

Identifying Refactoring Opportunities Using Logic Meta Programming

Tom Tourwé and Tom Mens*

Programming Technology Lab

Vrije Universiteit Brussel

Pleinlaan 2, 1050 Brussel, Belgium

Email: {tom.tourwe.tom.mens}@vub.ac.be

Abstract—In this paper, we show how automated support can be provided for identifying refactoring opportunities, e.g., when an application’s design should be refactored and which refactoring(s) in particular should be applied. Such support is achieved by using the technique of logic meta programming to detect so-called bad smells and by defining a framework that uses this information to propose adequate refactorings. We report on some initial but promising experiments that were applied using the proposed techniques.

I. INTRODUCTION

Refactoring is the process of changing an application’s design without changing it’s overall behavior [19]. Its goal is to prevent the design from aging and to ensure the appropriate flexibility to enable smooth integration of future extensions. Although the definition of refactoring has been around for several years, its importance in object-oriented development and reengineering has only recently been acknowledged. Most major integrated development environments for object-oriented programming languages incorporate support for refactoring [22], [23], refactoring is more and more discussed in the context of reengineering legacy applications [9], [26] and it is included as an explicit activity in agile development processes [2], [28].

We can identify three distinct steps in the refactoring process:

- 1) detect when an application should be refactored
- 2) identify which refactoring(s) should be applied and where
- 3) (automatically) perform these refactorings

The last step is often divided into two different phases: checking if the appropriate preconditions of the refactoring hold (to ensure the refactoring is behavior preserving) and actually applying the necessary changes.

Currently, no development environment offers support for this complete process. The support offered by most environments is limited to step 3, e.g. they present a list of refactorings to the developer, and upon selection of any one of these, automatically perform the corresponding changes to the application. Although this relieves the developer from the difficult and error-prone process of performing these changes manually, it still requires him to apply step 1 and 2 by hand. Both steps can

be considered as hard as, or perhaps even harder than, the task of manually performing changes, due to the following reasons:

- Current-day development environments only offer a narrow and local view on the source code of an application. Most environments are file based, which makes it difficult to browse an entire class hierarchy, let alone the different implementations of a method implemented across this hierarchy. Consequently, such environments are incapable of presenting a global overview of the overall structure and design of the application.
- Current-day development environments only allow predefined querying of the source code of an application. Most environments provide support for finding all users of a particular class, or all senders and implementors of a particular method. They do not allow more sophisticated queries, however, such as finding all methods that directly access a particular instance variable of a class, while that class explicitly provides corresponding accessors and mutators for that purpose. Such a request can only be realized by manual inspection.
- Documentation of the application’s design is often missing or completely outdated. Consequently, developers may not be aware of particular design guidelines or coding conventions that are used throughout the application. As a result, they may violate such guidelines and coding conventions without even knowing. Clearly, this degrades the design, which will eventually lead to the problem of design erosion [29].
- Even if developers become aware of the fact that an application’s design is degrading, or when they have spotted design guideline violations, they may not know about refactoring and will try to remedy the situation manually (which is, as already mentioned, a time-consuming and error-prone process). If they know about refactoring, they may have a hard time finding out which refactoring can be applied, as there are many, and some of them are even largely similar. Even if they know which refactoring(s) should be applied, they may lack the appropriate information necessary for applying it (e.g., which classes and methods the refactoring should change).

These are exactly the kind of problems we want to tackle in this paper. The approach we put forward consists of identifying *bad smells* in the source code of an application. Originally coined by Kent Beck [10], the term bad smell refers to *structures in the code that suggest (sometimes scream for) the*

* Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium). This research is funded by FWO research grant G.0452.03: “A formal foundation for software refactoring”

possibility of refactoring. Once identified, we can use the information about these bad smells to propose adequate refactorings, that can be used to reduce the bad smell, or even remove it altogether. Furthermore, we will show that even more opportunities for refactoring can be derived, based on the refactoring opportunities identified by the bad smells.

The remainder of this paper is structured as follows: the following section introduces the technique of logic meta programming, which we will use to overcome the above mentioned problems and detect bad smells in the code automatically (section III). Section IV proceeds to show how the so-gathered information can be used to propose a list of refactorings that can remedy the situation. Section V introduces the notion of *cascaded refactoring opportunities*, which are refactoring opportunities derived from already identified opportunities. Subsequent sections discuss the initial experiments we conducted, the tool support we provide and related work.

II. LOGIC META PROGRAMMING

Logic meta programming (LMP) is currently being investigated as a technique to support state-of-the-art software development. It is based on a tight symbiosis between an object-oriented base language and a declarative meta language. This makes it possible to reason about and to manipulate object-oriented programs in a straightforward and intuitive way [31]. The technique has already been used to check and enforce programming conventions and best-practice patterns [17], to detect design pattern instances in existing source code [30], to specify and reason about (the evolution of) design patterns [18], [27], and to check conformance of a software implementation to its intended architecture [16].

The LMP technique is independent of the particular base language that is used. Up till now, we experimented with both Smalltalk and Java as the base language [31], [8]. All experiments reported on in this paper were conducted using SOUL, a logic programming language implemented on top of the object-oriented language Smalltalk [30], [31], as the meta language.

A. Syntax

SOUL is a variant of Prolog [6] with some minor syntactic differences. Below we give an example of the syntax. Like in Prolog, lines starting with % indicate comments and a comma denotes a logical conjunction. The main differences with Prolog are that logic variables are always preceded by a question mark (e.g., ?P, ?C, ?D).

```
% two logic facts
subclass(TermVisitor,SimpleVisitor).
subclass(SimpleVisitor,NamedVariableVisitor).

% two logic rules
hierarchy(?P,?C) :- subclass(?P,?C).
hierarchy(?P,?C) :- subclass(?P,?D), hierarchy(?D,?C)
```

The logic rules above simply state that a class ?P is an ancestor of a class ?C if ?C is a subclass of ?P, or if there exists an intermediate class ?D, which is a subclass of ?P and an ancestor of class ?C. Logic queries can be used to trigger the above logic clauses. For example, the query `hierarchy(TermVisitor,?C)` determines whether a descendant of class `TermVisitor` exists, and

retrieves the result in the variable ?C (in this case there are two solutions ?C=`SimpleVisitor` and ?C=`NamedVariableVisitor`). The query `hierarchy(TermVisitor,NamedVariableVisitor)` checks whether the class `NamedVariableVisitor` is a (possibly indirect) descendant of `TermVisitor`, and returns `true`.

