Subj,Obs,ObsF),
6      updateMethod(Obs,UpdateM),
7      not(NotifyM = AttachM),
8      not(AttachM = DetachM),
9      not(DetachM = NotifyM).

10  notifyMethod(NotifyM,ObsF,UpdateM) :-
11     % Access to own Observers field candidate:
12     accesses(NotifyM, Block, 'null', ObsF)
13     % Invocation of UpdateM candidate on Observer candidate :
14               calls(NotifyM, Block, UpdateM,[_]),
15     % Both are in the same loop within NotifyM:
16     forLoopBody(_, NotifyM, Block).
17
18  registryMethod(Method,Subj,Obs,ObsF) :-
19     instanceMethod(Method,Subject,[Param],void),
20     modifierT(Method, public),
21     param(Param,Method,Obs),
22     accesses(Method, _, 'null', ObsF).

23  updateMethod(Obs,UpdateM) :-
24     instanceMethod(UpdateM,Obs,[Param],void),
25     modifierT(UpdateM, public),
26     param(Param,UpdateM,Obs).
```

Figure 3: Detection of Observer pattern occurrences

5.3 Observer

We have defined detectors for several design patterns, but due to space constraints we focus on a single well-known and widely used pattern to illustrate the concern queries we use to evaluate the three logic-based infrastructures. The Observer design pattern [5] is a non-trivial design pattern involving several program elements in different roles. Role types are *Subject* and *Observer*, role methods are *attach()*, *detach()*, and *notify()* on the *Subject* and *update(Subject)* on the *Observer*. The *Subject* type also contains the *observers* role field. We are assuming the pull model, i.e., *Subject*s passing self-references as part of the updating process.

The `mineObserver` predicate implementing the observer detection identifies the individual role elements in turn and specifies their relationships, as shown in Figure 3. First, we specify the constraints for the individual role methods: we must identify candidates for the *notify()*, *attach()*, *detach()* and *update()* role method role (line 3-6). Note that there is no specific detector for the *observers* role field since we found that it has no sufficiently selective characteristics that are independent of the way how it is used by the method roles. Therefore, the detection of the *observers* field simply results from the agreement of the method role detectors on the same *observers* field candidate.

The detection of a *notify()* method involves identifying a method that access an own field that could be the *observers* field (see line 11 — 'null' represents the access to 'this') and invokes a method that could be the *update()* method (line 12). Since these criteria alone are too weak we additionally check that both happens in the body of the same loop (line 13). Note that the check shown here for brevity is quite restrictive since it only checks for `for` loops and ignores a possible deeper nesting of calls. Still it is already sufficient to detect 3 of 5 occurrences of the observer patten in JHotDraw. A more general variant will have better recall.

The *attach(Observer)* and *detach(Observer)* have a similar structure: they are both public instance methods that access the *observers* field. This is expressed by using the `registryMethod` detector (lines 15-19) in both cases.

The `updateMethod` predicate describes the characteristics of the role method *update(Subject)*: a public method on the *Observer* type with an argument of type *Subject*.

6. EVALUATION

We found that among the tested systems, JTransformer was the only one that could express all the relevant queries. In this section we describe and analyze the performance results for JTransformer on the full benchmark. In order to show that our experiments where realistic, we also evaluate the quality of the design pattern detection approach with respect to precision and recall. In the next section we focus on the differences in expressiveness, performance and scalability between JTransformer, JQuery and Code Quest.

We have tested the query frameworks on two software systems: the JHotDraw open source graphical editing framework [29] and the Eclipse integrated development environment [4]. JHotDraw was chosen since it is considered a benchmark system for concern mining and design pattern detection. The Eclipse sources, on the other hand, were chosen to test the scalability of our approach.

6.1 Query Performance

We first tried our detectors with JTransformer running on SWI-Prolog. This yielded absolutely unsatisfactory performance (e.g. more than two hours for detecting observers in JHotDraw).

Then we repeated them with the Conditional Transformation Core (CTC), a language-independent core for program analysis and transformation frameworks, developed at the university of Bonn as a future replacement of JTransformer. The CTC delegates the parsing and creation of source code to plugin components, focusing on the efficient analysis and transformation of the generated factbases. Java programs are processed by using the JTransformer as a plugin. The numbers in Table 2 are reported for the CTC/JT combination. The improvements compared to pure JTransformer experiments result from the fact that the CTC includes a compiler that optimizes detectors before executing them.

