Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2003

A Logic Meta-Programming Framework for Supporting the Refactoring Process

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Francisca Muñoz Bravo

Promotor: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotors: Dr. Tom Tourwé and Dr. Tom Mens (Vrije Universiteit Brussel)

Abstract

The objective of this thesis is to provide automated support for recognizing design flaws in object oriented code, suggesting proper refactorings and performing automatically the ones selected by the user.

Software suffers from inevitable changes throughout the development and the maintenance phase, and this usually affects its internal structure negatively. This makes new changes difficult to implement and the code drifts away from the original design. The introduced structural disorder can be countered by applying refactorings, a kind of code transformation that improves the internal structure without affecting the behavior of the application.

There are several tools that support applying refactorings in an automated way, but little help for deciding *where* to apply a refactoring and *which* refactoring could be applied.

This thesis presents an advanced refactoring tool that provides support for the earlier phases of the refactoring process, by detecting and analyzing bad code smells in a software application, proposing appropriate refactorings that solve these smells, and letting the user decide which one to apply. This tool relies on the technique of logic meta programming, a variant of the logic paradigm that allows to reason about code at a high level of abstraction.

Logic meta programming is used for querying about code and expressing structural relations like message sends, usage of variables or inheritance relations. This allows our tool to detect a number of bad smells, most of them highly time-consuming to detect manually. When a bad smell is detected, the relevant entities are analyzed in order to propose appropriate refactorings with the necessary parameters. For effectively applying the refactorings, the tool relies on existing refactoring implementations provided by the Refactoring Browser.

The tool was implemented in SOUL, a logic meta-programming language on top of Smalltalk, and validated by detecting the bad smells for three different case studies on medium-sized object-oriented applications: The Collection hierarchy, the HotDraw framework and the SOUL application itself. Most detected bad smells indicated situations that were worthy of attention, and many of the proposed refactorings were actually useful to resolve the bad smells.

Thanks to all the persons who made the EMOOSE program possible, it was a great life experience. Thanks to my advisors Tom Tourwé and Tom Mens for their continuous advice and support. Thanks to Johan Brichau for his helpful comments and all the people behind SOUL, I found it was an amazing application. Thanks to my fellow Emoosers for the all pleasant times we spent together, both in Nantes and Brussels. Thanks to my family and friends in Chile, who always kept an eye on me.

Contents

Chapter 1

Introduction

During the development of object oriented software and its maintenance phase, it is very likely that the code will experience modifications. This can happen due to a number of reasons, for instance changes in the requirements, bug fixes or the addition of unexpected features.

All these changes affect the behavior of the system, and they are usually introduced to keep the software up-to-date in a new or evolving environment. Any software that is related to a real-world problem domain, must continuously adapt to new conditions and scenarios as the problem domain changes. This constant production of new software versions is called *software evolution* and it normally increases the complexity of the system.

If the changes are performed by only extending the system without preserving or simplifying its structure, the code distances from the original design and new changes are more difficult to implement. Even though the software continues to work, the overall disorganization has augmented. This is reflected in bad coding practices like code duplication, lack of modularity, classes with too much functionality or violation of code conventions.

The structural disorder that may have been introduced by behavioral changes can be minimized by applying another kind of changes that only affect the internal structure of the system. They are called *refactorings* and include transformations like renaming of entities, moving methods between classes or hierarchy rearrangements. By applying appropriate refactorings to a system, the legibility and robustness of the system increases, new changes are easier to insert and the software maintenance cost is reduced.

In the following sections we give our approach for applying refactorings, the motivation and the general and specific objectives of this thesis.

1.1 Approach

Even if there is no formal definition of the refactoring process as a whole, there are some basic steps that should be performed to assure the effectiveness of the refactoring [33], [37].

1. Decide when an application should be refactored.
2. Identify which refactoring(s) can be applied, and where.
3. Assess the quality of the refactorings.
4. Perform the refactorings.
5. Evaluate the effect of the refactorings and check postconditions.

As for the questions of *when* and *where* the refactorings should be applied, Martin Fowler [26] has defined a number of bad programming or design practices called "bad smells", which indicate that something is being mishandled in the software. For example too large classes, procedural code, code duplication, etc. If bad smells are detected, the code should be refactored and the entities where the bad smells are detected is the place where the refactoring should be applied. Regarding to *which* refactoring should be applied, Fowler [26] also suggests suitable refactorings depending on the bad smell that has been found.

To assess the quality of the refactoring it is necessary to prove that the general state of the system will be improved after the application of the refactoring. This can be done by means of metrics [33, 21] or by reducing the total number of bad smells.

The refactorings can be applied by automated means, and there is a large amount of theoretical and practical research in this area. [40] [28] [46]

Sometimes due to efficiency reasons, we want to evaluate the effect refactoring after it is applied rather than before. This includes checking for side-effects and determine the postconditions of the entities related to the refactoring.

1.2 Motivation

Currently there are several tools like the Refactoring Browser [45], IntelliJ IDEA [16] and Eclipse [30] that support refactorings in a safe and automated way, ensuring that the refactoring is behavior preserving. These tools relieve the developer from the task of making the transformations manually, but they do not focus on the places where the code should be refactored, forcing him to realize time-consuming manual inspections to decide where to apply the refactorings.

Other tools perform automatic code inspections like LINT for C [31], LINT for Java [1], Smalltalk CodeCritic provided by the Refactory Browser and TogetherCC [15]. They point out the places where the code has bugs or other technical defects like error handling, misuse of operators, unused code or entities not following the coding standards. More advanced design issues are not addressed, like the bad smells defined in Fowler's Refactoring book [26].

The process of finding the right place to refactor, identify the proper refactoring and the parameters involved and see if it will improve the overall quality becomes a major task to the developer. Taking into account that nowadays there is no tool that covers the complete process, we propose to investigate whether automated support can be provided for performing some steps of the refactoring process. Specifically we will analyze the code statically, focusing on finding places containing bad smells in the code, obtaining the related entities and proposing suitable refactorings, and performing the refactorings. For applying the refactorings automatically we rely on existing efficient and safe implementations.

1.3 Objectives

The objective of this thesis is to provide automated help based on static analysis for recognizing bad smells in object oriented code, suggesting proper refactorings, and performing automatically the ones selected by the user.

In this way we expect to ease the refactoring process, so developers may be able to maintain the software structure in a highly automated manner, without consuming unnecessary resources.

Specifically, we will provide a flexible framework with a friendly user interface that handles bad smell definitions, analyzes these bad smells, proposes possible refactorings and performs composite refactorings. This framework will be implemented using the technique of logic meta programming (LMP) [57], having Smalltalk as the target language to reason about. The refactorings that the tool proposes will use the ones defined in the Refactoring Browser [45], but can be redefined over other tools with similar characteristics.

In order to accomplish these objectives, the following tasks have to be performed:

- Analyze the bad smells defined in Fowler's Refactoring book [26] and distinguish which of them can be detected automatically.
- Extend the Refactoring Browser in order to apply the refactorings defined in Fowler's Refactoring book.
- Provide a generic framework to link the relevant entities of the bad smells to possible refactorings.
- Apply the tool in different well-known applications to prove its usefulness and analyze its results.

1.4 Organization of the Dissertation

- The second chapter introduces context, this is the technique of logic meta programming and more details about refactoring, including related work.
- The third chapter describes the refactoring process by means of logic meta programming, as well as the provided tool support.
- The fourth chapter details the experiments carried out on three case studies, and report upon the results.
- The fifth chapter contains the conclusions.

Chapter 2

Context

This chapter gives a general introduction to the relevant subjects of this thesis. The first topic is logic meta programming, a variant of the logic paradigm that allows to query about code and express structural relations in various levels. The second topic is refactoring, a technique used to improve the quality of the code. Finally the related work is discussed.

2.1 Logic Meta Programming

Most of the existing tools for developing software are either oriented to the high level description of its design or to its low level API, with no relation between them. This makes it unfeasible to express high level structural information about the implementation itself, to propagate the changes in one level to the other automatically or to generate working code based on design constructs.

Part of these problems can be tackled using a *meta language* to extract and modify structural information of the program written in a *base language* [20].

Logic meta programming merges a logic language with a standard object oriented base language. Logic languages, like Prolog, are declarative languages that use logic inference as the only control structure, and fact and rules to establish the relation between data. In logic meta programming, the logic language operates by making descriptive statements about the base program and using a general control structure for stating this statements as facts and rules concerning its structure. Logic meta programming has a declarative approach that focuses on *what* the base language does by means of present entities and relations, instead of *how* the computations are effectuated.

This section introduces a logic meta programming language and explains how this language is suitable for representing high level queries, code entities relationships and design concepts.

2.1.1 Logic Meta Programming with SOUL

SOUL (Smalltalk Open Unification Language) [57, 56] is a variant of Prolog [18] constructed over Smalltalk Visualworks [13]. It provides the basic features of a logic language, which includes prolog-like list handling, pattern matching and logic primitives (and, or, not).

Syntax

The syntax of SOUL is very similar to the one of Prolog, where a comma denotes logical conjunction and dots are used to separate predicates. The main difference is the notation for variables: in SOUL, they start with a question mark (`?`) while in Prolog they are capitalized symbols. Regarding the separation of the predicate declaration from the body, Prolog uses the characters `:-` and SOUL uses the keyword `if`. SOUL handles lists in the same way as Prolog, only that the delimiters are angle brackets <`?list`> instead of square brackets `[List]` like in Prolog. To illustrate the difference of syntax between SOUL and Prolog we define the predicate `subset` which tests whether a list is subset of another:

```
% SOUL                                % Prolog
subset(<>,?).                         subset([],_).

subset(<?el| ?rest>,?list2) if        subset([El | Rest], List2) :-
    member(?el,?list2),                   member(El,List2),
    subset(?rest,?list2)                  subset(Rest,List2).
```

LiCoR Library

The LiCoR library is a set of pre-defined logic predicates to reason about the base-code. These predicates can also be extended by the user depending on his own needs. LiCoR is as language independent as possible, relying on a language dependent meta-level interface (MLI) that executes instructions for extracting the base language structure as seen on figure 2.1. This figure shows the different layers that are present in this logic meta-programming approach: a query using a predicate of LiCoR to reason about code is performed in the logic engine; the predicate of LiCoR invokes the MLI, which consults the code repository to extract the actual structure from the code.

In this case, the implementation of the predicates rely on the MLI, which is language dependent. This approach allows LiCoR to be extended for reasoning about other object oriented base languages like Java [24], but since our base language is Smalltalk, we use the MLI implementation for Smalltalk code.

An important feature of SOUL is the manipulation of objects from the underlying Smalltalk system. Smalltalk code can be inserted into the declarations of terms and the value of this term

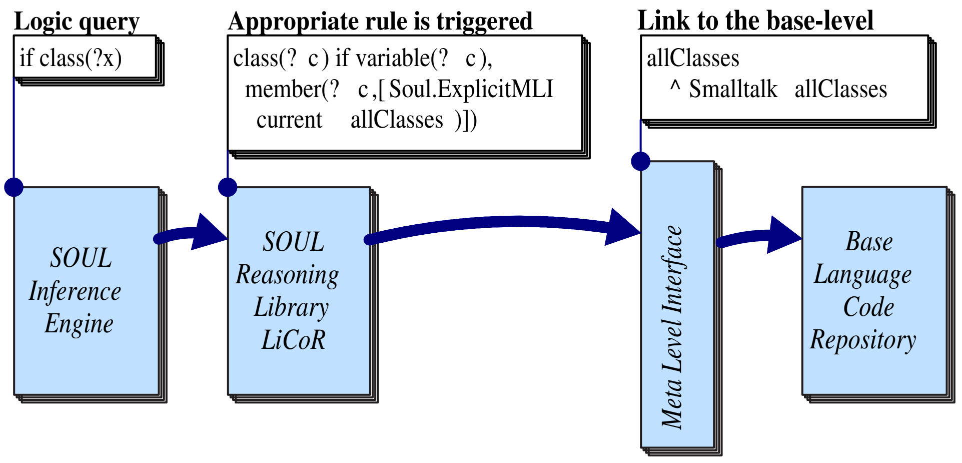

Logic query
if class(?x)
Appropriate rule is triggered
class(? c) if variable(? c), member(? c,[Soul.ExplicitMLI current allClasses)])
Link to the base-level
allClasses
^ Smalltalk allClasses
SOUL Inference Engine
SOUL Reasoning Library LiCoR
Meta Level Interface
Base Language Code Repository

Figure 2.1: The relation of SOUL with the base language

is the value returned by the evaluation of the code block. In the following code, we show how to generate a Smalltalk `Interval` using the SOUL variables `?from` and `?to`:

```
equals(?interval,[ Interval from:?from to:?to ])
```

The predicate `equals` verifies if two values are equal. Expressions between brackets (‘`[`’ and ‘`]`’) are blocks of Smalltalk code, and the value of these expressions are the result of the evaluation of the block. This means that `[ Interval from:?from to:?to ]` evaluates to an instance of the class `Interval`, and that this value is bound to the variable `?interval` in the only solution for this query.

Besides their use as terms, blocks of Smalltalk code can be also used as clauses. For this, the blocks are required to always return either true or false.

2.1.2 Using Logic Meta Programming for Reasoning

As SOUL is constructed over Smalltalk, it works directly on the current Smalltalk image and it is possible to access Smalltalk code directly from within the SOUL environment. This makes SOUL a dynamic and powerful environment for coding user defined extensions. Using Prolog as the language that states the structure of Smalltalk allows to express the structure, query it and modify it using few and usually legible concepts.

In practice, SOUL and LiCoR can be used as the base system in many software development and analysis tools for software evolution management, design patterns management, aspect oriented code generation [10] and refactoring for example.