B. Virtual Logic Facts

The essential distinguishing feature of SOUL compared to other logic-based approaches (such as [4]) is its use of *virtual* logic facts. All entities in the object-oriented source code (i.e., classes, methods, variables, inheritance relationships, ...) can be directly accessed from within the SOUL environment through a metalevel interface of *representational mapping* predicates. The main advantage of this approach, as opposed to having a separate repository of logic facts extracted from the code, is that we will always reason about the latest version of the source code, thus avoiding consistency problems.

Table I lists some of the representational mapping predicates. Typically, these predicates are used like ordinary logic predicates (i.e., for *checking* and *retrieving* information).

III. DETECTING BAD SMELLS

In this section, we explain how logic meta programming can be used to detect bad smells. We consider two examples, with increasing complexity. In the first example, we detect the situation where a method definition includes a formal parameter that is never used. In the second example, we analyze a class hierarchy and verify whether it features a clear interface.

A. Obsolete Parameter

A method defines an obsolete parameter if it includes a formal parameter in its signature that is never used in its implementation. The implementation of a method may span several classes, of course, since the method can be overridden in a subclass. Given a particular class, the developer does not know beforehand which method implementation he has to inspect, nor does he know which subclasses of the class override which methods, and which subclasses don't. Suppose the class implements m methods, and suppose n subclasses of this class override this method. The developer then needs to inspect $m \times n$ methods to detect an obsolete parameter. Moreover, he needs to do this *for every formal parameter* that the method defines. Clearly, detecting the occurrence of an obsolete parameter manually is a very hard and time consuming process. In what follows, we will show an example of the occurrence of an obsolete parameter and we will explain how this situation can be detected automatically in our LMP environment.

1) *Example*: Figure 1 depicts an example of the obsolete parameter bad smell in the code. The `TermVisitor` class defines an abstract `objectVisit:` method with one parameter¹. This method is overridden in two subclasses, `FixVisitor` and `SimpleVisitor`. Neither of the two implementations of the `objectVisit:` method in these classes uses the parameter. As such, the parameter is obsolete.

¹Note that Smalltalk uses *keywords* to identify the parameters a method defines. For example, the name `objectVisit:` consist of one keyword, while `prettyPrintOn:scope:` consists of two keywords. The first method thus defines one parameter, and the second defines two.

Representational Mapping Predicate	Description
<code>class(?C)</code>	<code>c</code> must be a class
<code>hierarchy(?P, ?C)</code>	class <code>c</code> must be a (possibly indirect) subclass of class <code>P</code>
<code>classImplements(?C, ?M)</code>	<code>c</code> implements a method named <code>M</code>
<code>instanceVariable(?C, ?V)</code>	<code>v</code> must be an instance variable of class <code>c</code>

TABLE I
REPRESENTATIONAL MAPPING PREDICATES

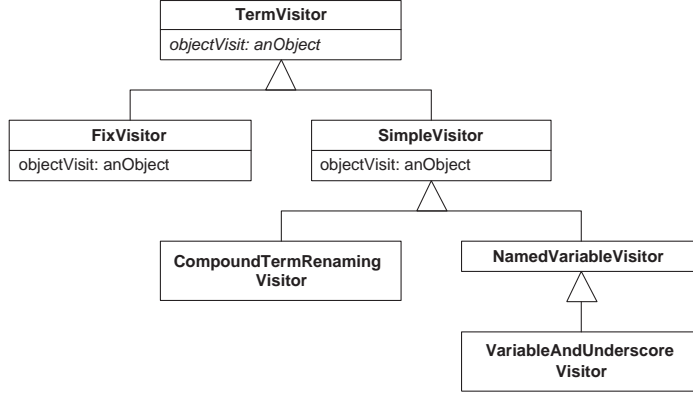


Fig. 1. A method with an obsolete parameter

2) *Detecting the Bad Smell*: Detecting whether a formal parameter is not used by a method boils down to checking whether the method itself and none of its overriding methods uses this parameter. We can use the following two logic rules to implement such an algorithm:

```

obsoleteParameter(?class, ?selector, ?parameter) :-
[1] classImplements(?class, ?selector),
[2] parameterOf(?class, ?selector, ?parameter),
[3] forall(subclassImplements(?class, ?selector, ?subclass),
[4]      not(selectorUsesParameter(?subclass, ?selector,
                                ?parameter)))
  
```

First, we retrieve the methods implemented by a given class (line 1, through the `classImplements` predicate, which is part of the representational mapping), and we retrieve all the parameters of each method at line 2 (by means of the `parameterOf` predicate). For each subclass of the given class that implements the given method (gathered by means of the `subclassImplements` predicate), we check if it uses the given parameter (line 4, using the `selectorUsesParameter` predicate).

The `selectorUsesParameter` predicate itself is implemented as follows:

```

selectorUsesParameter(?class, ?selector, ?parameter) :-
[1] classImplementsMethodNamed(?class, ?selector, ?method),
[2] parsetreeUsesVariable(?method, ?parameter).
  
```

It uses a variant of the `classImplements` predicate, the `classImplementsMethodNamed` predicate that also returns the parsetree of the method identified by the class and the selector (line 1). The `parsetreeUsesVariable` predicate is then used to traverse this parsetree and look for uses of the specified parameter (line 2). It is defined in terms of the `traverseMethodParseTree` predicate, that implements a general parse tree matching algorithm.