Table 2 shows the performance results for the two investigated systems, along with their sizes in terms of numbers of non-comment, non-empty lines of code, number of classes, fields, methods, total number of generated logic facts and the space required for storing them on disk. These numbers include the source code of the analysed system and interface-level information from used byte code libraries. The table focuses on the detection of Observer pattern occurrences. The numbers show that the CTC/JT combination exhibits excellent performance. It clearly achieves the desired goal of being able to run individual detectors even on very large systems such as the Eclipse platform in a few seconds.

The times given for the queries were obtained on a Dell D620 with an 2 GHz Intel DualCore CPU and 2 GB of RAM. On a Dell D600, with a single Intel 1.8 GHz processor and just 0.5 GB of memory the observers where detected for JHotDraw within 60 milliseconds, which is a surprisingly low difference to the 40 milliseconds on the much faster computer. We will run all our detectors also for Eclipse on the older machine in order to evaluate the influence of memory size.

It is noteworthy that the size of Eclipse is 34 times the size of JHotDraw whereas the time for Eclipse is 200 times the time for JHotDraw. This suggests a linear increase of time over size with a factor of 5.8. However, the line numbers are not a good indicator. We will conduct more detailed analyzes on more examples and measuring the exact numbers of facts in the database.

In addition to query time we also measured startup time. JTransformer needed 50 minutes for compiling the complete Eclipse Core. The generated factbase can be saved to a file. Reloading the 233 MB file holding the Eclipse factbase takes around 3 minutes, which we consider bearable for a project of this size. On the JHotDraw example, JTransformer needed 20 seconds for the initial compilation. Updates of the factbase after edits are performed incrementally, propagating only the changed parts of a project. In most cases they are hardly noticeable to users.

6.2 Accuracy

Besides performance/efficiency, accuracy is the prime concern for design pattern detection approaches. Since design patterns are not necessarily code fragments but rather solution structures, their actual implementation can vary considerably. Ideally, a design pattern detection approach properly identifies all conceivable variants.

The accuracy criterion can be broken down into *precision* (few false positives) and *recall* (few false negatives). Since we are planning to utilize the results of our pattern queries as input for partly or fully automated refactorings, we focus more on precision (i.e., to avoid refactoring a non-pattern). As a consequence, our pattern queries closely correspond to the pattern structure described in the literature [5].

We identified Singleton and Observer pattern occurrences in both JHotDraw and Eclipse. Our approach exhibits excellent precision: all design pattern instances identified are correct, and in all cases (for both patterns and both software systems) roles are properly assigned to the program elements comprising the pattern implementation.

Recall is harder to evaluate as it requires knowledge of all

Table 1: Design Pattern Occurrences in JHotDraw

Subject-Observer	Detected?	Deviation
StandardDrawing-DrawingChangeListener	Yes	—
CompositeFigure-Figure	Yes	—
StandardDrawingView-Figure	Yes	—
CommandMenu-Command	No	- *detach()* method does not access *observers* field
StandardDrawingView-Painter	No	- Two overlapping Observer occurrences - *notify()* method does not reference *observers* field

Table 2: Performance Measurements for Executing the Observer Query with CTC/JT

	LOC	Classes	Fields	Methods	Facts	Disk space	Occurrences	Time
JHotDraw 6.0b1	28,399	1.405	3.754	14.143	71,786	7.7 MB	3 [of 5]	0.04 sec.
Eclipse 3.1 Core	974,527	14.616	45.324	103.663	2.009.404	233.1 MB	612	8.00 sec.

pattern occurrences in the target code base, which we do not have for the Eclipse project. For JHotDraw, we can use the findings of others (e.g., [24]), which identify two Singleton and five Observer occurrences. Our Singleton query properly identifies all Singleton occurrences. The Observer query finds three of the five patterns. The other two vary in structure from the GoF pattern description so that they are not recognized. Table 1 lists the Observer occurrences and possible deviations from the GoF pattern description.

It is clear that, in terms of recall, our approach has room for improvement. The instances our queries did not capture are reasonable variants and should have been detected. Form the point of view of using the query results as input to a (semi-) automated refactoring approach such as role-based refactoring [10] however, it is clear that only those pattern occurrences can be refactored, which match the expected code structure exactly (and that is the case for the detected patterns).