In the area of software evolution management, SOUL can be used to in various ways:

- Express the desired structure of the software before its implementation, for generating some code to support this desired structure and for reconstructing the structure of an existing software. This makes it easier for new users to understand and explore the code at a higher level, not depending on some arbitrary documentation. [17] [20]
- SOUL can also synchronize some parts of the implementation and the design, control different software versions based on the structure, enforce the use of style conventions and detecting structure or style violations. [57]
- Design Patterns [27] can also be stated in terms of logic predicates, expressing the classes involved and the restrictions and relations they follow. [57] [36]
- Code generation is also possible using predicates of SOUL. It can be generated for defined high level structures like *design patterns* or other kind of designs, making the implementation more structured and controlled. Code generation can also be used for *refactoring purposes*, like deleting duplicated code in the subclasses and generating it in the superclass. [36]

In this thesis we focus on providing support for the refactoring process by means of detecting design flaws, analyzing the entities related to the flaw and proposing appropriate refactorings for the user to apply. In order to accomplish this goal, we use the predicates provided by SOUL for representing code entities and for reasoning about methods in detail. In the following sections, we explain some of the predicates provided by SOUL, how to use these predicates for defining new structural relations, and for detecting advanced design flaws.

Representational Mapping

In order to support the mentioned steps of the refactoring process, we will establish a flexible framework over the basic relations between code entities provided by SOUL.

This is possible by combining and extending basic predicates that represent code entities as shown in tables 2.1 and 2.2. For example `class(?C)` states that the variable `?C` is a class that exists in the current Smalltalk image, resulting in the unification of `?C` with all the classes of the image, see figure 2.1. If the predicate is used in the form `class([Array])`, it will answer true or false if the given class belongs to the image or not. This is a direct result of Prolog's unification of predicates with different results. Other interesting predicate is `methodInClass(?M,?C)`, which states that the class `?C` that has the method `?M`; if a method `?M` is provided, it returns the classes that has it; if a class `?C` is provided, it returns all the methods of the class; if none is provided, it matches all the possibilities, this means that it returns all the pairs of class - method that are present in the system; if both a class and a method are provided, it checks if the method belongs to the class or not.

RepresentationalMappingPredicate	**Description**
`class(?C)`	`?C` is a class
`instanceVariableInClass(?V,?C)`	`?V` is an instance variable of class `?C`
`superclassOf(?C,?P)`	class `?C` is the superclass of class `?P`
`subclassOf(?C,?P)`	class `?C` is the subclass of class `?P`
`abstractClass(?C)`	`?C` is an abstract class
`methodInClass(?M,?C)`	`?C` implements the method `?M`
`classUnderstandsSelector(?C,?S)`	`?C` understands the selector `?S`
`methodWithName(?M,?N)`	`?M` is a method with name `?N`
`methodWithNameInClass(?M,?N,?C)`	`?M` is a method with name `?N` defined in class `?C`

Table 2.1: Representational Mapping Predicates

MethodReasoningPredicate	**Description**
`abstractMethodInClass(?M,?C)`	`?C` implements an abstract method `?M`
`argumentsOfMethod(?A,?M)`	`?A` is the list of the arguments of `?M`
`temporariesOfMethod(?A,?M)`	`?A` is the list of the temporary variables of `?M`
`statementsOfMethod(?S,?M)`	`?S` is the list of the statements of `?M`
`methodWithAssignment(?M,?Var,?Value)`	`?M` has an assignment of `?Value` to `?Var`
`methodWithSend(?M,?Rec,?Msg,?Args)`	`?M` has a send to receiver `?Rec`, with message `?Msg` and arguments `?Args`
`methodWithVariable(?M, ?Var)`	`?M` uses a variable `?Var`

Table 2.2: Method Reasoning Predicates

Simple Structural Relations

The predicate `subclassOf` is not defined in the MLI since it can be easily derived from the predicate `superclassOf`.

```
subclassOf(?subclass,?superclass) if
[1]     superclassOf(?superclass,?subclass)
```

Using the predicates defined in the MLI, we can also define logic relations for hierarchy of classes as shown in the following code:

```
% first predicate
classInHierarchyOf(?P,?P).

% second predicate
classInHierarchyOf(?C,?P) if
[1]     subclassOf(?D,?P),
[2]     classInHierarchyOf(?C,?D)
```

The predicates above state that a class `?P` is an ancestor of another if it is the class itself (first predicate), or if there exists an intermediate class `?D`, which is a subclass of `?P` (line 6) and an ancestor of class `?C` (line 7). Logic queries can be used to trigger the above logic clauses. For example, the query `classInHierarchyOf(?C,[Array])` determines whether a descendant of class `Array` exists, and retrieves the result in the variable `?C`. The query `classInHierarchyOf([LargeWordArray], [Array])` checks whether the class `[LargeWordArray]` is a (possibly indirect) descendant of `[Array]`, and returns true.

Other simple example is the predicate `overridingSelector` that states whether a selector of a class is overriding another implementation in the same hierarchy or not.

```
overridingSelector(?class,?selector) if
[1]     methodWithNameInClass(?,?selector,?class),
[2]     superclassOf(?superclass,?class),
[3]     classUnderstandsSelector(?superclass,?selector)
```

Detecting Design Flaws

More high level predicates can be defined, that take into account the structure of the system. A typical design flaw that is encountered in object oriented programs, is the definition of abstract methods that are not implemented in the hierarchy. This is easily detected using logic meta programming:

```
    abstractMethodNotImplemented(?method,?class) if
    [1]     abstractClass(?class),
    [2]     abstractMethodInClass(?method,?class),
    [3]     methodWithName(?method,?selector),
    [4]     not(  one(
    [5]          and(methodWithNameInSubclassesOf(?subMethod,?selector,
       ?subclass,?class),
    [6]          not(abstractMethod(?subMethod)) )))
```

This predicate checks that for an abstract class `?class` (line 2) having an abstract method with name `?selector` (line 3 and 4) there is no valid declaration of a method named `?selector` in the direct or indirect subclasses of `?class` (lines 5, 6 and 7). Valid method in this case means that if a method `?submethod` is found int the hierarchy (line 6), this method can not be abstract (line 7).

The auxiliary predicate `methodWithNameInSubclassesOf` is defined as follows:

```
    methodWithNameInSubclassOf(?method,?selector,?subclass,?class) if
    [1]     classInHierarchyOf(?subclass,?class),
    [2]     not(equals(?class,?subclass)),
    [3]     methodWithNameInClass(?method,?selector,?subclass)
```

This predicate returns the method `?method` that has the name `?selector` in the hierarchy of `?class`.

The predicate `abstractMethodNotImplemented` can be used for searching abstract methods that are not implemented, for checking if a specific abstract method is not being implemented in the hierarchy or for checking if a certain abstract class has a method with this characteristics.

2.1.3 Conclusion

The logic meta programming approach presents a very natural way for reasoning about object oriented systems and detecting design flaws. This is achieved by using logic predicates to express and query the structure of the system.

This technique can easily express structural patterns, query the state of the code or detect flaws by means of logic predicates. The advantages of reasoning with SOUL and LiCoR for developing our logic framework are the following:

- We are able to express assertions about the code in a declarative nature, hence intuitive and readable.
- For constructing our framework, we rely on specific benefits of logic languages: *Multi-way reasoning* allows the same rule to be used in many different ways; *Unification* provides a powerful pattern matching mechanism; *Backtracking* enables finding all possible solutions of a query.
- By using SOUL for reasoning about code, our approach becomes *base-language independent*. The rules describing the structure can also be used to reason over other object oriented languages.
- SOUL is highly *customizable*: we can declare and use our own set of rules to support bad smell detection, propose refactorings and apply them to the code.

2.2 Refactoring

Refactoring is the technique that allows to change the structure of a program without affecting its functionality. In order to understand the importance of refactoring, we explain the problems of software evolution and how proper refactorings address some of these problems. For applying proper refactorings, it is important to recognize which part of the code contains design flaws, to compare the contribution of possible refactorings and finally applying a behavior preserving refactoring.

This section explain these concepts more in detail, and includes a list of useful refactorings and existing refactoring tools.

2.2.1 Software Evolution

Even though software maintenance is a highly resource-consuming phase, where the software has to adapt to changing scenarios or introduce new customer requirements, the way to handle software changes after the initial development is not clear.

The Software Life Cycle

The classic software life cycle states that software maintenance starts right after the first release to the user. This view does not take into account that changes are inevitable necessary and that each new change introduces architectural erosion and enlarges the amount of code, making following changes more difficult to apply. This will lead to a point where changes are too expensive to implement.

One view about software evolution called the *staged model* [6] has been proposed based on empirical observation. This model represents the software lifecycle as a sequence of stages. After the first successful release, the software has to be adapted to the ever-changing user requirements, changing environments or new business rules. This stage is called *evolution* and substantial changes are likely to be performed. The release to the user happens during the evolution of the software, experimenting iterations before and after this milestone. The following stage is called *servicing* and starts when the software is not flexible enough to perform important changes, only minor corrections and enhancements. The later stages of software are *phase out* and *close down*, where the software does not admit any more changes and is shut down.

Evolution is Unavoidable

Most of the changes that are demanded in the evolution phase are user-requested extensions as well as modifications and changes in the software environment, and they all transform the behavior of the system. The main difficulty in this analysis is that these changes cannot be anticipated at design time and that predicted changes not always happen [34]. Even more, if the design is made in the anticipation of future extensions, the structure increases its complexity and valuable time is lost, as the extensions may never take place. Therefore, software must be able to address unexpected requirements, when they are really needed, without affecting the ability to respond to new requirements. If the initial architecture is flexible and the introduced changes preserve the structure, the software will not lose its ability to evolve very easily.

If the architecture is inflexible from the start and new changes are introduced in a disorganized way, the code will distance from the design and new changes will be expensive and time consuming. Bad coding practices will appear, like code duplication, lack of modularity, classes with too much functionality or violation of code conventions.

Behavior Preserving Changes

The disorder introduced in the structure by changes that alter the functionality of the system can be reduced by applying different kind of changes that only affect and improve the internal structure. These transformations are called *refactorings* and include restructurings like renaming of entities, moving methods between classes or hierarchy arrangements. Appropriate refactorings can help to increase the legibility and robustness of the system, simplify the insertion of new changes and reduce the cost of expected software evolution.

2.2.2 Definition of Refactoring

Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [26].

The aim of refactoring is to improve the internal structure of the software without adding functionality. Opportune refactorings improve simplicity, understandability and flexibility.

For example by using refactorings duplicated code can be removed, unused parameters or variables can be removed, classes not doing enough can be inlined in others and complex conditionals can be turned into inheritance. In order for refactorings to be really helpful though, they should be automated to avoid introducing new errors. Luckily, some excellent tools already exist that allow these kind of transformations, and much more are being developed and improved.

Before applying a refactoring, the involved code entities must comply to some preconditions. For example, the refactoring *addClass* receives the following parameters: the name of the class that needs to be created, the class that will become the superclass of the new class, and the list of subclasses of the superclass that will now extend the created class. In figure 2.2 we show a concrete example, where the created class `IntermediateCollection` is added to the hierarchy having as subclasses `OrderedCollection` and `LinkedList` that originally were subclasses of `SequenceableCollection`. The preconditions for this refactoring are: the name the class we need to create must not define an existing class, the given superclass must be a class and the list of subclasses must be subclasses of the given superclass. If the assertions defined by the preconditions are satisfied, the refactoring apply the transformations to the code. On the other hand, postconditions specify how the assertions are transformed by the refactoring. Postconditions can be used to reduce the amount of analysis for sequential or composite refactorings or to calculate dependencies between refactorings. These precondition/postcondition definitions were originally formalized in the context of the Refactoring Browser of Smalltalk [46] but they are now being used for a language independent refactoring framework [52].

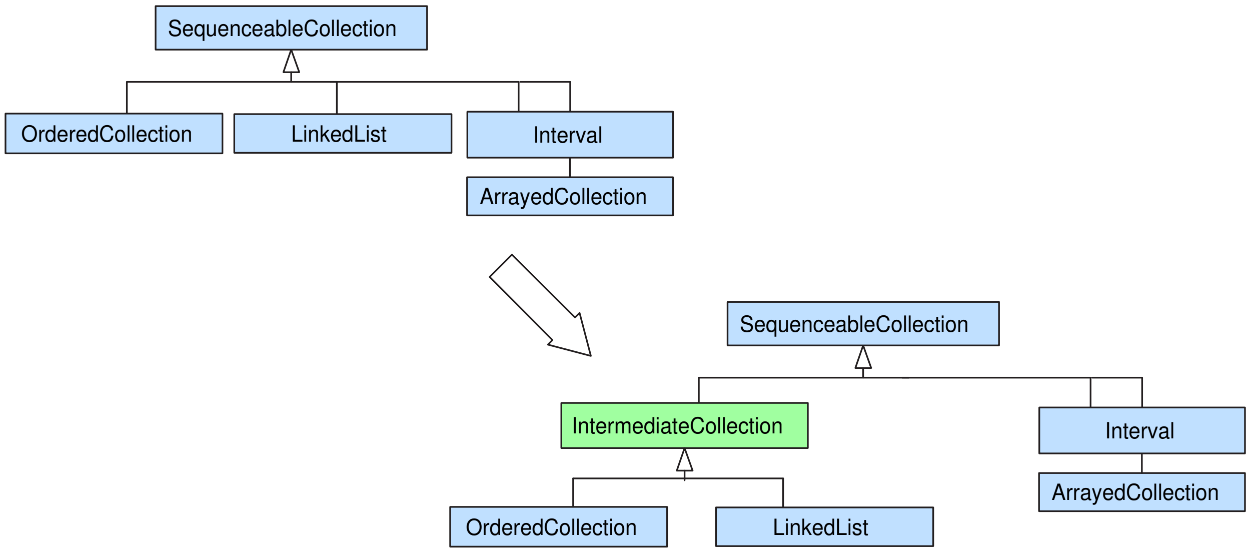


Figure 2.2: The *Add Class* refactoring

2.2.3 Refactoring List

There are several refactoring definitions and implementations in the literature, starting with the initial study performed by Opdyke [40]. In his PhD thesis Opdyke defined several low level refactorings related to creating, deleting, changing and moving code entities such as classes, methods and variables.

The Smalltalk Refactoring Browser's List

Most of Opdyke's refactorings were shown to be behavior preserving and the ideas of Opdyke on refactoring were augmented by Roberts Brant and Johnson [46] to create the Refactoring Browser [45] for Smalltalk. The refactorings supported by the Refactoring Browser are mostly primitive refactorings and composing them into more complex ones is a hard task. The refactorings implemented in the Refactoring Browser so far are the following:

Class Refactorings: add class to the hierarchy, rename class, safely remove class, convert the class to a sibling of its subclasses adding a new class in the hierarchy.