B. Inappropriate Interfaces

Good interfaces are extremely important when designing flexible and reusable object-oriented systems. Any situation in which the interface of a class is inappropriate, incomplete or unclear should thus be avoided at all costs. In the initial stage, classes may define the appropriate interface, but due to the constant evolution of an application, and due to different developers working on the same code base, the interface may deteriorate over time.

Detecting the inappropriate interface bad smell manually is quite a difficult task because one has to analyze an entire class hierarchy, and the interfaces it defines. Supposing this hierarchy consists of m classes, we have to consider all possible combinations of these classes and check whether any of those combinations share an interface. Since there are 2^m such combinations, this is quite cumbersome. Even an automatic approach is not feasible when dealing with such huge numbers. The problem can be alleviated however, by first applying some lightweight techniques (such as metrics [12]) to detect class hierarchies that should be investigated further and in more detail.

We will first show an example of the inappropriate interfaces bad smell, and afterwards discuss how we can detect the problem by using our LMP environment. We will not elaborate on the metrics we use for identifying class hierarchies for further examination, since this is outside the scope of this paper.

1) *Example*: Consider the `AbstractTerm` hierarchy depicted in Figure 2. As can be observed, the `CallTerm`, `CompoundTerm`, `SmalltalkTerm` and `QuotedCodeTerm` classes each provide an implementation for the `terms` method, whereas all other classes (including `AbstractTerm`) do not. This situation is problematic

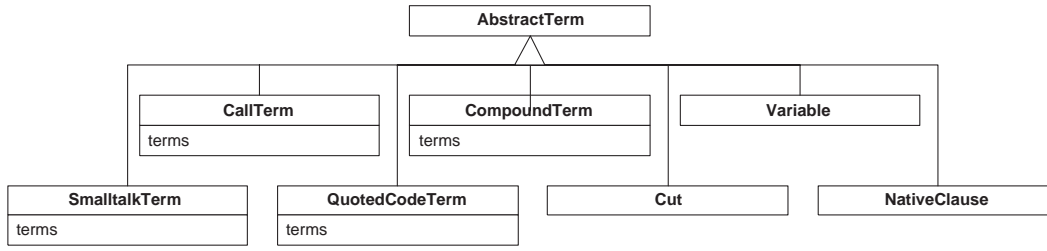


Fig. 2. An example of an inappropriate interface

for two reasons:

- The classes in this hierarchy cannot be used polymorphically, at least not in a statically-typed language (such as Java and C++), since there is no common ancestor that includes the `terms` method in its interface.
- Extending the `AbstractTerm` hierarchy with a new class becomes harder. It is absolutely unclear from the design which subclasses of `AbstractTerm` should provide an implementation for the `terms` method and which classes should not. A developer confronted with this situation should thus know exactly what he is doing.

2) *Detecting The Bad Smell:* We use the following algorithm for detecting the problem of inappropriate interfaces in a hierarchy of classes:

- retrieve all direct subclasses of the root class of the hierarchy.
- compute all possible subsets of this set of classes.
- for each of these subsets, compute the intersection of the interfaces of all classes contained in the subset.
- in each of the resulting intersections, exclude all those methods that are present in the interface of the root class of the hierarchy

Clearly, this algorithm grows exponentially with the number of subclasses, because we compute all possible subsets. Therefore, we restrict it by only considering subsets of three or more classes that should share an interface of two or more methods. For explanatory purposes, we do not take these restrictions into account in the implementation of this algorithm below.

The purpose of the `commonSubclassInterface` predicate is to retrieve all subclasses of a given root class (line 2), find out whether some subclasses share a common interface (line 3), and compute the difference between this shared interface and the interface of the root class (line 4).

```

commonSubclassInterface(?class, ?interface, ?subclasses) :-
[1] classInterface(?class, ?classInterface),
[2] allSubclasses(?class, ?scs),
[3] sharedInterface(?scs, ?commonInterface, ?subclasses),
[4] difference(?commonInterface, ?classInterface,
              ?interface).
  
```

The `allSubclasses` and `difference` predicate are basic predicates. The former retrieves the list of all subclasses of a given class, while the latter computes the difference between two sets. Their implementation is not considered here. The implementation of the `sharedInterface` predicate looks as follows:

```

sharedInterface(?classes, ?interface, ?subset) :-
[1] subset(?subset, ?classes),
[2] commonInterface(?subset, ?interface).
  
```

It simply computes a subset for the set of classes passed to it (line 1), and derives the common interface between the classes of this subset (line 2). The `subset` predicate is a library predicate that computes a possible subset of a given set. The `commonInterface` predicate is implemented as follows:

```

commonInterface(?classes, ?interface) :-
[1] findall(?itf,
[2]     and(member(?class, ?classes),
[3]         classInterface(?class, ?itf)),
[4]     ?interfaces),
[5] intersection(?interfaces, ?interface)
  
```

It first computes the interfaces of all classes in the provided set (first four lines), and then computes the intersection of this set of interfaces (last line).

One particular technical problem of this algorithm is that it computes the shared interface between any combination of subclasses of a given class. As such, the shared interface that is detected between four classes of this set, may also be detected for any combination of three classes of these four classes. In other words, the results returned by the algorithm may possibly include duplicated entries. These can be removed easily however, which results in the following implementation for the `inappropriateInterface` predicate:

```

inappropriateInterface(?class, ?interface, ?subclasses) :-
  findall(<?itf, ?scs>,
    commonSubclassInterface(?class, ?itf, ?scs),
    ?result),
  removeDuplicates(?result, ?nodups),
  member(<?interface, ?subclasses>, ?nodups).
  