As part of our future work we will investigate how pattern queries can be formulated to achieve increased recall with little or no negative impact for precision. Of particular interest is the question of how to define query/refactoring pairs, i.e., how to deal with the fact that pattern variants are not only difficult to detect, but also require different refactoring instructions.

6.3 Extraction

In order to use the analysis results for a subsequent refactoring to aspects, we need to integrate analysis and program transformation. Neither CodeQuest nor JQuery and Soul [22] support this integration, except for manual work with assert/retract statements in Prolog.

Pure Prolog approaches are only of limited use for realizing transformations due to the interference of transformation and analysis inherent to SLD resolution. Upon backtracking, the search for more results of a query already sees the modified state of the database, which might be the reason for extremely weird behavior of predicates. Most importantly, Prolog predicates that contain assert and retract statements lack a declarative semantics, which in turn means that it is impossible to analyze their effects.

In contrast, conditional transformations (CTs) [14] provide a declarative abstraction for logic-based program transformations and enable generic analyzes for interferences between different transformations [15, 16]. Our approach is to use CTs, supported by JTransformer and the CTC, since we found them be much better suited for addressing the challenges involved in refactoring the detected patterns.

Compared to other logic-based frameworks, the CTC/JT approach is the only one that provides a seamless integration of analysis and transformation at higher abstraction level than Prolog, which we think is essential for a smooth integration of concern exploration, extraction, and refactoring to aspects.

An alternative is to employ a role-based refactoring approach [10]. It can provide support for refactoring patterns once the roles of the comprising program elements have been properly identified. Since our mining approach yields a correct role mapping in all cases, integrating the two will be straight-forward.

7. COMPARISON

To better judge the strengths and weaknesses of the three investigated analysis frameworks, we compared CTC/JT in detail to both CodeQuest and JQuery taking into account their peculiarities.

7.1 CodeQuest

As the CodeQuest engine was not available to us, we could not use it to run our design pattern queries. Instead, we compared the two approaches by running the queries presented in the CodeQuest paper [7] using either JTransformer alone or the CTC/JT combination. Because we couldn't run our experiments on precisely the same hardware and software configuration as used by [7] our results cannot be taken as an exact comparison. Still, we were able to determine if the two approaches' performance is within the same order of magnitude and to derive some interesting results about the effectiveness and costs of some optimization techniques.

Regarding scalability, we did not experience the problems reported in [7] for the evaluated Prolog system, XSB. With CTC/JT running on SWI-Prolog we were able to run our analyzes even on the Eclipse platform, the largest benchmark analyzed in [7]. This was surprising since our test machine had just half as much main memory as the one running XSB

Table 3: Comparison of JQuery and CTC/JTransformer Query Performance

	JQuery		**CTC/JTransformer**	**Ratio JQ:JT**	
	First [ms]	Cache [ms]	Time [ms]	First	Cache
mineSingleton	7,934	397	16	**496**	**25**
publicGetters	3,992	380	50	**80**	**8**
updateMethod	3,606	183	16	**225**	**11**
registryMethod	14,609	387	15	**974**	**26**
notifyMethod	—	—	1	—	—
mineObserver	—	—	656	—	—

Table 4: General Comparison of JQuery and JTransformer

	JQuery	**JTransformer/CTC**
Expressiveness	Interface, Calls, Field Accesses	Full AST
Startup	Immediate	Seconds up to 50 minutes (Eclipse)
Scalability	up to 0.5K Classes (JHotDraw)	over 11K Classes (Eclipse)
Multi-Project Support	Yes	Yes
Integration	Analysis	Analysis and Transformation
Applicability	Java 1.4 programs	Java 1.4 programs
Compatibility	Java 5 / Eclipse 3.1	Java 5 / Eclipse 3.2.x
Installation and Documentation	+++	+

and SWI's clause database is known to be less space effective [2] than the one of XSB. Apparently, the CTC optimizations require significantly less memory than the tabling technique implemented in XSB. This result makes us confident that we will encounter no scalability problems with CTC/JT, even if it does not use the database-centric approach of CodeQuest.

Regarding performance, our results tend to confirm the numbers reported in [7] for XSB. Again, this was surprising, since XSB is a highly optimized implementation of a Prolog interpreter with tabling and the queries where manually optimized to achieve the best results on XSB. In contrast, our queries where not optimized manually and the experiments where run on SWI-Prolog at a time when it was not among the fastest Prolog implementations[2].