Instance/Class Variable Refactorings: add variable to class, rename variable, remove variable, push up variable, push down variable, create accessors, abstract variable, concrete variable.

Method Refactorings: move to another class, rename method, safely remove method, add parameter, remove parameter, push up method, push down method, inline all the senders within the class.

Code Refactorings: extract method, inline parameter, inline temporary, convert temporary to instance variable, rename temporary, move a temporary to the tightest scope, extract a message to a temporary, inline a message send.

Some Refactorings from Martin Fowler's List

A less formal but more practical approach was taken by Fowler [26] in his Refactoring book. He defines nearly hundred different kinds of code refactorings, some of them oriented only to strongly-typed languages. Each of the refactoring definitions include the motivation of why the refactoring should be performed and illustrates the mechanics of the code transformation. Most of these refactorings have to be performed manually, and only few of them are available in commercial tools. Some useful refactorings that will be referred in the development of this thesis are:

Dealing with Long Methods: extract method, move method to a new class transforming the temporaries into instance variables, replace temporary with a message call.

Simplifying Conditional Expressions: decompose conditionals, introduce null object instead of asking for null values, replace conditional with polymorphism.

Moving Features between Objects: extract class, inline class, move method.

Organizing Data: encapsulate field, replace data value with object.

Making Method Calls Simpler: introduce a class for repeated parameters, preserve whole object, rename method.

Dealing with Generalization: merge classes for collapsing the hierarchy, extract interface, extract subclass, pull up method, replace inheritance with delegation.

2.2.4 When to Apply Refactorings

Now that we have stated the convenience of performing refactorings, it is important to determine *when* to apply these refactorings to the code in the software development process.

Extreme Programming

Refactoring is a fundamental technique to keep design simple practiced by the eXtreme Programming software development process [5]. They propose short iterative development cycles where they separate code extensions and modifications from internal restructurings. Refactorings are applied continuously during the development process in different circumstances like behavioral changes and code reviews.

Every behavioral change is preceded by refactorings oriented to integrate the change properly into the structure. Both the refactorings and the changes are followed by comprehensive testings to ensure that no error was introduced.

Code review is a systematic and disciplined process where careful examination of the code leads to propositions for improvement. Mainly based on common sense and the experience of the reviewers, large or small refactorings may arise.

Many integrated development environments like *Smalltalk VisualWorks* [13], Eclipse [30] and Together Control-Center [15] provide support both for applying refactorings and unit testing, but none of them assists adequately in detecting the places where the refactoring should be applied.

Bad Smells

A complementary approach, which states that code should be refactored when bad coding practices or "bad smells" [26] are encountered, focus on the quality of the reviewed code. Bad smells are considered as warning signs and indicate that refactoring may improve the structure. Different degrees of automation can determine the presence of a bad smell in the code by means of manual inspection, automated code inspection or metrics.

Metrics like cohesion-based distance are used to detect features that should belong together [48]. A tool using these measures proposes refactorings like move method, move feature, extract class and inline class to increase the level of cohesion in the system.

Some tools perform automatic code inspections [31], [15] pointing out the places where the code has bugs or other technical defects like error handling, misuse of operators, unused code or accomplishment to the coding standards. More advanced design issues are not addressed, like the bad smells proposed by Fowler.

Kent Beck and Martin Fowler have developed a list of more advanced design flaws they call bad smells [26]. Their presence in the code indicates that the code should be considered for refactoring. The detection of these smells is described by means of code structures, class interactions, systematic changes performed and sometimes very human perceptions. Here is a description for the smells they identified with a classification proposed in [55]:

Measured Smells: This category include smells that are easy to detect by means of software metrics. *Large Class*, *Long Method*, *Long Parameter List* and *Comments* are smells that belong to this category.

- *Large Class*: Large classes trying to do too much are difficult to understand and maintain.
- *Long Method*: Long methods should be decomposed for clarity and ease of maintenance.
- *Long Parameter List*: Long parameter lists are hard to understand and difficult to use. Instead of passing everything a method needs, consider passing objects.
- *Comments*: Comments may be present in the code because the code is bad.

Duplication: One of the worst smells, where the same code structure is present in more than one place. This duplication can be syntactic or semantic.

Data: Smells related to the usage of data in procedural means. Includes the smells *Data Class*, *Data Clump* and *Primitive Obsession*.

- *Data Class*: Classes with just fields, getters, setters and nothing else.
- *Data Clumps*: Data that is used together together in lots of places like fields in classes or parameters in method signatures should be a class of its own.
- *Primitive Obsession*: A primitive data type can be turned into classes to make it clear what it is for and what sort of operations are allowed on it (like creating a Time or Date class instead of using a couple of integers).

Interfaces: The usage of well designed interfaces helps the client to use them better. Some problems related to this area are the smells *Alternate Classes with Different Interfaces*, *Refused Bequest* and *Incomplete Library Class*.

- *Alternative Classes with Different Interfaces*: Classes that do similar things, but have different names should be modified to share a common protocol.
- *Refused Bequest*: A subclass does not want or need all of the behavior of its base class, maybe the class hierarchy is wrong.
- *Incomplete Library Class*: A library is not doing all it should do.

Responsibility: Unbalanced responsibilities between objects can make code very complex to understand. Therefore identifying smells like *Feature Envy*,*Inappropriate Intimacy*, *Message Chains* and *Middle Man* are important to keep the code clear. Some smells of this category can also be considered as *maintenance smells* [54] and manifest themselves when changing from one version to another. This smells are *Divergent Change*, *Shotgun Surgery* and *Parallel Inheritance*.

- *Feature Envy*: A method of one class seems more interested in the attributes (usually data) of another class than in its own class.
- *Inappropriate Intimacy*: Classes knowing too much about others.
- *Message Chains*: Too many messages in a chain are hard to follow.
- *Middle Man*: A class is acting as a delegate, without performing useful extra work.
- *Divergent Change*: One class commonly suffers many kinds of changes for different reasons, making the system difficult to maintain.
- *Shotgun Surgery*: One kind of change recurrently alters many classes, making it hard to find all the right places that do need changing.
- *Parallel Inheritance Hierarchies*: Whenever a subclass of a class is created, a subclass of another one has created to match. It is a special case of the Shotgun Surgery smell.

Unnecessary Code: Entities that added for future usage but are never required, or a class that have been downsized when refactoring are examples of unnecessary code that should be removed. The specific smells defined are *Speculative Generality*, *Lazy class* and *Temporary Field*.

- *Speculative Generality*: Often parameters, variables or methods are designed to do things that in fact are not required.

- *Lazy Class*: A class that is not doing enough work should be eliminated. It can be a class that was downsized during refactorings, or a class that was added because of changes that were planned but not made.
- *Temporary Field*: Instance variables that are only set sometimes are hard to understand. It is expected that an object needs all its variables.

Conditional Logic: Using conditionals indiscriminately can make the code difficult to understand. The smell *Switch Statements* defines problems from bad usage of polymorphism to poorly defined conditional expressions.

This list addresses some of the most common flaws that can be present in object oriented code, but this list will never be complete. Different projects and domains will need to detect a different set of code smells, for example smells that can occur in unit test code [19].

2.2.5 The Refactoring Process

The refactoring process goes from the identification of where the refactoring should take place to the application of the refactoring. Before actually applying the refactoring, the value of the refactoring should be taken into account. Based on this, the project manager can decide if the proposed restructuring is worth or not in terms of a cost/effect analysis [33].

When the decision to restructure the code has been done, there are some general steps that should be performed to assure the overall contribution of this restructuring:

1. Identify which refactoring(s) should be applied, and where.
2. Assess the quality of the refactorings.
3. Perform the refactorings.
4. Evaluate the effect of the refactorings and check postconditions.

A more detailed description of what happens in each step is shown in figure 2.3, which has been adapted from [33]. The first step of identifying the refactoring possibilities can be performed by detecting bad smells in the code. This provides the place where the refactoring should be applied. By analyzing the context where the bad smell was found, like the features of the surrounding entities, proper refactorings are proposed. In this analysis, a number of possible refactorings may be identified. Their contribution to the quality of the code is variable, and some of them may be harder to apply than others. The value of these refactoring possibilities should be assessed. This allows the project manager to decide which refactoring to apply, if some, taking into account the resources that are available and the benefits that would result. The quality of the refactoring also has to ensure that the refactoring really solves the bad smell and that after applying it, the behavior of the code is preserved. The following step is to determine the details of the refactoring application responsibilities and actually apply it.

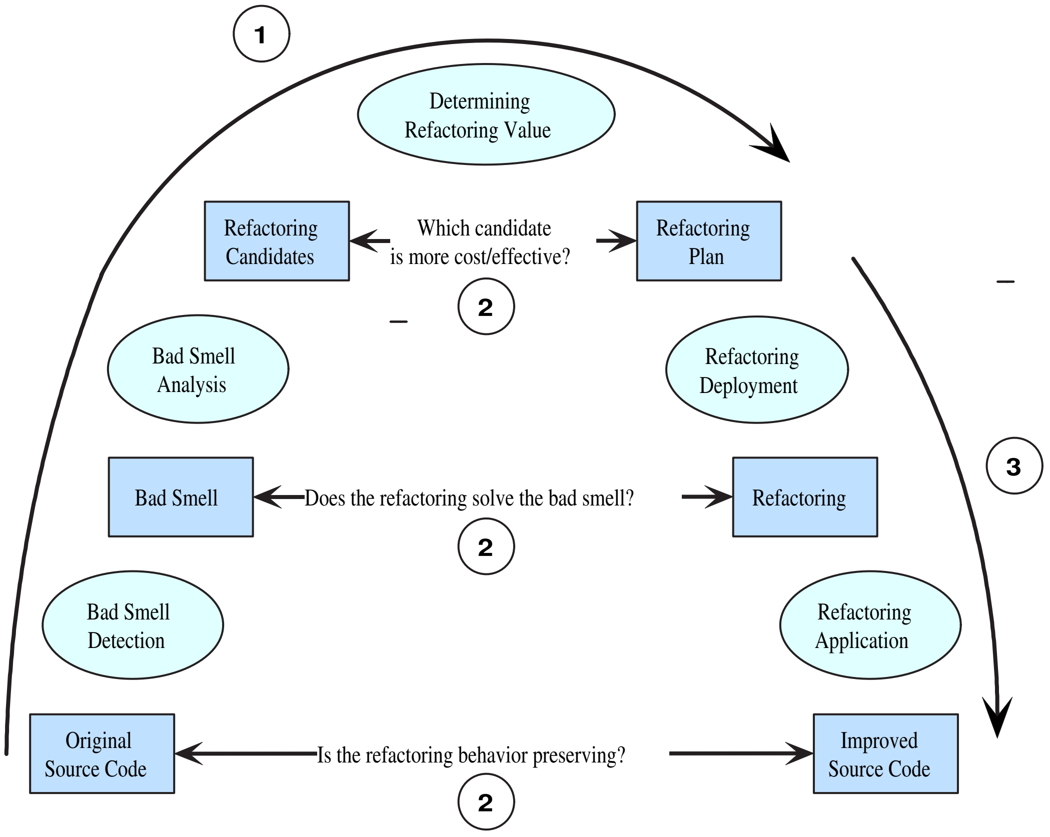


Figure 2.3: The Refactoring Process

Evaluating the Refactoring Effect

This aspect of the process focusses on the increase of quality that the refactoring entails.

[33] proposes to use coupling metrics to determine the improvement in terms of maintainability of the program. This metric only determines the maintainability enhancement of certain refactorings such as extract class, extract method or move method. Other metrics are intended to be used for covering the analysis of more refactorings.

[21] is a more extensive metric-based approach for describing the impact of refactoring on the internal program quality that uses object-oriented metrics such as number of methods, coupling between objects, number of children, cohesion and response for class.

In both cases, the parameters that determine the impact of different refactorings depend on subjective data obtained in experimental manners.

A less formal approach is to weigh the bad smells in terms of how strong they are. If the refactoring removes the detected bad smell, its value can be defined as the difference of smelling before and after applying it. This does not mean that the bad smells have to be considered for the whole system, only for the entities involved in the refactoring.

2.2.6 Refactoring Tools

Even though refactorings can be performed manually, this is an error prone and time consuming activity. The benefits of refactoring will fade away if new errors are introduced or the functionality changes in unpredictable ways. For automating the application of refactorings, research has focused on primitive refactorings [40, 46] that can be completely automated and are provably correct. More complex refactorings can be created by composing primitive ones, resulting in behavior preserving compositions, but tool support for composite refactorings is lacking.

Automated tools for applying refactorings are pretty safe to use in terms of behavior preservation and drastically reduce the time, cost and effort of performing refactorings.

Support for refactorings is provided by a number of tools, for several languages. Some tools available for object oriented languages are described:

Smalltalk's Refactoring Browser

The Refactoring Browser [45, 46] is the standard browser for the VisualWorks Smalltalk integrated development environment (IDE). Originated on the research of Opdyke [40] that was C++ oriented in the beginning, it is until now the most complete refactoring tool available. The total integration with the development environment makes it a very useful and used tool, extremely popular among Smalltalk developers.

The Refactoring Browser has a semi-automatic approach for applying refactorings. This means the user interaction is related to the naming of new entities, as the legibility and consistency of the system is considered fundamental in object oriented programming.

In the provided framework, every Refactoring is associated to a set of preconditions that have to be met before applying the actual transformations. The check for the preconditions and the application of the transformation are methods that can be accessed separately. This is a useful feature for composing refactorings or proposing valid refactorings for a certain bad smell, for example. Although, the Refactoring Browser itself does not proportionate an adequate framework for composing refactorings.

Some of the reasons behind the large amount of refactorings supported by the Refactoring Browser and their reliability are the following:

- The powerful reflective characteristics of Smalltalk. The capability of manipulating directly the entities on the code without using a separate metalanguage simplifies the refactoring definitions. On the other hand, the fact that Smalltalk code uses reflection also makes it more difficult to prove that refactorings are really behaviour-preserving.

- The representation of the program has the form of parse trees that most of Smalltalk implementations provide. Transformations between parse trees are reflected directly in the code without any kind of translation.