```

C. Discussion

As can be seen from these two examples, the technique of logic meta programming is extremely well suited to detect bad smells in source code. Thanks to distinguishing and powerful features as backtracking and unification, we can write complicated search algorithms in a straightforward, understandable and concise way. Moreover, the LMP environment has full access to an application's source code, which means we are not hampered by the narrow view offered by standard development environments. In combination with the fact that the logic paradigm naturally allows us to express sophisticated queries, this also means we are no longer limited to the basic querying tools offered by these environments.

IV. PROPOSING REFACTORINGS

Now that we have shown how we can detect bad smells using LMP, we are ready to explain how the information gathered in this way can be used to propose appropriate refactorings. We

want to stress that we do not aim to apply these refactorings automatically. Many times, several refactorings can be chosen to remedy a particular situation, and it is impossible to infer automatically which of these refactorings is most appropriate. Therefore, we present the developer with a list of refactorings that he can use, and it is his responsibility to pick out the appropriate one(s).

A. Overview

We will use one single and general predicate for proposing refactorings: the `proposeRefactoring` predicate. This predicate thus provides one single hook for tools that want to know which refactorings are applicable. The predicate has the following form:

```
proposeRefactoring(?entity,?refactoring,?arguments)
```

Its first argument represents the entity for which we want to detect refactoring opportunities. It can be any source code artifact, but at the moment we only use classes, methods or instance variables, since these are all entities for which some refactorings have been defined [19], [10]. The second argument identifies the particular refactoring that should be applied. It's value can thus be `addClass`, `pullUpMethod`, `abstractVariable` or any other refactoring that is defined [10]. The last argument of the predicate identifies the list of arguments that should be passed to the refactoring. These arguments can be any source code artifact [25].

B. Obsolete Parameter

Obsolete parameters can be removed from a particular method by applying the `removeParameter` refactoring. This refactoring makes sure that the obsolete parameter is removed from a particular method, all of its overriding methods, and all of the method's callers. The refactoring requires as arguments the class in which the method is defined, the method defining the obsolete parameter, and the obsolete parameter itself. This is exactly the information that has been gathered by the `obsoleteParameter` predicate, as presented previously. We can use this information in the following way to propose the appropriate `removeParameter` refactoring:

```
proposeRefactoring(?class,removeParameter,
    <?class,?selector,?parameter>) :-
    obsoleteParameter(?class,?selector,?parameter)
```

Applying the following query on the example presented in Figure 1

```
:- proposeRefactoring(TermVisitor,?refactoring,
    ?arguments)
```

returns the following result:

```
proposeRefactoring(TermVisitor,removeParameter,
    <AbstractTerm, objectVisit:>)
```

which is exactly what is to be expected.

C. Inappropriate Interfaces

There are two general solutions to overcome the problem of inappropriate interfaces. A developer can either insert an intermediate superclass between the root class of the hierarchy

and the subclasses that implement a shared interface, or he can augment the interface of the root class of the hierarchy with the interface shared by the subclasses. These two solutions correspond to an `addClass` and an `addMethod` refactoring respectively.

The `addClass` refactoring requires as arguments the root of the hierarchy, a list of subclasses of this root class that should become subclasses of the newly introduced class, and a list of selectors that are shared by the subclasses and that should be implemented in the newly introduced class². Since this is exactly the kind of information that is derived by the `inappropriateInterface` predicate, the refactoring can be proposed in the following way:

```
proposeRefactoring(?class,addClass,
    <?class,?interface,?subclasses>) :-
    inappropriateInterface(?class,?interface,?subclasses)
```

Similarly, the `addMethod` refactoring requires as arguments a list of methods to be added and the root class to which they should be added. The refactoring can thus be proposed as follows. Note how information about the subclasses that share the interface is not used in this case.

```
proposeRefactoring(?class,addMethod,<?class,?interface>) :-
    inappropriateInterface(?class,?interface,?)
```

Applying the following query on the `AbstractTerm` class hierarchy

```
:- proposeRefactoring(AbstractTerm,?refactoring,
    ?arguments)
```

yields two results

```
proposeRefactoring(AbstractTerm, addClass,
    <AbstractTerm, terms,
    <CallTerm, CompoundTerm,
    SmalltalkTerm,
    QuotedCodeTerm>)
proposeRefactoring(AbstractTerm, addMethod,
    <AbstractTerm, terms>)
```

Note that this is only a very basic example, since it involves a class hierarchy which is only one level deep. More complex hierarchies may require more complex refactorings. Due to space limitations we cannot present the algorithm that covers these cases as well here.

V. CASCADED REFACTORING OPPORTUNITIES

Many times, part of the purpose of a particular refactoring is to enable the possibility of performing another refactoring. For example, before an instance variable shared by a number of subclasses can be pulled up, it is required that all subclasses refer to this variable by the same name. A `renameVariable` refactoring may thus be mandatory before a `pullUpVariable` refactoring can be applied. The application of one particular refactoring may thus open possibilities for other refactorings to be applied. We call this phenomenon *cascaded refactoring opportunities*. Our LMP environment naturally allows us to detect cascaded refactoring opportunities. We can easily use the information gathered about a particular refactoring opportunity to discover even more opportunities, as we will show next.

²Our definition of the `addClass` refactoring differs from existing definitions [19], [25] because it explicitly includes an extra parameter that holds this shared interface. This is only a minor technical issue, however.

A. A Cascaded removeParameter Refactoring Opportunity

As an example, consider the case where an obsolete parameter bad smell is detected for a particular method m . The appropriate refactoring to be applied is a *removeParameter* refactoring, which will remove the parameter from m 's definition, as well as remove it from all call sites. A method n , in which a call to m occurs, may as well include this parameter in its definition, and may not use it besides for calling m : (in which case we call it a *delegated parameter*). As such, method n also defines an obsolete parameter, all be it an indirect one.