Thus, it seems that the precompiler included in the CTC is quite successful in most cases. However, it performed poorly on CodeQuest Query 2 ('Find all methods M that write a field of a particular type T or a subtype of T.'). Running Query 2 on JHotDraw using standard SWI-Prolog takes 500.28 seconds. The optimized version automatically created by the CTC is still slow (115.66 seconds). However, if we use a slightly different predicate definition (merely reordering the parameters by hand), producing all the 3598 results for the query on JHotDraw with CTC/JT takes less than 0.05 seconds.

We analyzed Query 2 in order to understand why our automated optimizations failed and whether our manual optimization could be generalized and applied automatically in a future version of the CTC compiler. We found that Query 2 is structured so that the way the recursive predicate is called prevents the input arguments of the call being used in the loop body. The input is passed only to the recursive invocation. This is a problem for Prolog's top-down SLDNF resolution, which fails to take advantage of input values in such a case. If we reorder the calls, so that the recursive invocation is the first in the loop, SLD resolution of this case does not terminate. In contrast, the bottom-up evaluation by loop fusion employed in CodeQuest can gracefully handle such scenarios.

[2]See [26] for an overview of SWI-Prolog performance bottlenecks and planned optimizations.

Since we ran our experiments on JHotDraw, for which [7] provides no performance numbers, our interim results do not represent a direct comparison with CodeQuest but can only provide an indication of certain trends. For a conclusive performance comparison with CodeQuest we will repeat our tests on the data samples used in [7].

7.2 JQuery

JQuery is publicly available and comes with very detailed documentation. Thus it seemed feasible to test our pattern detectors with JQuery by adapting the predicate definitions to the representation of Java elements used by JQuery. Unfortunately, this was not possible in all cases because JQuery does not offer access to the entire AST. In particular, information about loops and blocks is not represented in JQuery but required for identification of the *notify()* method (see `notifyMethod` detector, Figure 3, lines 10-14).

Because of these reasons, we had to restrict our comparison to predicates expressible in JQuery: the Singleton detector, a predicate to determine all public getter methods, and the update and registry method detectors. Table 3 shows the query execution times for these predicates, the notify method detector and observer detector. JQuery optimizes query run-times by caching query results. The table reflects this by separately listing the time for the first invocation of a predicate (column 'First') and for the subsequent invocationsm which just access values from the cache (Column 'Cache'). The ratio of JQuery to CTC/JT performance is also shown for both of these categories.

The table shows that CTC/JT is 2 to 3 orders of magnitude faster for initial queries, and still one order of magnitude faster than JQuery's access to the cache. This was surprising, since the TyRuBa system underlying JQuery performs indexing, caching, and some additional, non-standard optimizations (e.g. mode declaration based reordering of literals). Currently, we do not know ho to explain our findings. In order to asses the influence of the underlying Prolog implementations it is necessary to compare TyRuBa against SWI-Prolog on some data-intensive benchmarks. In order to asses the influence of the CTC precompiler it would be necessary to port it to JQuery/TyRuBa and evaluate its effectiveness in that environment. Both are topics for future investigation.

In [8] JQuery has been found to be unable to work on large systems, such as the Eclipse sources. Unfortunately, our experiments confirmed this result.

On JHotDraw example, JQuery showed extremely good startup times. This is due to the fact that, unlike JTransformer, JQuery does not automatically compile the entire project at startup. Instead, it supports demand-driven creation of factbases. Only the parts that are necessary for evaluating a particular query are compiled on the fly and cached for later use. The downside of this technique is that it may result in very long query evaluation times, as shown above.

Overall, JQuery differs from the CTC/JTransformer approach in a number of ways, which are summarized in table 4. It should be noted that JQuery is easier to install and use, especially for non-expert users. JQuery's particular strength is the very nice representation of query results in an easily configurable tree view style. As a result of our evaluation we are now working with the JQuery team on an integration of the strengths of both systems (JQuery and the JTransformer/CTC combination).

8. SUMMARY AND CONCLUSIONS

In this paper we addressed the issue of a programmer-friendly and versatile infrastructure for concern identification and exploration that is able to express easily many different detection techniques, run the corresponding detectors efficiently on the code to be analyzed, and seamlessly use the analysis results for subsequent extraction.

We have identified desirable properties for such an infrastructure and have compared different logic-based candidate infrastructures, using detection of design patterns based on their structural properties as an example. We have shown how the JTransformer system can be applied to search software systems for occurrences of design patterns and have shown its advantages with respect to the other evaluated systems.