Other Tools

Advanced refactoring tools for other object oriented languages exists and are being perfected.

C++ The more complex the language, more difficult it is to define automated refactorings. For example in C++, an intrinsic complex language, this becomes a very hard task, and limited refactoring support is available. Even so there is some tool support like [25].

Java Since Java is a very popular language for developing all kind of applications, it is expected to obtain robust refactoring support from the development environments. Some useful tools are the following:

- IDEA [16] is a complete commercial Java IDE developed by IntelliJ. It is the Java tool that provides several quality refactorings.
- Eclipse [30] is a non-commercial Java IDE, with strong refactoring capabilities.
- TogetherCC [15] is a commercial case tool that perform various refactorings on code and UML diagrams.
- JFactor [29] is a plugin for VisualAge.
- JRefactory [47] is a standalone tool and plugin for Elixir, JBuilder and NetBeans IDE.
- XRefactory [58] is a refactoring plugin for Emacs.

C# Even if C# is a recently created language, it already counts with two refactoring tools:

- C# Refactory [49] is a plugin for Microsoft Visual Studio.NET IDE that provides some useful refactorings.
- C# Refactoring Tool [42] also integrated for Microsoft Visual Studio.NET IDE.

Other Languages Tools for other languages have been developed as well, but not with the same energy as for Java or Smalltalk. Refactoring support is being provided for languages like Python [50] and is planned for languages like Ruby and Perl.

The oldest refactoring tools are not really utile at this point, as the languages are not commercially used. A refactoring tool was developed for Eiffel [12] and Guru is the refactoring tool for Self [38]. Guru is a completely automated refactoring tool, that does not interact with the user.

2.2.7 Conclusion

Software evolution is real and impossible to ignore in the software development process. Refactoring is a technique that helps to maintain the structure of the system while it evolves.

Although it is possible to refactor manually, tool support is considered crucial, as it is necessary that the restructuring is performed fast and safely. A number of tools perform the refactorings selected by the user in an automated way.

The definition of a refactoring process that includes bad smell detection and analysis, and the validation of the possible refactorings, implies that more automated help than the existing can be provided. More support in the detection of bad smells, links with possible refactorings and a measurement for comparing the contribution of the refactorings to the quality of the system is required.

2.3 Related Work

The aim of this work is to provide support for the whole refactoring process through a logic meta programming framework. Related work can be found in the different steps related to the refactoring process and the usage of logic as a mechanism to reason about code.

2.3.1 Logic Based Approaches

The use of logic based query approach is not new in software engineering. For example [14] represents high level structural information like dependency relationships among modules in a Prolog database. [11] propose a reverse engineering tool that stores all the program information in a logic code repository to perform data flow analysis. [41] present an algebraic framework for modeling source code and expressing high level queries over it.

2.3.2 Applying Refactorings

As mentioned, a number of tools provide support for applying refactorings. The most used ones are the Refactoring Browser for Smalltalk, and the Eclipse and IntelliJ IDEA environments for the Java programming language. They all provide a number of simple and safe refactorings that can be selected and applied by the user. None of these tools is language independent, but research is being made in this direction [52], based on the concepts of preconditions/postconditions defined initially for the Refactoring Browser.

The Refactoring Browser's framework provide support for composable refactorings, but the tight integration with the user interface makes this task complicated for regular developers. The separation of the preconditions and the transformations in the Refactoring Browser's framework provides some help for composing refactorings in an independent manner.

The possibility of stating the postconditions of refactorings, would make this task much easier. Studies both for Smalltalk [46] and Java [39] have been done following this approach.

The mentioned tools apply the refactorings directly to the code, but recent research are also focusing on refactoring UML diagrams [2, 51, 7, 8].

2.3.3 Automatic Code Inspection

Code Auditing

Automatic code inspection is performed by a number of tools. The most well known is the C analyzer LINT [31], and its Java variant JLint [1] that points out the places where the code has bugs or other technical defects like error handling, misuse of operators, unused code or accomplishment to the coding standards.

Code Critic is a *Lint*-like tool [31], 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. For example, the *Code Critic* tool also detects the *long method* bad smell that was proposed by Beck and Fowler [26]. The major shortcomings of this tool is that it strongly relies on the Smalltalk environment and its powerful meta programming capabilities.

Borlands Together Control Center [15] is a complete commercial development tool that includes statically auditing code by means of detecting superfluous code, structural weaknesses and prevent common errors.

Generally, auditing tools adopt a very technical perspective, not addressing more advanced design issues, like bad smell detection nor refactoring propositions.

Which Refactoring Should be Applied and Where

The detection of suitable places for applying refactorings is being studied using a number of approaches.

[23] and [32] indicates where refactorings may be applicable by automatically detecting program invariants. An invariant is for example "a temporary variable that holds the value of an expression". This represents a bad smell that is solved with the *Replace temp with query* refactoring. The invariants are discovered by analyzing dynamically the behavior of the application. This means that the inference of invariants depends on the test-suites that were defined. Nevertheless, good results have been obtained in practice, and the dynamic approach appears as complementary to other statical ones.

[48] uses object oriented metrics to statically identify bad smells and propose appropriate refactorings. A tool using cohesion metrics proposes refactorings like move method, move feature, extract class and inline class intended to increase the level of cohesion in the system.

[54] detect bad smells in Java by defining the bad smells as program patterns. Programs are represented as parse trees where nodes are program entities. The smell detection contemplates the presence of relations between code entities or structural patterns present in parse tree. The results are displayed graphically by showing what parts of the system are affected by bad smells and where the concentration of smells is higher.

[22] also represents the program as a parse tree, and furthermore, provides support for applying refactorings in the affected parts. When bad smells are detected, the parse tree is analyzed, specially the parts that present the bad smell and the tool proposes a number of appropriate refactorings to the user for application.

2.3.4 Finding Code Duplication

There are different approaches for detecting duplicated code in software systems. Some of them are based on syntax tree information, while others take into account the text of the source code:

[35] uses the syntactical structure for calculating metrics over entities, expressions and control flow of functions. Two functions are considered as clones if their metrics are similar. This approach only detects similar methods, but not similar segments of code.

[4] gives a text based approach that detects pairs of longest textual matches of sections of code. These matches contemplate sections of code that are textually identical except for the consistent usage of constants, variable names and others.

[43] presents a language independent 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 that looks for specific patterns in a comparison from every line to every other. The refactorings that are proposed consist of removing duplicated methods, extracting duplicated code from within a method and inserting an intermediate subclass to factor out the common code.

2.3.5 Assessing the Impact of Refactorings

2.3.6 Supporting the Whole Development Process

[33] implements the Refactoring Assistant tool, oriented to a comprehensive support for performing the refactoring process, from bad smell detection to the appliance of the refactoring. They propose to assess the quality of the refactoring before applying it in several levels. At the management level, the contribution of the possible refactorings that solve a bad smell should

be stated for being taken into account together with the time/cost resources that are available. This would lead to an informed decision of which refactoring is most convenient to apply, if the requirements of the project allow to do so.

A more technical approach is taken by [22], who analyzes the presence of bad smells in a the program with a parse tree representation. Based on this analysis they propose proper refactorings, letting the user decide which one to apply.

2.3.7 Conclusion

While several tools are implemented, there is no single tool that supports all necessary steps of the refactoring process. Even if we do not intend to cover all these steps, we will use logic meta programming as a mechanism for detecting bad smells, performing analysis on found bad smells or proposing refactorings for solving them. The ability to query and modify the structure from a declarative point of view, simplifies the legibility and extensibility of the proposed framework.

Chapter 3

Refactoring Process Using Logic Meta Programming

3.1 Introduction

As we have seen in the previous chapter, logic meta programming is a well-suited technique for reasoning about the code at a high level of abstraction. In the context of this thesis, we use logic meta programming for providing support in various steps of the refactoring process.

The refactoring process involves much more analysis and decisions than just applying the refactoring to the code. This includes deciding *when* to apply a refactoring, detecting *where* to apply it, *which* refactoring should be applied, assessing the contribution of possible refactorings, evaluating the effect of the refactorings and checking the valid postconditions after the refactoring has been applied.

Some of these steps are difficult to automate, like the decision of *when* to apply a refactoring, which depends heavily human criteria, and the common advice is to refactor whenever it is necessary. Other complicated phase to achieve is assessing the effect of the refactorings before applying them. This involves having very accurate definitions of the postconditions of the refactorings, a situation that is not yet achieved by refactoring tools.

This chapter explains how logic meta programming is used for supporting some automatable steps of the refactoring process. We explain in how bad smells are detected, how refactorings are proposed and how we perform the actual application of existing and composed refactorings from the Refactoring Browser.

Finally we describe the tool that makes this support practical.

3.1.1 Supported Refactoring Steps

The steps of the refactoring process we will tackle are the following:

1. Detect *where* to apply refactorings by recognizing bad smells in the code, and weighing these bad smells in order of importance.
2. Help the user to decide *which* refactoring to apply to resolve the bad smell by analyzing the entities involved in the bad smell and proposing a number of refactorings to solve it. In this step, the entities that are used as the parameters of the refactorings are identified.
3. Applying the selected refactorings with the necessary information. This is achieved by integrating our tool with an existing implementation provided by the Refactoring Browser, which transforms the code efficiently, preserving the behavior of the system.

The third step could merely be a link that passes the relevant entities as parameters to an existing refactoring framework. The difficulty is that the refactorings Martin Fowler suggests to solve the bad smells, are not directly supported by any tool so far. Therefore, our approach includes the extension and composition of basic behavior preserving refactorings through logic predicates.

Regarding the kind of code analysis that will be performed in the distinct steps, all logic relations we define depend on static characteristics. This means the system will not be executed for determining the runtime behavior, and will only be analyzed statically.

Figure 3.1 shows the interrelation between the different areas we support of the refactoring process:

The detection of bad smells generates two relevant results, the indicators of how strong the smell is (for example, the amount of duplicated statements) and the relevant entities related to the place where the bad smell was detected. The indicators are used to weigh the bad smells so the user can assess their relative importance. In order to propose proper refactorings with the necessary parameters, our tool analyzes the related entities of the bad smells.

3.1.2 Environment

The logic meta programming language we are using is *SOUL*, which used with the library *LiCoR* allows reasoning over *Smalltalk* code at a high level of abstraction. Even if most of the logic predicates are language independent, some of them describe language specific relation like the predicate `keywordsOfMethod`, which takes into account the specific syntax Smalltalk uses for identifying the parameters a method receives. For example in the method definition `at: index put: anObject`, the keywords are `at:` and `put:` and `index` and `anObject` are the parameters of the method.

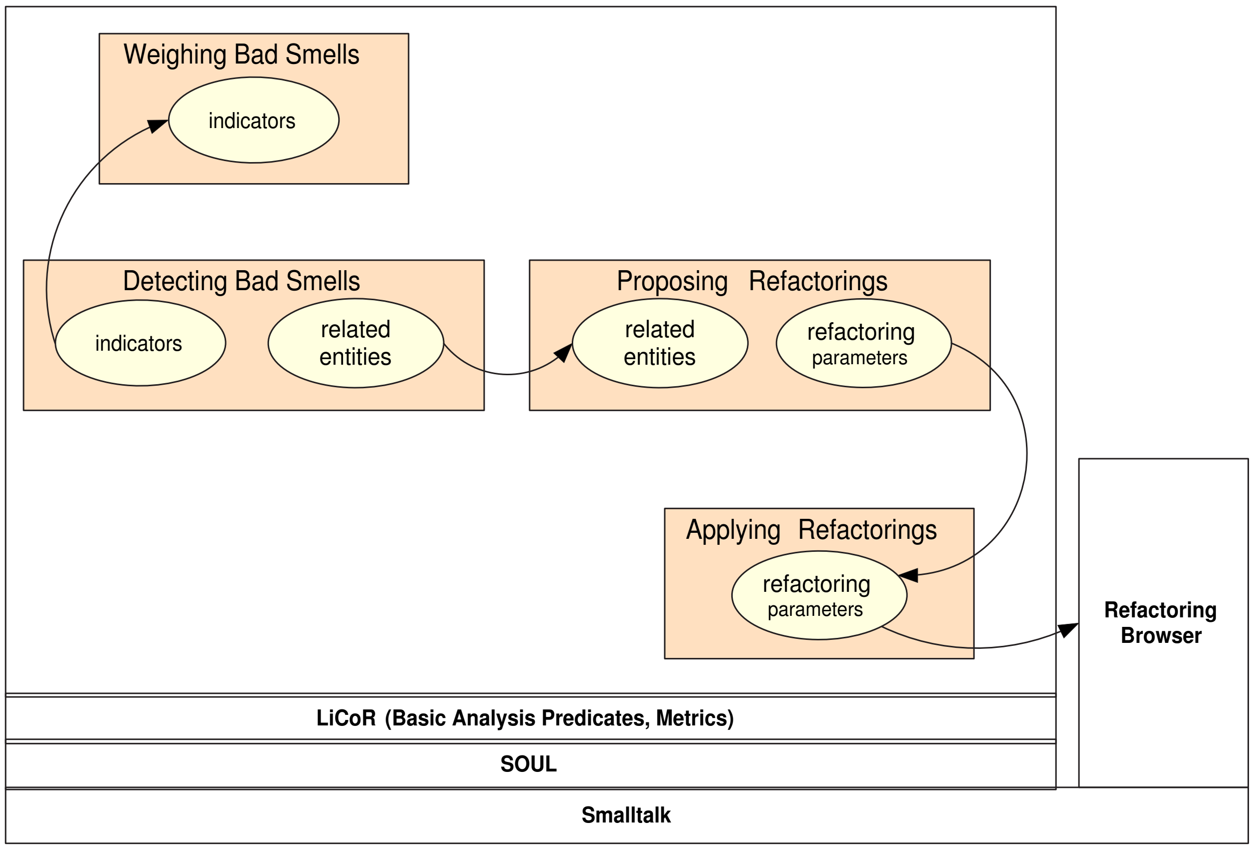


Figure 3.1: Supported steps in the refactoring process: detecting and weighing bad smells, proposing refactorings and applying refactorings

For applying the proposed refactorings, we integrate our framework with the Refactoring Browser, which provides a number of useful but rather primitive refactoring implementations. Although the refactorings proposed by Martin Fowler can not be applied directly by the Refactoring Browser, we have extended and composed the available refactorings by means of logic predicates. This solution increases the language independence of our approach, as these primitive refactorings are pretty standard and are starting to be supported by many other tools.