This bad smell can be detected by means of the following rule:

```
proposeRefactoring(?class1, removeParameter,
  <?class1,?selector1>) :-
  proposeRefactoring(?class2, removeParameter,
    <?class2,?selector2,?parameter>),
  senderOf(?class1,?selector1,?selector2),
  delegatedParameter(?class1,?selector1,?parameter)
```

The `senderOf` predicate is used to determine all senders of a particular method, while the `delegatedParameter` predicate checks whether the parameter is used in the calling method as a *delegated parameter* only. If this is not the case, the additional refactoring cannot be proposed, since it means the parameter is used elsewhere in the method.

B. A Cascaded pullUpVariable Refactoring Opportunity

Based on the inappropriate interface bad smell, many different cascaded refactoring opportunities can be proposed. We will only provide one example of these.

Subclasses sharing a common interface often also share some state (e.g. some instance variables). When an intermediate superclass is inserted by means of an *addClass* refactoring, this common state can be factored out from the subclasses into the intermediate superclass. Observe that without this intermediate superclass, such a refactoring would not be possible. The state would then have to be factored out into a more general superclass, which may have other subclasses that do not have to contain this state.

Detecting when a shared state can be factored out after an *addClass* refactoring has been applied, and proposing the appropriate *pullUpVariable* refactoring, is achieved by using the following logic rule:

```
proposeRefactoring(?class, pullUpVariable,
  <?variable,?subclasses>) :-
  proposeRefactoring(?class, addClass,
    <?class,?,?subclasses>),
  sharedVariable(?subclasses,?variable)
```

The `sharedVariable` predicate simply checks whether the list of subclasses passed to it share a common variable. If this is the case, the *pullUpVariable* refactoring can be proposed.

C. Discussion

Clearly, applying one single refactoring may give rise to a multitude of other refactoring opportunities. The question may thus be raised whether searching for cascaded refactoring opportunities is both useful and scalable.

As for its usefulness, we firmly believe it is of prime importance that as many refactoring opportunities as possible are discovered in one single step. The benefits of automatically

identifying refactoring opportunities largely vanishes when the developer is forced to check for such opportunities constantly. Rather, he wishes to check once and get as many identified opportunities as possible.

The question of scalability refers to the fact that each existing refactoring may depend on all other existing refactorings. As such, identifying cascaded refactoring opportunities requires checking opportunities for each and every possible refactoring, which could take quite some time. When a new refactoring is introduced, this also requires to identify how it relates to all existing refactorings and change all the logic rules that check for refactoring opportunities. Clearly, this is quite cumbersome. Given the initial status of our work, we are not yet in the position to study such issues. This should thus be considered future work.

VI. EXPERIMENTS

Of particular importance for initially validating our techniques is that we test them on an application of which we have intimate knowledge. Only then will we be able to assess the correctness of the identified bad smells and the usefulness of the proposed refactorings. The SOUL application itself is the most obvious candidate for testing purposes, since we developed it ourselves and we thus know its implementation very well.

We used version 3.0 of SOUL, which was the latest version at the time we started doing our experiments. In this version, the implementation consists of 84 classes and approximately 1100 different methods (not counting methods that are overridden, so there are many more method *implementations*), which makes it a small to medium-sized application. The implementation is based around 5 important class hierarchies:

- `AbstractTerm` consists of 30 classes and 116 methods.
- `HornClause` consists of 4 classes and 26 methods.
- `TermVisitor` consists of 6 classes and 18 methods.
- `AbstractRepository` consists of 6 classes and 63 methods.
- `Frame` consists of 5 classes and 12 methods.

We searched for refactoring opportunities in these five hierarchies, only considering those bad smells presented in the previous sections.

A. Proposed Refactorings

The refactoring opportunities that were proposed do not include cascaded refactoring opportunities, as no instances of those were found.

1) Obsolete Parameter:

```
removeParameter(AbstractTerm,
  unifyWithKeywordedCompound:inEnv:myIndex:hisIndex:-
  inSource:,
  myIndex:)
```

```
removeParameter(AbstractTerm,
  unifyWithKeywordedCompound:inEnv:myIndex:hisIndex:-
  inSource:,
  hisIndex:)
```

```
removeParameter(AbstractTerm,
  unifyWithKeywordedCompound:inEnv:myIndex:hisIndex:-
  inSource:,
  inSource:)
```

```
removeParameter(TermVisitor,
```

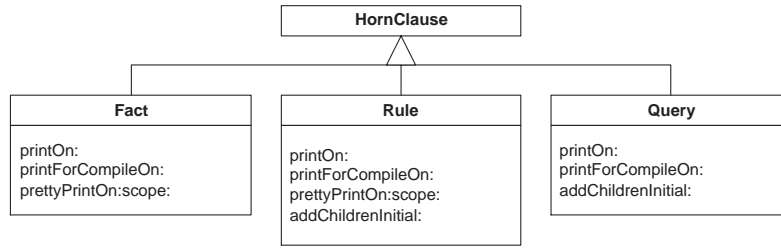


Fig. 3. The original HornClause class hierarchy

```

objectVisit:,
objectVisit:)

```

The first three refactorings are proposed due to the fact that the `unifyWithKeywordedCompound:inEnv:myIndex:hisIndex:-inSource:` method in the `AbstractTerm` class does not use its three last formal parameters. The method is implemented by three different classes of the hierarchy, but neither implementation uses these parameters. Similarly, the `objectVisit:` method of the `TermVisitor` class does not use its single parameter. This method is overridden in two subclasses of the hierarchy, but neither one uses the parameter.