We believe that the fact that (pattern) queries are resolved almost instantly makes the pair CTC/JTransformer suitable for interactive code exploration, and thus useful for design pattern detection in particular and concern mining in general. Developers can define both simple and complex queries for code fragments and receive immediate results that can be seamlessly used for subsequent concern extraction.

9. ACKNOWLEDGEMENTS

The official version of JQuery 3.1.5 available when we started our experiments aborted queries that returned more than 2500 results. This was intended to not overwhelm users and to avoid run-time and memory consumption getting too high. We are deeply indebted to Kris de Volder's who provided us with an update that made it possible to disable query abortion, thus enabling fair comparisons. In addition we are indebted to Kris for his patient answering of many questions about JQuery.

We want to thank Oege de Moor for taking time to write a very detailed review of the section about CodeQuest in the initial version of this paper. His feedback helped us eliminate an easy to misinterpret statement and definitely improved the presentation of this section. Furthermore, Oege de Moor and Elnar Hajiyev provided details about the benchmarks used for CodeQuest that will enable a more thorough comparison than the one reported in this paper.

10. REFERENCES

[1] Francesca Arcelli, Claudia Raibulet, Yann-Gael Gueheneuc, Giuliano Antoniol, and Jason McC Smith. Workshop on design pattern detection for reverse engineering. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, page 316, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Vítor Santos Costa. Prolog performance on larger datasets. In Michael Hanus, editor, *PADL*, volume 4354 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2007.

[3] Markus Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, 2001.

[4] Jim des Rivières and John Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Stefan Hanenberg Günter Kniesel, Tobias Rho. Reusable pattern implementations need generic aspects. Presented at the Workshop on Reflection, AOP and Meta-Data for Software Evolution at ECOOP '04, 2004.

[7] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[8] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. Codequest: querying source code with datalog. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103, New York, NY, USA, 2005. ACM Press.

[9] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pages 161–173. ACM Press, 2002.

[10] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 135–146. ACM Press, 2005.

[11] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development (AOSD '03)*, pages 178–187. ACM Press, 2003.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[14] Günter Kniesel. A Logic Foundation for Conditional Program Transformations. Technical report IAI-TR-2006-01, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, January 2006.

[15] Günter Kniesel. Detection and resolution of weaving interactions. In *Transactions on Aspect-Oriented Software Development, Special issue 'Dependencies and Interactions with Aspects'*. Springer, (to appear) 2007.

[16] Günter Kniesel and Uwe Bardey. An analysis of the correctness and completeness of aspect weaving. In *Proceedings of Working Conference on Reverse Engineering 2006 (WCRE 2006)*, pages 324–333. IEEE, October 2006.

[17] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - a framework for load-time transformation of Java class files. In *First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 100 – 110. IEEE Computer Society Press, November 2001. ISBN 0-7695-1387-5.

[18] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - A Powerful Back-End for Aspect-Oriented Programming. In Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, chapter 9. Addison-Wesley, 2004. ISBN 0-321-21976-7.

[19] Helge Koch. Ein Refactoring-Framework für Java. Diploma thesis, CS Dept. III, University of Bonn, Germany, April 2002.

[20] Andreas Ludwig and Dirk Heuzeroth. Metaprogramming in the large. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 178–187, London, UK, 2001. Springer-Verlag.

[21] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*, pages 132–141. IEEE Computer Society, 2004.

[22] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Special Issue of Elsevier Journal on Expert Systems with Applications*, 23(4):405–413, 2002.

[23] Ladan Tahvildari and Kostas Kontogiannis. On the role of design patterns in quality-driven re-engineering. *csmr*, 00:0230, 2002.

[24] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[25] Marek Vokác. An efficient tool for recovering design patterns from c++ code. *Journal of Object Technology*, 5(1):139–157, 2006.

[26] Jan Wielemaker. SWI-Prolog performance issues. http://gollem.science.uva.nl/twiki/pl/bin/view/

Development/PrologPerformance, October 2006.
[27] The Eclipse java development tools (JDT) project. http://www.eclipse.org/jdt/.
[28] The Eclipse web site. http://www.eclipse.org.
[29] The JHotDraw project web site. http://www.jhotdraw.org.
[30] The JTransformer project. http://roots.iai.uni-bonn.de/research/jtransformer/.