3.1.3 Basic Analysis Predicates

In order to support the mentioned steps of the refactoring process, we will establish a flexible framework using the basic relations provided by SOUL for reasoning between code entities as seen in figure 3.1.

The LiCoR library contains a number of rules that describe basic characteristics and relationships between classes, methods, interfaces, variables, symbols and other code entities. The most basic ones are defined using the meta level interface, that strongly relies on the reflective capabilities of the base language, like the ones mentioned in table 2.1. Other group of basic predicates are the ones related to the characteristics of methods, mentioned in table 2.2. None of these predicates are very useful by themselves, however they give the basis for generating more interesting results. Some examples of existing intermediate level predicates that we will use for helping to detect and analyze bad smells are the following:

- `accessor(?class,?selector,?variable)`, checks whether `?selector` defined in `?class` is an accessor used to get the value of `?variable`.
- `mutator(?class,?selector,?variable)`, checks whether `?selector` defined in `?class` is a mutator that sets the value of `?variable`.
- `overridingSelector(?class,?selector)`, checks whether `?selector` is overriding a method definition in the hierarchy of `?class`.
- `highestOverridingSelector(?class,?selector,?superclass)`, checks whether `?superclass` is the highest class that defines `?selector`, which is being overridden in `?class`.
- `selectorUsesParameter(?class,?selector,?parameter)`, checks whether `?selector` defined in `?class` uses at least one time the parameter variable `?parameter`.
- `methodWithConditional(?method,?receiver,?message,<?trueBlock,?falseBlock>)`, checks whether `?method` contains a conditional message `?message` (such as `ifTrue:ifFalse:` in the case of Smalltalk) send to `?receiver`, with arguments `?trueBlock` and `?falseBlock`.

Metrics

LiCoR also provides a category of logic queries that compute object-oriented metrics. These are used to help detecting and analyzing bad smells. Sometimes they are used to find places in the code that are worthy of further investigation because they are likely candidates for bad smells [48]. Once these locations in the source code have been identified, we can use our other queries to analyze these parts of the program in more detail.

Most of the defined metrics are easy to implement as they mostly count the number of results of more complex predicates. The format of the predicate is `metric(?metricName,?entity,?number)`, where `?metricName` is the name of the metric, `?entity` is the entity that is being analyzed and `?number` is the value of the metric. A simple example is the metric that computes the Number Of Statements of a method (NOS):

```
metric(NOS, primitive, method(?method), ?number) if
    statementsOfMethod(?statementList,?method),
    length(?statementList,?number)
```

Other metrics defined in LiCoR we use are: Number Of Implemented Methods (NOIM) in a class, Number Of Instance Variables (NOIV) in class, Number Of Temporaries of Method (NOTM) and Number Of Parameters of Method (NOPM).

3.2 Detecting Bad Smells

For detecting the place where refactorings can be applied, we are detecting bad smells in the code. These bad smells indicate possible bad design and coding practices, like duplicated code, unused parameters or classes that delegate the majority of their functionality to other classes.

In the Refactoring Book [26] Martin Fowler gives a list of 22 bad smells, which address common flaws that can be present in object oriented code. These smells are originally presented in terms of human intuition, but many of them can be formalized for automated recognition by verifying structural characteristics and relations present in the code. Specifically, we cover eight of these smell completely and five of them partially.

This section explains our approach for detecting the bad smells described in section 2.2.4, recount which bad smells are detected and which are not, describes in detail the smells *Unused Parameter*, *Parameter Clump*, *Inappropriate Interface* and *Duplicated Code* in terms of logic predicates and gives illustrative examples. The rest of the smells are explained in the appendix A.

The concrete examples given in the following sections and in the appendix are taken from the experiments we performed over the Smalltalk `Collection` library, the `SOUL` application and the `HotDraw` framework. More details are given in chapter 4 Experiments.

3.2.1 Our Approach for Detecting Bad Smells

We propose to define as many bad smells as possible in terms of logic predicates to be applied on code entities. However, there are three kinds of smell we will not address. Smells that are too fuzzy to be defined in automated means, smells related to maintenance, smells that need static typing information and smells that need runtime information. Nine of the 22 bad smells defined by Fowler satisfy these criteria. We also defined a new bad smell, *Inappropriate Interfaces* that is explained in section 3.2.6.

Our bad smell predicates are applied on classes or methods and return the related entities that will be used in later steps of the refactoring process. They also return indicators associated to the smell that can be used for weighing the smell more properly. For example the *Duplicated Code* smell returns all the entities related to the duplication, and also the indicators of the duplication, that in this case consist on the number of duplicated statements and the amount of places where the duplication was found.

In general, the decision of which bad smells we should consider, is intimately related to the kind of application we are analyzing. For example different users will consider different threshold values for the smell *Too Many Statements* or the minimum length that a *Message Chain* smell should have. For tuning the smells depending on the usage, we include user configurable thresholds in our tool.

With the assistance of a tool, a number of smells can be detected at the same time. It is important for the user to categorize them properly, in order to solve the most important smells first. A classification of the relative importance between bad smells is provided, configurable by the user to reflect his experience and personal considerations on the matter. This classification can be integrated with the parameters returned by the bad smells in order to weigh the smells more properly.

3.2.2 What Smells are Detected?

Tables 3.1 and 3.2 give a comprehensive list of the smells defined by Martin Fowler and show whether they are covered in our work or not. The first column represents Fowler's bad smells, and the second column describe the cases present in these smells. For example *Large Class* includes the cases *Too Many Methods* and *Too Many Instance Variables*. Usually different cases of the same smell have different solutions, so we will address them separately. We extended the list with two bad smells. *Inappropriate Interface* between a class and its subclasses, and a case of the speculative generality smell, *Abstract Method Not Implemented* in the hierarchy. In total we address nine bad smells completely (including our addition) and five bad smells partially.

Even if we intend to detect as many bad smells as possible, there are some that are out of the scope of this work. We are not addressing smells we consider too fuzzy to be defined by automated means, smells that need dynamic runtime information, smells related to changes through different versions, that are considered as "maintenance smells" [54], and finally, we are not addressing smells that need static typing information.

- **Fuzzy Smells**

  Unhelpful Comments. The classification of unhelpful is pretty fuzzy, and the possibility of counting the comments of the code is not possible as SOUL does not consider comments as valid entities.

  Alternate Classes with Different Interfaces. Related to classes that do "similar" things, with different interfaces. The definition of "similar" in this case is also pretty fuzzy.

  Unnecessary Delegation, part of the *Speculative Generality* smell. It is hard to determine automatically when a delegation is necessary or not.

- **Dynamic Smells**

  Semantic Duplication, part of the *Code Duplication* smell. This refers to methods that have the same effect by using different algorithms. This is extremely difficult to determine by static means. However, this small can be partially addressed by using

	Measured Smells		
1)	Large Class	Too Many Instance Variables Too Many Methods	detected detected
2)	Long Method	Too Many Statements Too Many Temporaries	detected detected
3)	Long Parameter List	Too Many Parameters Redundant Parameter Parameters are Values from an Object	detected detected not detected (types)
4)	Comments	Unhelpful Comments	not detected (fuzzy)
	Duplication		
5)	Duplicated Code	Magic Numbers Duplicated Code in Methods Semantic Duplication	detected detected not detected (dynamic)
	Data		
6)	Data Class	Data Class	detected
7)	Data Clump	Parameter Clump Instance Variable Clump	detected not detected (types)
8)	Primitive Obsession	Primitive Obsession	not detected (types)
	Interfaces		
9)	Refused Bequest	Refused Interfaces Refused Inheritance	detected detected
10)	*Inappropriate Interface*	*Inappropriate Interface*	detected
11)	Different Interfaces	Alternate Classes with Different Interfaces	not detected (fuzzy)
12)	Incomplete Library	Incomplete Library Class	not detected (dynamic)

Table 3.1: (table 1 of 2). Detected Bad Smells in the following categories: Measured Smells, Duplication, Data and Interfaces. The smell *Inappropriate Interface* was added to the category Interfaces

Responsibility		
13) Feature Envy	Feature Envy	detected
14) Inappropriate Intimacy	Inappropriate Intimacy	not detected (types)
15) Message Chain	Message Chain	detected
16) Middle Man	Middle Man	detected
17) Parallel Inheritance	Parallel Inheritance Hierarchies	not detected (changes)
18) Divergent Change	Divergent Change	not detected (changes)
19) Shotgun Surgery	Shotgun Surgery	not detected (changes)
Unnecessary Code		
20) Lazy Class	Lazy Class	detected
21) Speculative Generality	Unused Parameter Unused Instance Variable *Abstract Method Not Implemented* Unnecessary Delegation Odd Name	detected detected detected not detected (fuzzy) detected
22) Temporary Field	Temporary Field	not detected (fuzzy)
Conditional Logic		
22) Switch Statements	Switch on Type Codes Switch Statement over Parameter Switch with Nil Checking	not detected (types) detected detected

Table 3.2: (table 2 of 2). Detected Bad Smells in the following categories: Responsibility, Unnecessary Code and Conditional Logic. The case *Abstract Method Not Implemented* was added to the smell *Speculative Generality*

control flow analysis and data flow analysis for determining the runtime behavior of the system.

Incomplete Library Class. Refers to a library that is missing behavior. This smell needs to define which classes are library classes and can only be triggered when the user tries to add a method to these classes. This dynamic behavior makes it complicated to implement.

- **Maintenance Smells**

  Shotgun Surgery. One kind of change recurrently alters many classes, making it hard to find all the right places that do need changing.

  Parallel Inheritance Hierarchies. Whenever a subclass of a class is created, a subclass of another one has to be created to match. It is a special case of the smell *Shotgun Surgery.*

  Divergent Changes. One class commonly suffers many kinds of changes for different reasons, making the system difficult to maintain.

- **Typing Smells**

  Primitive Obsession. A primitive data type can be turned into a class to make it clear what it is for and what sort of operations are allowed on it (like creating a *Date* or *Time* class instead of using a couple of integers). As Smalltalk does not use data types, this smell is not detectable without using some sort of type inference engine.

  Switch Statement on type codes, part of the *Switch Statements* smell. For detecting this smell, we also need typing information we do not have. In this case for recognizing different switch statements over the same type code.

  Inappropriate Intimacy. Classes knowing too much about others are coupled. For knowing statically to whom the messages are sent or which class is being called we need typing information.

  Instance Variable Clumps. A group of instance variables of the same type appears in several classes.

  Parameters that are Values from an Object, part of the *Long Parameter List* smell. A group of parameters are generated from the same object and sent as parameter in all calls of the method. To detect that it is the same object that is the source of the parameters in every case, we need typing information.

Considering all bad smells and their cases, we cover completely nine bad smells and we cover partially five bad smells out of 23 (including our added smell, *Inappropriate Interface*). Taking into account the different cases of each smell (listed in the second column of tables 3.1 and 3.2), we address 23 out of 37.

3.2.3 How are the Bad Smells Detected?

For actually detecting the bad smells, we use a generic predicate with the form:

```
badSmell(?smellName, ?entity, ?relatedEntities, ?smellIndicators)
```

Where `?smellName` is the name of the smell, that can be simple (e.g. <`dataClass`>) or a list (e.g. <`speculativeGenerality,unusedParameter`>). The detection of the smell usually relies on more specific predicates that receive threshold values and find structural flaws in the code. The `?entity` corresponds to a class or a method that is being checked for bad smells. The `?relatedEntities` is a list that contains the relevant entities related to the bad smell that are analyzed for possible refactorings. Finally, the `?smellIndicators` is a list with measures related to the smell. For example the ratio of envy in the *Feature Envy* smell or the number of duplicated statements and places of duplication in the *Duplicated Code* smell.

For example, the case *Too Many Instance Variables* of the smell *Large Class* is defined in the following way:

```
badSmell(<largeClass,tooManyInstanceVariables>, ?class, <?class>, <?
   numberInstanceVariable>) if
    userThreshold(<largeClass,tooManyInstanceVariables>,
       numberOfInstanceVariables, ?thresholdNumber),
    classWithMoreInstanceVariablesThan(?class,?numberInstanceVariable,
       ?thresholdNumber).
```

In most cases there is an underlying predicate that does the “real work”, like `classWithMore-InstanceVariablesThan` does in this case. The `badSmell` predicate is mostly used for retrieving threshold values and handling the bad smells in a uniform way through our framework.

In the following sections we explain in detail how to detect the smells *Unused Parameter*, *Parameter Clumps*, *Inappropriate Interfaces* and *Duplicated Code*. The examples are taken from the experiments we performed over the Smalltalk `Collection` library, the `SOUL` application and the `HotDraw` framework.

3.2.4 Recognizing Unused Parameters with Logic Predicates

The *Unused Parameter* bad smell is a case of the more general bad smell *Speculative Generality*. A method defines an unused parameter if it defines a formal parameter in its signature that is not used in its implementation. In dynamically typed languages, all selectors with the same signature are equivalent and polymorphic, so actually we need to check that none of the implementations sharing the same signature uses the parameter. Given a particular class, the developer does not know in advance which method implementation he has to inspect, or which parameter is likely to be unused. Even if he detects manually a parameter that is not being used in a specific method body, he has to examine all other implementations of methods with the same signature for assuring that the parameter is not being used.

We will first show an example of the occurrence of an unused parameter and afterwards explain how this situation can be detected automatically by using logic meta-programming predicates. Finally we describe the generic bad smell predicate we use for detecting this smell.

Example

Figure 3.2 shows an example of the bad smell *Unused Parameter*. The class `PosVariable` in the SOUL application, defines the selector `unifyWithUnderscoreVariable: inEnv: myIndex: hisIndex: inSource:`, where none of the parameters is used in the body of the method nor in the other implementation of the selector present in its superclass `AbstractTerm`.