2) Inappropriate Interface:

```

addClass(AbstractTerm,terms, <CallTerm, SmalltalkTerm,
                               QuotedCodeTerm, CompoundTerm>)

addMethod(AbstractTerm, terms)

addClass(HornClause, <printOn:,printForCompileOn:>,
          <Query, Rule, Fact>)

addClass(HornClause, <printOn:,printForCompileOn:,
                  addChildrenInitial:>,
          <Query, Rule>)

addClass(HornClause, <printOn:,printForCompileOn:,
                  prettyPrintOn:scope:>,
          <Rule, Fact>)

addMethod(HornClause, <printOn:,printForCompileOn:>)

addMethod(HornClause,
          <printOn:,printForCompileOn:,
            addChildrenInitial:>)

addMethod(HornClause,
          <printOn:,printForCompileOn:,
            prettyPrintOn:scope:>)

```

The first two refactorings are proposed due to the inappropriate interface bad smell in the `AbstractTerm` hierarchy depicted in Figure 2. All other refactorings are proposed due to the same bad smell in the `HornClause` hierarchy, which is shown in Figure 3.

As can be observed, these results overlap to some extent: the `printOn:` and `printForCompileOn:` methods are reported three times. This is unavoidable given the current implementation of our detection algorithm, which searches for a shared interface between any combination of classes in a particular class hierarchy. As such, one combination may share an interface that is a subset of the interface shared by another, slightly different, combination. Detecting such situations is possible, but requires a more advanced, and hence more complex, search algorithm. We may however prefer the above results, since they show more clearly which interfaces are shared between which combinations of classes, and such information is important for

the refactoring process.

B. Solutions

The following refactorings were effectively used in practice to improve the design of the SOUL application.

1) Obsolete Parameter:

- We could have applied the three *removeParameter* refactorings that were proposed for the `unifyWithKeywordedCompound:inEnv:myIndex:hisIndex:inSource` method. However, we chose to apply a *removeMethod* refactoring instead, which removed the method from the implementation. This last refactoring was proposed, based on the detection of the bad smell that this particular method was never called.
- The `objectVisit:` method participates in an instance of the *Visitor* design pattern. Its formal parameter represents the element object that is being visited. One of the coding conventions used in the SOUL implementation is that a visitor method (such as the `objectVisit:` method) should always define such a parameter. Therefore, we do not remove it. The fact that this parameter is not used in this particular situation, may hint that this is too strict a restriction, however.

2) Inappropriate Interface:

- We changed the superclass of the `Fact` class from `HornClause` to `Rule`. Such a refactoring was actually not proposed by our techniques, but after studying the code of both classes, we found out that such a change would result in an improved and cleaner design. This change solves the problem of the `prettyPrintOn:scope:` method. This method only needs to be implemented by the `Rule` and `Fact` classes, since queries (represented by the `Query` class) never need to be pretty printed.
- The *addMethod* refactoring that is proposed to add the `printOn:` and `printForCompileOn:` methods to the interface of the `HornClause` class, is applied. These methods are defined as abstract methods, and since all subclasses of the `HornClause` class already provide an implementation for them, no additional changes are required.
- The second *addMethod* refactoring, that is proposed to add the `addChildrenInitial:` method to the interface of the `HornClause` class is also applied. Since the `Fact` class does not define this method, we should provide an implementation for it. However, since we also changed the inheritance relationships (see the first refactoring), it turns out this implementation is not necessary.

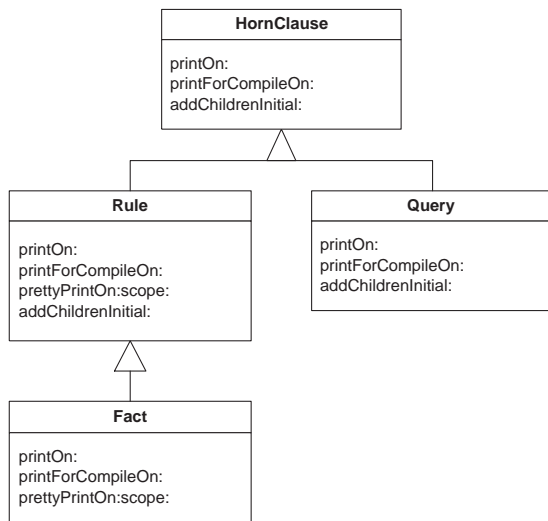


Fig. 4. The improved `HornClause` class hierarchy

The resulting and improved design of the `HornClause` hierarchy is depicted in Figure 4.

C. Discussion

SOUL is a research prototype tool, and as such is a small-to medium-sized application, which is being worked on by a small number of people, and which is developed with flexibility and extensibility in mind (to allow us to experiment with new language features easily, for example). Furthermore, we applied our techniques to detect refactoring opportunities based on only two bad smells. Given these facts, the results are quite surprising. Most of the refactorings that were proposed were effectively applied to achieve a cleaner and better design. If a particular refactoring was not applied, for whatever reason, a closer look at the identified bad smell revealed that there was indeed a problem, which should be solved. We can assume that such problems will only aggravate for large-scale applications worked on by many more developers.

A potential disadvantage of the approach is that many refactorings can be proposed, certainly if large(r)-scale applications are considered and cascaded refactoring opportunities are taken into account (as discussed in Section V). The developer may thus be presented with a large list of refactorings, and may no longer see the wood for the trees. This is unavoidable, however, since one particular bad smell can often be remedied by a multitude of refactorings. Furthermore, we do not want to refactor the design automatically [3]. The only solution thus consists of presenting the whole list to the developer, who should then pick out the appropriate refactorings himself. The problem may be alleviated up to some extent in two ways, however:

- 1) We could induce a particular order on the refactorings that are proposed, based on some criteria. For example, a developer would like to see a *removeMethod* refactoring before *removeParameter* refactorings, since the application of the former refactoring makes the latter ones obsolete (as was explained earlier in this section). This would also allow us to reduce the size of the list dynamically:

as the first refactoring in the list is applied, later refactorings that become obsolete may be removed. This requires that we study dependencies between different refactorings, which should be considered future work.

- 2) Include refactoring in the development process as a specific task that should be performed on regular times (as suggested in [2]). Regularly checking for refactoring opportunities and applying the corresponding refactorings will ensure that the list of possible refactorings remains small and manageable. We envision an approach where checking for refactoring opportunities happens at the same time as unit testing. The unit tests will then ensure the behavioral correctness of the application, whereas the refactoring tests will ensure its "structural" correctness.

VII. TOOL SUPPORT

Tool support for our approach is clearly indispensable, to shield a developer from the logic environment and to provide him with a straightforward and easy to use interface. The tool should be driven by the programmer, who selects a particular source code entity, such as a class or a method, to be analyzed. This analysis is performed automatically without user intervention, and results in a list of refactorings that can be applied. The developer is then able to pick out the refactoring(s) he wants to apply, and the tool automatically makes the necessary changes.

To provide this kind of tool support, we integrated our approach into the Refactoring Browser, which is the standard browser for the VisualWorks Smalltalk Integrated Development Environment. We augmented this browser with a SOUL tab (see Figure 5), which exists next to the other tabs already available (such as the *Comment*, *Code Critic* and *Hierarchy Diagram* tabs). This tab contains a list of bad smells that can currently be detected by the tool. The *Check the class interface* and *Check for obsolete parameters* entries that were explained above, are included in this list, for example. Upon selection of a class in the upper left pane, the developer can select a number of these bad smells (or all of them) and clicks the *Execute* button. As a result, a logic query will be launched that will check the source code for the selected bad smells. The results of this query (e.g., the refactorings that can possibly be applied) will be shown in the lower right pane. The developer can then simply select one of these refactorings, and execute it.

It should be noted that we explicitly allow a developer to select a number of bad smells (or even one single bad smell) to be checked, instead of automatically checking for all of them. The specific reason is that the current implementation of our LMP environment is not optimized, and as a consequence, checking for all bad smells on large software systems may take quite some time. Furthermore, a developer may know that a certain bad smell does not occur in a particular class hierarchy, and may thus not want to spend the time checking for it.

VIII. RELATED WORK

Obviously, the use of logic-based or query-based approaches for improving software evolution tool support is not new. For example, [4] propose a reverse engineering tool that stores all program information in a logic code repository to perform data

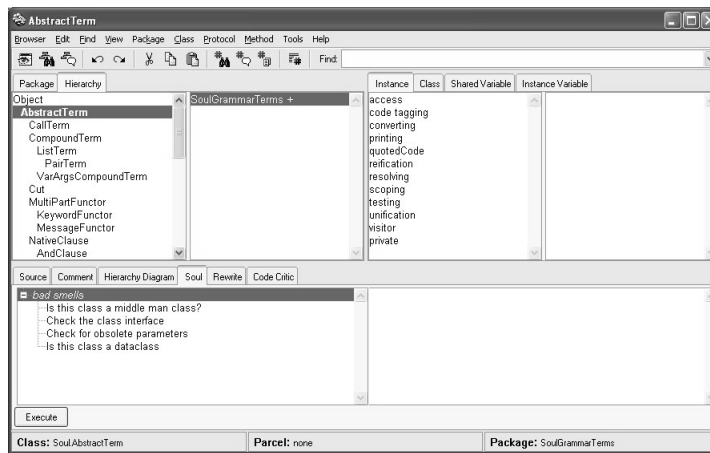


Fig. 5. Tool Support for Identifying Refactorings

flow analysis. [5] represent structural design information (more specifically, dependency relationships among modules) in a Prolog data base. A graphical query language is used to query the Prolog data base to identify and remove cyclic control dependencies. [21] present an algebraic framework for modelling and querying source code. This facilitates expressing high-level source code queries.

As already mentioned, a number of programming environments exist that provide support for refactoring. Most notably, the Refactoring Browser, which exists for many Smalltalk programming environments, and the Eclipse and IntelliJ environments for the Java programming language. The kind of support offered by these environments is limited to automatically applying the refactorings selected by a developer. They have no special provisions for detecting bad smells, nor for identifying which refactorings can be applied to remove them.

Other tools exist that can be used to verify source code quality [14], [1], [20]. The tool presented in [20], for example, is capable of detecting some basic coding errors, such as using `=` instead of `==` in an `if` statement. Generally, such tools lack a global notion of a program, and, as a consequence, they can not be used to detect more advanced coding errors, let alone bad smells on the design level.

A more advanced tool is the *Code Critic* tool included in the Refactoring Browser, that can be used to detect some bad smells. *Code Critic* is a Lint-like tool, that has been extended to include global design information. As such, it is not only able to detect various coding errors, but can also identify interface conflicts (such as methods that are sent but not implemented, or vice versa) and design flaws. The major shortcoming of this tool is that it strongly relies on the Smalltalk environment and its powerful meta programming capabilities. Smalltalk is a general-purpose programming language, and is thus not specifically designed to express rules or constraints, or straightforwardly implement reasoning algorithms. This hampers the extensibility and usability of the tool. For example, although it can be achieved, implementing an algorithm that detects the problem of inappropriate interfaces would be much harder, would be much more complex and would as a result be less readable. Furthermore, the *Code Critic* tool does not pro-

pose refactorings that can be applied to remedy the bad smells.

More recently, some techniques are being developed with the specific goal of identifying refactoring opportunities and proposing the corresponding refactorings [15], [7], [24], [13].

[15] reports on a tool that is able to derive program invariants from the source code automatically, and that uses these invariants to identify when a refactoring could be necessary. For example, one invariant may be that a certain parameter of a method is always constant, or is a function of the other parameters of a method. In that case, it might be possible to apply a *removeParameter* refactoring. The main problem with this approach is that it needs to run an application to infer the program invariants. To this extent, it uses a representative set of test suites. It is however impossible to guarantee that a test suite covers all possible uses of an application. Therefore, the invariants may not hold in general. Nonetheless, good results have been obtained, and we believe the approach is complementary to ours. Combining both approaches, resulting in a tool that uses static as well as dynamic information, seems promising.