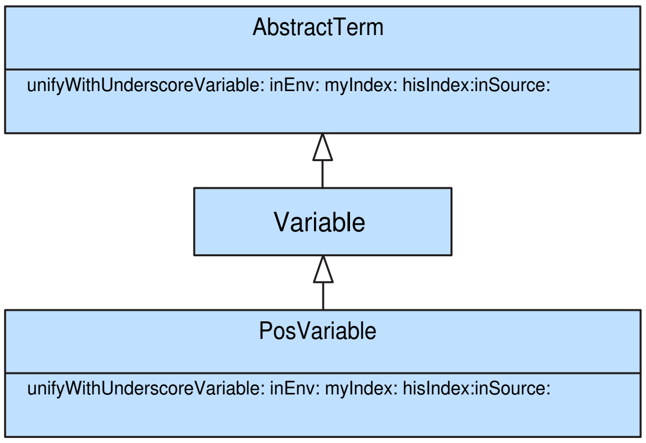

AbstractTerm
unifyWithUnderscoreVariable: inEnv: myIndex: hisIndex:inSource:
Variable
PosVariable
unifyWithUnderscoreVariable: inEnv: myIndex: hisIndex:inSource:

Figure 3.2: *Unused Parameter* in the hierarchy of `AbstractTerm` in the SOUL application

Detecting the Bad Smell

For detecting whether a formal parameter is not used by a method, we check that none of the methods sharing the same signature uses this parameter, including the method itself.

We use the following two logic rules to implement such an algorithm:

```
unusedParameter(?class,?selector,?keyword,?numberNotUsed) if
    methodWithNameInClass(?method,?selector,?class),
    keywordOfMethod(<?keyword,?parameter>,?method),
    classesWithSelector(?classList,?selector),
    forall(
      member(?otherClass,?classList),
         not(selectorUsesParameter(?otherClass,?selector,?keyword))),
    length(?classList,?numberNotUsed)
```

Where the `keywordsOfMethod` predicate retrieves the pair (keyword,parameter) of the method. This predicate takes into account the specific syntax Smalltalk uses for identifying the parameters a method receives. For example in the method definition `unifyWithUnderscoreVariable:aTerm inEnv:anEnv myIndex:myIndex hisIndex:hisIndex inSource:inSource` of the `PosVariable` class, `unifyWithUnderscoreVariable:`, `inEnv:`, `myIndex:` and `hisIndex:` are the keywords, and `aTerm`, `anEnv`, `myIndex`, `hisIndex` and `inSource` are the parameters of the method.

The `classWithSelector` predicate retrieves all the classes that contains the given selector, and `selectorUsesParameter` predicate is used for checking that all those classes do not use the keyword in question. The number of times that the parameter is not used is the same as the amount of classes that contains the parameter, so this value is returned for weighing purposes. Retrieving all the classes that contain a given selector is a requirement for dynamic typed languages. In the typed language version, we traverse to the highest class that implements the selector and retrieve the whole hierarchy for a posteriori checking of parameter usage.

```
selectorUsesParameter(?class,?selector,?keyword) if
    methodWithNameInClass(?method,?selector,?class),
    keywordOfMethod(<?keyword,?parameter>,?method),
    methodUsesVariable(?method,?parameter)
```

The second predicate defines when a parameter is used or not by a selector. After determining the name of the parameter in the specified selector, we only have to check if the method uses the variable.

Bad Smell Predicate

The bad smell predicate for detecting this bad smell is the following:

```
badSmell(<speculativeGenerality,unusedParameter>, ?class, <?selector,?keyword>
, <?numberNotUsed>)
```

Where `?numberNotUsed` corresponds to all definitions of the selector that does not use the parameter. In this case we do not return the `?class` where the unused parameter was found as part of the related entities list. And instead of returning the exact parameter that is the value the keyword takes in the specific class, we return the keyword, that is common to all selectors. The objective of this, is that for every class that implements the selector with the unused parameter we obtain the same result. If we ask for the bad smell in the entire system, we will get an identical duplicated result list for every class that implements the selector, which are easy to discard before presenting them to the user.

3.2.5 Recognizing Parameter Clumps with Logic Predicates

This bad smell is part of the more general smell *Data Clump.* A parameter clump is a group of parameters with no logical object, but that naturally belong together. For detecting it in a specific method, the developer has to choose a particular combination of keywords, and detect if this combination is present is any other methods in the system. Detecting this smell manually is a very hard task, as it involves a large number of possibilities, and requires the inspection of all the signatures of the system.

For detecting this bad smell we use a heuristic, since formally we should detect groups of parameters of the same type, and this information is lacking in our dynamically typed environment. Our searches for parameter clumps are based on the name of the keywords found in the selectors.

We show two illustrative examples of parameter clumps, and afterwards we explain how to detect this smell using logic meta-programming predicates, and explain the generic bad smell predicate.

Examples

The first example was found in the Collection hierarchy. The class `ByteArray` contains the following clump of eleven keywords:

```
width:atX:y:from:stride:width:atX:y:width:height:rule:
```

It appears in nine selectors, all starting with the prefix `copyBits` or `tileBits`, and stored in the protocols `byte processing` or `private`.

The second example was found in the SOUL application. The clump `inEnv:myIndex:hisIndex:-inSource:` repeated 73 times in the hierarchy of the class `AbstractTerm`, and 96 times taking into account the whole application. They all appeared in selectors starting with the prefix `unifyWith` and belongs to the protocol `unifying`. This may indicate that the parameters are used in a consistent way in the found selectors. Figure 3.3 illustrates the appearance of the clump in the hierarchy.

Detecting the Bad Smell

For detecting data clumps in the system, the specific steps we perform are the following:

1. Retrieve all the methods of the given class that have more parameters than a user-defined threshold value.
2. Compute all subsets of these methods, with size larger than a user-defined threshold value.

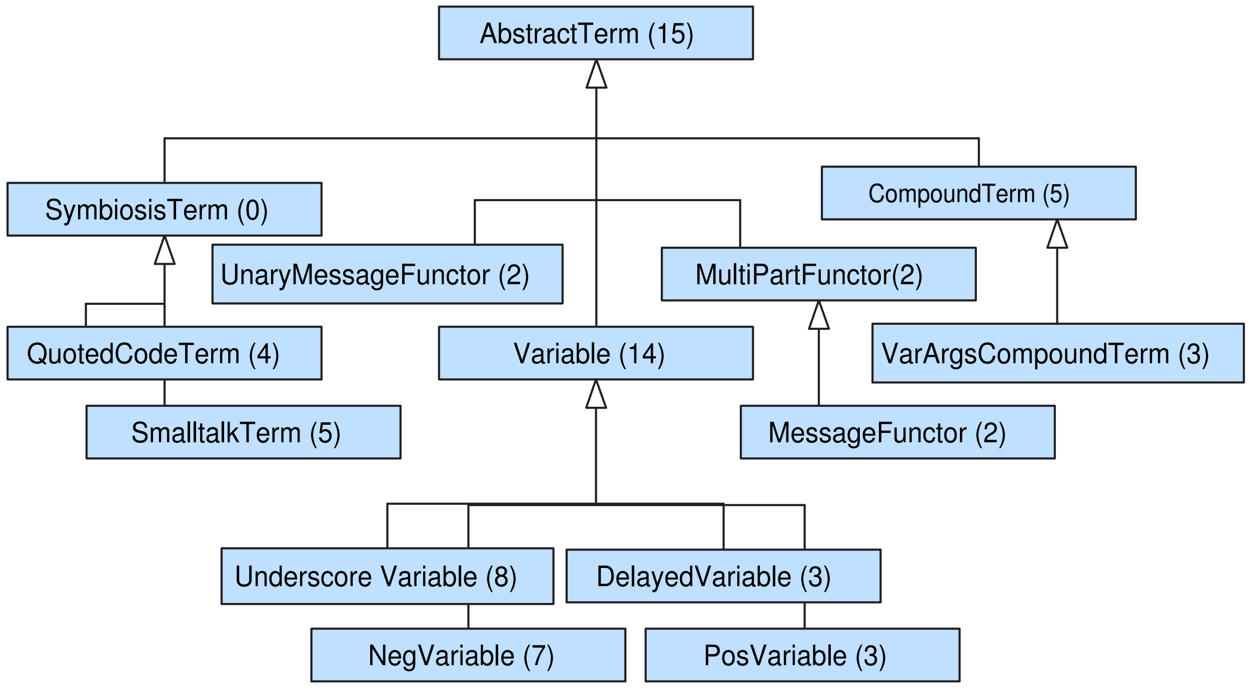


Figure 3.3: *Parameter Clump* `inEnv:myIndex:hisIndex:inSource` in the hierarchy of `AbstractTerm`

3. For each of these subsets, retrieve the keywords and compute the intersection between the keywords of the methods.

4. Match the detected data clump with the other selectors in the hierarchy.

Note that this algorithm grows exponentially with the number of methods because of the subset calculation. The usage of the threshold values inside the algorithm helps to reduce the calculation time and retrieve less but more interesting results. Another possible solution that reduces the amount of computation is a heuristic based on searching parameter clumps only in methods belonging to the same protocol.

The two most important predicates we use for detecting this bad smell are `methodsSharingKeywords` and `commonArgumentsLargerThan`:

```
methodsSharingKeywords(?allMethods,?commonKeywords,?subset,?
   methodThreshold,?keywordThreshold) if
    subset(?subset,?allMethods),
    length(?subset,?length),
    greaterOrEqual(?length,?methodThreshold),
    commonArgumentsLargerThan(?subset,?commonKeywords,
        ?keywordThreshold)
```

This predicate receives a list with the methods that are being considered in the class (all methods with more parameters than threshold), and generates all the subsets of this methods, using the auxiliary predicate `subset`. If the subset is larger than the threshold value, we call the predicate `commonArgumentsLargerThan`.

```
commonArgumentsLargerThan(+?methods,-?commonKeywords,+?
   keywordThreshold) if
    findall(?keywords,
        and(member(?method,?methods),
            onlyKeywordsOfMethod(?keywords,?method)),?keywordList),
    intersection(?keywordList,?commonKeywords),
    length(?commonKeywords,?length),
    greaterOrEqual(?length,?keywordThreshold)
```

This predicate retrieves all the keywords of the given list of methods, and calculates the intersection. If the amount of common keywords present in the intersection is larger than the threshold value, the predicate returns the common keywords.

Bad Smell Predicate

The bad smell predicate for detecting parameter clumps is the following:

`badSmell(<dataClump,parameterClump>, ?class, <?keywordList,?selectorList>, <?numberSelectors,?sizeClump>)`

Like in the *Unused Parameter* smell, we do not return the `?class` as part of the result list. The related entities are the list of the keywords that conform the parameter clump, and the list of selectors where this clump is present. This means that for every class that contains the parameter clump we obtain the same result in a global search for smells.

3.2.6 Recognizing Inappropriate Interfaces with Logic Predicates

The consistency of the interfaces is extremely important when designing flexible and reusable object-oriented systems, as discussed in [44]. This smell consists on inconsistent interfaces between a group of subclasses and its superclass. This means a group of subclasses have some methods in common that are not provided by the superclass.

Detecting manually the inappropriate interface in a particular class is a complex task to perform. The developer has to analyze an entire class hierarchy, and the interfaces it defines. All the combinations of subclasses and interfaces they possibly share must be taken into account, which is a very complicated task to perform without automated assistance.

In what follows we will show an example of the occurrence of an inappropriate interface, and afterwards we will discuss how we recognize the problem by using logic predicates and describe the generic bad smell predicate that detects the bad smell.

Example

Figure 3.4 shows how the classes `OrderedCollection` and `LinkedList` declare the methods `addFirst:`, `addLast:`, `removeFirst` and `removeLast`, which are not present in the other siblings nor in the superclass `SequenceableCollection`.

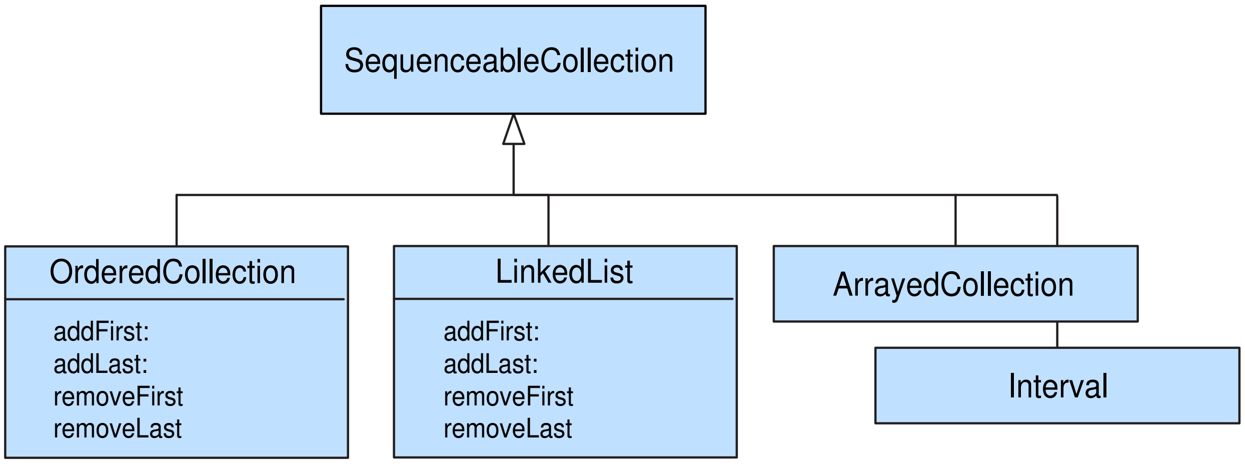


Figure 3.4: Inappropriate Interface in the hierarchy of `SequenceableCollection`

This situation generates confusions 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 methods `addFirst:`, `addLast:`, `removeFirst:` and `removeLast:` in its interface.
- If a developer wants to extend the hierarchy of `sequenceableCollection`, it is unclear whether the new class should contain the mentioned methods in its interface or not.

Detecting the Bad Smell

We use an algorithm similar to the one used in the smell *Parameter Clump* for detecting the problem of inappropriate interfaces in a hierarchy of classes:

- Retrieve all direct subclasses of the given class.
- 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. Here we only take into account only the methods that are not overriding other in the hierarchy, because we are interested in interfaces not defined in upper hierarchies.

Like the algorithm in *Parameter Clumps*, this The amount of subclasses taken into account, as well as the size of the computed interface depends on user configurable values.