[7] sketches an approach to detect duplicated code in an (object-oriented) application and proposes refactorings that can eliminate this duplication. It is based around an object-oriented meta model of the source code and a tool that is capable of detecting duplication in code. The refactorings that are proposed consist of removing duplicated methods, extracting duplicated code from within a method and insert an intermediate subclass to factor out the common code. Although we currently have no support for detecting code duplication, our LMP environment is general enough to allow us to implement this as well. We could thus incorporate the findings of [7] into our approach.

[24] uses object-oriented metrics to identify bad smells and propose adequate refactorings. They focus on *use relations* to propose *move method/attribute* and *extract/inline class* refactorings. The key concept underlying their approach is the *distance-based cohesion* metric, which measures the degree to which some parts (methods and variables of a class) belong together. The main difference between a metrics-based approach and our approach, is that metrics are still subject to interpretation, whereas our detection technique is more strict (a bad smell occurs or it does not). The approaches are thus clearly comple-

mentary, since some bad smells are subject to interpretation as well, whereas others are more strict.

IX. CONCLUSION & FUTURE WORK

In this paper, we have shown how support can be provided for detecting when a design should be refactored, as well as identifying which refactorings could be applied to improve this design. The approach we have demonstrated uses the technique of logic meta programming, which proves to be extremely well suited for detecting bad smells and for deriving the necessary information for the proposed refactorings. We provided two non-trivial but representative examples, that prove this. Furthermore, we illustrated how this approach complements existing refactoring tools, and how it can be integrated into such tools to effectively provide an environment supporting the complete refactoring process.

Future work first of all includes extending the tool to detect many more bad smells, such as those found in [10]. Since some of these bad smells are not as strict and clear-cut as the two examples we presented here, we may have to resort to the use of metrics (as in [24]). Our LMP environment can certainly be used to implement such metrics, and in fact already contains a metrics framework. We should also study the scalability of the (cascaded) refactoring opportunities, as discussed in Sections V and VI. We would like to investigate which refactorings depend upon other ones, and how and why this is the case. This should allow us to identify more clearly which refactoring opportunities can or will give rise to other opportunities. Another interesting research track is to incorporate *design pattern smells* into the approach, which would enable us to detect deteriorated design pattern implementations and propose adequate refactorings to correct them. This would require us to incorporate some fuzzy logic techniques (as suggested by [13]) or explanation-based constraint logic programming (as used by [11]), to identify such deteriorating design patterns. We would also like to integrate our ideas with the approach proposed by [15] to study where they overlap, where they complement each other, and if the resulting approach is more than the sum of its parts.

REFERENCES

- [1] A. V. Aho, B. W. Kernigan, and P.J. Weinberger. *Awk - A Pattern Scanning and Processing Language*, 1980.
- [2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [3] F. W. Callis. Problems with automatic restructurers. *SIGPLAN Notices*, 23:13–21, March 1988.
- [4] Gerardo Canfora and Aniello Cimitile. A logic-based approach to reverse engineering tools production. *Transactions on Software Engineering*, 18(12):1053–1064, December 1992.
- [5] M. Consens, A. Mendelzon, and A. G. Ryman. Visualizing and querying software structures. In *Proc. 11th Int'l Conf on Software Engineering*, pages 138–157, 1992.
- [6] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.
- [7] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proc. Int'l Conf. Software Maintenance*, pages 109–118. IEEE Computer Society Press, September 1999.
- [8] Johan Fabry. Supporting Development of Enterprise Javabeans through Declarative Meta Programming. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, *Object-Oriented Information Systems*, number 2425 in LNCS. Springer Verlag, 2002.
- [9] Richard Fanta and Vaclav Rajlich. Reengineering Object-Oriented Code. In *Proc. Int. Conf. on Software Maintenance*, pages 238–246. IEEE Computer Society Press, 1998. Bethesda, Maryland, March 16-19, 1998.
- [10] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [11] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects. In *Proc. TOOLS USA*, 2001.
- [12] Brian Henderson-Sellers. *Object-Oriented Metrics*. Prentice Hall, 1995.
- [13] Jens Jahnke, Wilhelm Schaefer, and Albert Zuendorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In M. Jazayeri and H. Schauer, editors, *Proc. 6th European Software Engineering Conference*, pages 193–210. Springer-Verlag, 1997.
- [14] S. Johnson. *Lint, a C Program Checker*, 1978.
- [15] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated Support for Program Refactoring using Invariants. In *Proc. Int. Conf. on Software Maintenance*, pages 736–743, 2001.
- [16] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2000.
- [17] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting Software Development through Declaratively Codified Programming Patterns. *Journal on Expert Systems with Applications*, December 2002.
- [18] Tom Mens and Tom Tourwé. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *Proc. Int. Conf. Software Maintenance*, pages 570–579. IEEE Computer Society, 2001.
- [19] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana Champaign, 1992.
- [20] Santanu Paul and Atul Prakash. A Framework for Source Code Search using Program Patterns. *Transactions on Software Engineering*, 20(6):463–475, June 1994.
- [21] Santanu Paul and Atul Prakash. Supporting queries on source code: A formal framework. *Software Engineering and Knowledge Engineering*, 4(3):325–348, September 1994.
- [22] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
- [23] Jocelyn Simmonds and Tom Mens. A comparison of software refactoring tools. Technical Report vub-prog-tr-02-15, Programming Technology Lab, November 2002.
- [24] Frank Simon, Frank Steinbrückner, and Clause Lewerent. Metrics Based Refactoring. In *Proc. 5th European Conference on Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society Press, 2001.
- [25] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, 2001.
- [26] Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.
- [27] Tom Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.
- [28] Arie van Deursen and Leon Moonen. The video store revisited – thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76, 2002. Alghero, Sardinia, Italy.
- [29] Jilles van Gurp and Jan Bosch. Design Erosion: Problems & Causes. *Journal of Systems & Software*, 61(2):105–119, 2001.
- [30] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *Proc. TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.
- [31] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.