A predicate that is different from the ones presented in the smell *Parameter Clumps* is the one that retrieves the interface of the class which is not overriding methods defined higher in the hierarchy.

```
notOverridingClassInterface(?class,?interface) if
    findall(?selector,and(methodWithNameInClass(?,?selector,?class),
               not(overridingSelector(?class,?selector)) ),
                   ?interface)
```

This predicate retrieves all the selectors of a class that are not overriding methods.

Bad Smell Predicate

The generic bad smell predicate we use for detecting this smell is the following:

`badSmell(<inappropriateInterface>, ?class, <?class,?interface,?subclasses>, <?interfaceLength,?numberSubclasses>)`

The indicators in this case are the length of the shared interface of the subclasses, and the amount of subclasses that share this interface.

3.2.7 Recognizing Duplicated Code with Logic Predicates

Code duplication is one of the worst smells possible, as it has many negative consequences on maintainability, bug fixing and readability in general.

For manually detecting duplicated code segments in the system, the developer has to examine carefully the implementation of a number of methods, making it a very time-consuming and impractical task.

Different approaches for detecting duplicated code were given in the *Related Work* section 2.3. Our approach for detecting similar code segments is to traverse the parse trees of two given methods and check that the structures are equivalent rather than the same. Even if it is possible to run the predicate over any pair of methods, we foresee which cases are more useful in future refactorings. These cases are duplicated code in the same class, duplicated code in sibling classes and duplicated code between a class and its subclasses.

Example

In the sibling classes from the SOUL application, `TermSequence` and `MultiPartFunctor` which are subclasses of `AbstractTerm`, we found duplicated code in the method `printForCompileFor:`. The equivalent segments of code were found at the beginning of the bodies, and the only structural difference is present in the last statement. The code considered duplicated is the following (method declarations are only included for clarity):

```
printForCompileOn: aStream

aStream write: self class name , ' keywords: (OrderedCollection new '.
self keywords do:
        [:keyword |
        aStream write: 'add:('.
        keyword printForCompileOn: aStream.
        aStream write: ')']
    separatedBy: [aStream write: ';'].
```

```
printForCompileOn: aStream

aStream write: self class name , ' terms: (OrderedCollection new '.
self terms do:
    [:currentClause |
    aStream write: 'add:('.
    currentClause printForCompileOn: aStream.
    aStream write: ')']
separatedBy: [aStream write: ';'].
```

In this case the parameter names are the same, but there are different messages, literals and different temporary variables of blocks present in the code. The different temporary variables from the block codes are not taken into account, as they are equivalent from any point of view. The important differences are the following: In the first statement of both methods, different literals are sent as parameters for being written in a stream. In the second statement, the messages `terms` and `keywords` are different, but used consistently within the methods. The literal and message differences can be extracted to the sibling classes, for generating exactly the same code in both siblings. From here, the refactoring possibilities are diverse: the large duplicated segments of code could be extracted and pulled up, or they could be stored in an intermediate class, created specially to keep the common part together.

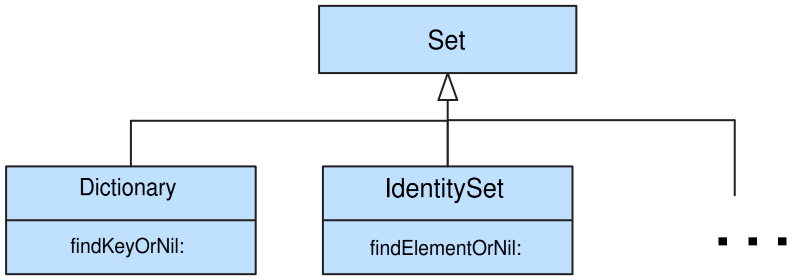


Figure 3.5: *Duplicated Code* in the classes `Dictionary` and `Set`

Detecting the Bad Smell

The detection of equivalent code is accomplished by the following:

- Variables are bound while the parse trees are traversed, so it does not matter if they are parameters or temporaries. If they are handled consistently in the code they will be considered as equivalent.
- We use the same approach for message calls. When different messages are found during the tree traversal, they are stored in an association list, and when any of them is found afterwards in one parse tree, the correspondent message must be found in the other parse tree. For duplicated code in the same class though, we ask all message sends to be equal, because of refactoring purposes that will be taken into account later on.
- Literals can take any value, except when searching duplicated code in the same class, where they have to be exactly the same literals in the same places.

For efficiency reasons, first we run a more general structural checker, that only verifies that the elements appear in the same places. For example, an assignment is followed by a message send, which is followed by a block of code, in both parse trees. The statements found structurally similar are passed to a more complex predicate that binds variables and messages, where instance variables are not exchangeable with other variables. We can observe this in the following predicate for the case of duplicated code in a subclass:

```
duplicatedCodeInSubclassOf(?statements1,?statements2,?subclass,
   ?superclass,?selector1,?selector2,+?thresholdNumber) if
      class(?superclass),
      subclassOf(?subclass,?superclass),
      methodWithNameInClass(?method1,?selector1,?subclass),
      methodWithNameInClass(?method2,?selector2,?superclass),
      duplicatedStructureInMethods(?statements1,?statements2,?method1,
            ?method2,?thresholdNumber),
      findall(<?iv,?iv>,instanceVariableInClassChain(?iv,?subclass),
            ?ivl),
      initialVariableList(?initial),
      append(?initial,?ivl,?variableList),
      compareStatements(?statements1,?statements2, ?variableList,
            ?varAssoc, <>, ?fullMethAssoc)
```

This predicate detects duplicated code between methods of a given superclass and a subclass of it. The first four predicates retrieve the methods that will be compared. After, we call the mentioned predicate `duplicatedStructureInMethods`. If there is structural duplication, we compare the statements that were found duplicated using the `compareStatements` predicate. We give as parameter the initial variable association list, which contains bindings for the variables `self` and `super`. We add the instance variables that the class understands to this list, since we do not allow instance variables to be bound to any other variable in our search for duplicated code. The initial method association list we use is an empty list, as we do not contemplate restrictions in this area.

Detecting the Bad Smell

The generic bad smell predicates for detecting duplicated code have the form:

```
badSmell(<duplicatedCode,codeInClass>, ?class, <?duplicatedTriplet>,
<?numberStatements,?numberPlaces>)
badSmell(<duplicatedCode,codeInSiblings>, ?class, <?duplicatedTriplet>,
<?numberStatements,?numberPlaces>)
badSmell(<duplicatedCode,codeInSubclasses>, ?class, <?duplicatedTriplet>,
<?numberStatements,?numberPlaces>)
```

Where `?duplicatedTriplet` is a list that contains triplets of class, selector and statements with equivalent code. This means that if the same code is present in three or four siblings, we will return only one result with all the information of the duplications. In the case of duplicated code in subclasses, this is accomplished by finding all occurrences of `duplicatedCodeInSubclassOf` in the subclasses of a given class, and grouping the results according to the duplicated statements that were found.

3.2.8 Weighing Bad Smells

In order to discriminate all possible results that can be obtained with an automated detection tool, it is necessary to rate the bad smells depending on their seriousness. Common sense says that finding *Duplicated Code* is more important than finding *Message Chains*. But if the duplicated code has two lines, and the size of the message chain is 15, we might reconsider our opinion.

As we can see in figure 3.1, we can use the information given by the bad smells for weighing the bad smells. We propose a `smellWeight` predicate that, given a bad smell's name and its smell indicators (number of duplicated lines, ratio of delegated methods or size of a parameter clump) gives a hint of how bad the smell actually is.

This predicate should also retrieve the user configurable values of which smell is more important than other without taking into account the indicators. The importance of these indicators in the calculation of the final weight is also configurable, and they should be based on the experience of the user. Therefore, general default values can be given, but the user should state which smells he considers worse, and how important the indicators are for this consideration.

Our first version is pretty simple, and a more accurate calculation is proposed for future work:

```
smellWeight(?badSmellName, ?indicators, ?weight) if
    userSmellClassification(?badSmellName,?classification),
    userSmellIndicatorsRelevance(?badSmellName,?indicatorRelevance),
    mapRelevance(?indicators,?indicatorRelevance,?weighedIndicators),
    average(?indicators,?averageIndicators),
    product(?averageIndicators,?classification,?weight)
```

Where the first predicate retrieves the user-defined classification of how important the smells are in relation to each other. The predicate `userSmellIndicatorsRelevance` retrieves the user-defined relevance of the different indicators of the smell (e.g. how important are the aount of duplicated statements or the amount of places where the duplication is found). The third predicate `mapRelevance`, calculates the actual weight of the indicators. Afterwards, our simple algorithm just calculates the average of the weighed indicators, and multiply them with the absolute classification value of the smell.

3.3 Analyzing Bad Smells and Proposing Refactorings

Now that we have illustrated how to detect bad smells by means of logic meta programming, we explain how to use the information provided by the bad smell predicates for proposing appropriate refactorings.

Bad smells do not always lead to a unique refactoring. Sometimes different refactorings solve the same smell, for example for *Duplicated Code in Siblings* we can push up the method directly to the superclass or create an intermediate class to hold the common parts. Even if there were only one refactoring proposed, we do not intend to perform refactorings automatically without user interaction. The developer must decide if one of the refactorings of the presented ones is most appropriate for the situation, or decide to leave the code intact.

By analyzing the information gathered during the detection of the bad smell, we are able to propose proper and applicable refactorings. We present only feasible refactorings for which the preconditions are satisfied to the user. This is needed to avoid proposing refactorings that will be refused by the refactoring engine. Sometimes the detected bad smell includes the checking of preconditions implicitly, like the case of unused entities, where the preconditions consists on checking that the entity is not used in the system, which is exactly what the bad smell does. In these cases we can safely propose refactorings, knowing that the preconditions are fulfilled. In other cases, like duplicated code, we need to check if the preconditions are accomplished or not. Therefore, we need to access the preconditions of each refactoring in the refactoring engine separately, which is achieved by invoking the method `preconditions` that is defined in the `Refactoring` class hierarchy.

For different reasons, not every detected bad smell will lead to refactoring propositions. There are some bad smells that we are able to detect because of their structural information, like *Message Chains* or *Middle Man*, but we cannot determine the related entities that are involved, because of the lack of typing information. The other case are smells that are easy to detect like *Large Class* and *Long Method*, where the refactoring possibillities are various and depend highly on the users decision. For the following smells we do not propose refactorings:

Data Class. The refactoring Fowler proposes is to see what the users of the data class do with the information they obtain, in order to move some behavior to the data class. This analysis needs dynamic typing information and besides it is a rather fuzzy.

Message Chains. For solving this smell, the refactoring *Hide Delegate* can be applied in some points of the chain that is detected. But we do not know who the receivers of the message sends are because this information is determined at runtime in our dynamically typed environment.

Middle Man. There is a refactoring *Remove Middle Man* which applies to this case. We can not proportionate the proper parameters to this refactoring as we do not know to whom the methods are being delegated to. This is also dynamic information.

Nil Checking. The refactoring *Introduce Null Object* should be proposed, but in a detected nil checking we do not know the identity of the object that is receiving the nil cheks, as this is dynamic information. This means we cannot identify the class from which we want to create a null subclass.

Parameter Conditionals. In this case, the refactoring *Replace Parameter With Explicit methods* should be proposed. This consists in extracting the different code run by the conditional into separate methods. The rest of the restructuring is rather fuzzy, as we do not know the semantic relation of the extracted methods with the original one.

Large Class and *Long Method.* For the *Large Class* case, Fowler proposes the *Move Instance Variable* refactoring, which is not supported by the Refactoring Browser, however the user can move any methods he wants to other classes in order to decrease the size of the class. For the smell *Long Method*, the user can extract any statements for decreasing its size. These approaches have no automatization at all and depend on the users selection between a large number of possibilities. For this reason we do not propose these refactorings, as we cannot determine in forehand the proper parameters for them.

3.3.1 Proposing Refactorings

Like in the detection of bad smells, we use a generic predicate for proposing refactorings. In this case the predicate is `proposeRefactoring`, and has the following form:

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

The 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 variables, since these are all entities for which some refactorings have been defined [40, 26]. The second argument identifies the particular refactoring that should be applied. For example `inlineParameter, pushUpMethod, replaceMethodWithMethodObject` or any other refactoring that is defined. 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, depending on the refactoring.

In the next sections, we will provide concrete examples of the `proposeRefactoring` predicate. We separate the proposed refactorings in two cases: the ones that just use the results of the bad smell and the ones that need to analyze the related entities of the bad smell in different degrees of depth. In both cases we can find more than one refactoring solving a particular smell. The bad smells referenced in the following sections that were not defined in section 3.2 can be found in the appendix A.

3.3.2 Simple Cases

We consider that the simple cases for proposing refactorings are the ones where the related entities of the bad smell are not analyzed. For example, the smell *Magic Number* is solved with the refactoring *replaceMagicNumber*. This is stated as follows in terms of logic predicates:

```
proposeRefactoring(?class,replaceMagicNumber,<?class,?magicNumber>) if
    badSmell(<duplicatedCode,magicNumber>,?class,<?class,?magicNumber)
       ,?).
```

This refactoring replaces all appearances of the symbol by a message call that returns the symbol in a given class. The parameters of the refactoring are the same as the ones returned by the `badSmell` predicate, and the name of the created method that returns the literal is asked to the user. We do not need to perform a precondition check, as all we need is the symbol to be present in the class, which we know occurs repeatedly.

Applying the query on the class `ByteArray`, a class with several magic numbers described in the previous section,

if `proposeRefactoring(`*[ByteArray]*`, ?refactoring, ?arguments)`

returns the expected result:

if `proposeRefactoring(`*[ByteArray]*`, replaceMagicNumber, <`*[ByteArray]*`,32>`

An example of a smell that can be solved by more than one refactoring is the *Too Many Temporaries* smell. In this case, we propose two refactorings, where none of them analyzes the related entities of the smell.

Replace Method with Method Object. This refactoring moves the method to a new class, converting all temporaries into instance variables. We do not need to check preconditions, as the new class will be empty, so we can safely move the method to it, and transform the temporaries in instance variables.

Replace Temp with Query. This refactoring replaces temporary assignments with a method call. We only propose this refactoring if the preconditions are met. Note that this refactoring is proposed for every temporary that complies to the preconditions.

Other straightforward refactoring propositions are the following:

Unused Parameter. The *removeParameter* refactoring is proposed, with parameters `class`, `selector` and `parameter`. The preconditions do not need to be checked as we are sure from the bad smell that the parameter is not being used in any selector that defines the associated keyword.

Unused Instance Variable. The *removeInstanceVariable* refactoring is proposed, with parameters `class` and `instanceVariable`. Here, checking preconditions is also not needed, as we know the variable is not used in the hierarchy.

Abstract Method not Implemented. The *removeMethodFromHierarchy* is proposed, with parameters `highestClass` and `abstractSelector`. The preconditions do not need to be checked, as the bad smell ensures that the method is positively not implemented in the hierarchy.

Feature Envy. The *moveMethod* refactoring is proposed, with parameters `highestClass` and `abstractSelector`. The preconditions do not need to be checked, as the bad smell ensures that the method is positively not implemented in the hierarchy.

Odd Names. We propose *renameMethod* or *renameClass*, with no preconditions, as the new name will be asked to the user only when he accepts to perform the refactoring.

Refused Interface. We propose *replaceInheritanceWithDelegation*, with the `subclass` that cancels the interface, and the `superclass` as parameters. This refactoring has not been implemented yet.

Parameter Clump. We propose *introduceParameterObject*, with the list of parameters that compose the clump as parameters to the refactoring. The data contained in the clump parameters are converted into a single object, and the signatures of all selectors containing the clump are transformed to use this new object and obtain the data from there. This refactoring has also not been implemented yet.

3.3.3 Complex Cases

The complex cases analyze the related entities of the bad smell in order to propose more appropriate refactorings.

Inappropriate Interface A more elaborate example is the case of the refactorings proposed for solving the smell *Inappropriate Interface*, which was introduced in section 3.2.6. The situation is that a number of subclasses share an interface that is not supported by the superclass, difficulting the maintenance of the involved classes. For this case there are two suitable solutions: Either we augment the interface of the class with the missing methods by using the *addMethod* refactoring as shown in figure 3.6, or we insert an intermediate superclass between the root class of the hierarchy and the subclasses that implement a shared interface by using the *addClass* refactoring, as shown in figure 3.7. Note that the last refactoring should not be proposed if the interface is shared by all subclasses of the superclass. In this case an intermediate class would be unnecessary.

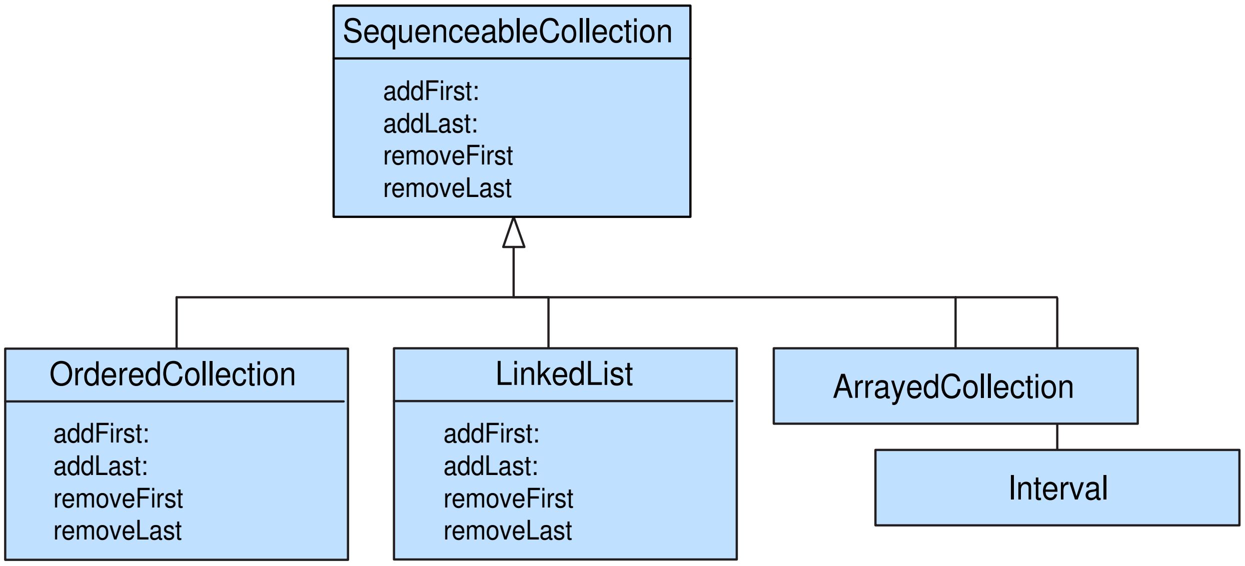


Figure 3.6: Applying the refactoring *addMethod* for solving the bad smell *Inappropriate Interface*

The first refactoring is proposed in every case, leaving to the user the decision of adding the methods in the superclass, making all the subclasses inherit this behavior.

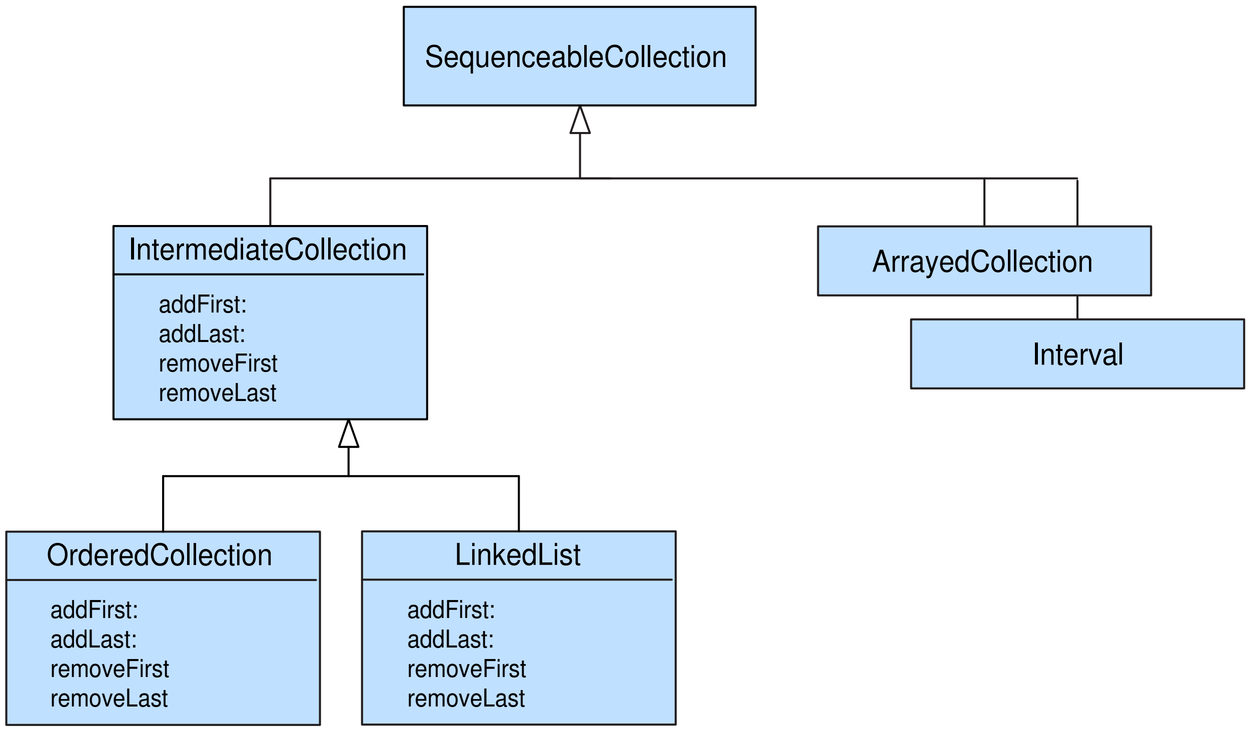


Figure 3.7: Applying the refactoring *addClass* for solving the bad smell *Inappropriate Interface*

```
proposeRefactoring(?class,addMethods, <?class,?interface>) if
    badSmell(<inappropriateInterface>, ?class, <?class,?interface,?>,
       ?)
```

Note that we do not need to check preconditions for proposing this refactoring, as we have the certainty that the methods we want to add are not defined in the class.

The second proposition depends on the characteristics of the related entities:

```
proposeRefactoring(?class,addClassWithInterface, <?class,?interface,
   ?subclasses>) if
    badSmell(<inappropriateInterface>, ?class, <?class,?interface,
        ?subclasses>, ?),
    metric(NOSC, class(?class), ?totalSubclasses),
    smaller(?numberSubclasses,?totalSubclasses)
```

The refactoring *addClassWithInterface* is based on the Refactory Browser refactorings *addClass*, that adds an intermediate class between the class and the given subclasses, and *addMethods*, that adds the desired method declarations to the recently created class. We do not need to check for preconditions because the created class will be empty, so no name clashes can occur when adding methods.

Continuing with our example, if we apply the query `proposeRefactoring` over the class `SequenceableCollection` we saw in the previous section:

if `proposeRefactoring(`*[SequenceableCollection]*`,?refactoring,?arguments)`

we obtain two proposed refactorings as depicted in figures 3.6 and 3.7:

`proposeRefactoring(`*[SequenceableCollection]*`,`
`addMethods, <`*[SequenceableCollection]*`, <addFirst:, addLast:,removeFirst,removeLast> >)`

`proposeRefactoring(`*[SequenceableCollection]*`, addClassWithInterface,`
`<`*[SequenceableCollection]*`,<addFirst:, addLast:,removeFirst,removeLast>,`
`<`*[OrderedCollection]*`, `*[LinkedList]*`> >)`

These proposed refactorings are given with all the information needed for performing the refactoring.

Lazy Class The bad smell *Lazy Class* can also be solved with different refactorings. The different propositions are:

Collapse Hierarchy with Superclass. This means to push methods and instance variables up to the superclass, and remove the empty class. This is proposed if the preconditions of the refactorings *Push Up Method* and *Push Up Instance Variable* are met, as shown in the predicate:

```
proposeRefactoring(?class,collapseHierarchyWithSuperclass,
    <?class>) if
    badSmell(<lazyClass>,?class,<?class>,?),
    forall(methodWithNameInClass(?,?selector,?class),
       checkPreconditions(pushUpMethod,<?selector, ?class>)),
    forall(instanceVariableInClass(?instanceVariable,?class),
       checkPreconditions(pushUpInstanceVariable,
        <?instanceVariable, ?class>))
```

Collapse Hierarchy with Subclass. If the lazy class has subclasses, we apply the same reasoning as with the superclass, but applied to the subclasses, and pushing down the functionality. This refactoring is proposed only if the preconditions of the refactoring *Push Down Method* and *Push Down Instance Variable* are fulfilled, as shown in the following predicate:

```
proposeRefactoring(?class,collapseHierarchyWithSubclass, <?class>) if
    badSmell(<lazyClass>,?class,<?class>,?),
    metric(NOSC, class(?class), ?totalSubclasses),
    greaterOrEqual(?totalSubclasses,1),
    forall(methodWithNameInClass(?,?selector,?class),
       checkPreconditions(pushDownMethod,<?selector, ?class>)),
    forall(instanceVariableInClass(?instanceVariable,?class),
       checkPreconditions(pushDownInstanceVariable,
        <?instanceVariable, ?class>))
```

Inline Class to all the callers, with the `class` as parameter. This refactoring is always proposed.

```
proposeRefactoring(?class,inlineClassToAllCallers, <?class>) if
    badSmell(<lazyClass>,?class,<?class>,?)
```

Duplicated Code The most complex case we tackle is the duplicated code, which involves more cases, with more details to take care. The detection of this bad smell was explained in section 3.2.7. In the three forms of duplicated code we consider there are some common characteristics: The duplicated code statements we find can be either code segments or whole methods. Also, all the analysis related to the renaming of methods and standardization of the duplicated code is made within the refactoring itself, after the user has decided to apply it. These details are explained in the following section, that discusses the applied refactorings and their characteristics.

The refactorings we propose for duplicated code are the following:

```
proposeRefactoring(?class, refactoringDuplicationInClass,
    <?duplicatedTriplets>) if
    badSmell(<duplicatedCode,codeInClass>,?class,
        <?duplicatedTriplets>,?).

proposeRefactoring(?class, refactoringDuplicationInSubclasses,
    <?duplicatedTriplets>) if
    badSmell(<duplicatedCode,codeInSubclasses>,?class,
        <?duplicatedTriplets>,?)

proposeRefactoring(?class, refactoringDuplicationInSiblings,
    <?duplicatedTriplets>) if
    badSmell(<duplicatedCode,codeInSiblings>,?class,
        <?duplicatedTriplets>,?)
```

Where `?duplicatedTriplet` is the list that contains triplets of class, selector and statements with equivalent code.

3.4 Applying Refactorings

Until now, our process helps the user to detect bad smells and suggests a number of refactorings that are possible to apply with the proper parameters. The following step is to apply the refactoring selected by the user.

As we are using Martin Fowler's Refactoring Book for determining the bad smells, we are also using some of the refactorings he defines. The problem is that most of his refactorings are not supported directly by the Refactoring Browser, although they can be. The lack of support for composing refactorings of the Refactoring Browser enforced us to compose the existing refactorings by logic meta-programming means.

This section explains how we wrap the refactorings provided by the Refactoring Browser and how we define the composite refactorings we propose to the user.

3.4.1 Wrapping Existing Refactorings

In order to apply refactorings in our framework we wrap the ones provided by the Refactoring Browser in logical predicates. For example, for the *removeParameter* refactoring we rely on the implementation of the Refactoring Browser, as shown in the code:

```
refactoring(removeParameter,<?parameter,?class,?selector>)  if
    symbolAsString(?parameter,?parString),
    [(Refactory.Browser.RemoveParameterRefactoring
        removeParameter:?parString in:?class selector:?selector)]
```

If needed, we convert the SOUL parameters to the appropriate Smalltalk entities, like in this case we do with the `parameter` that is converted to string by using the predicate `symbolAsString( ?parameter,?parString)`. We give the appropriate parameters to an instance of the desired Refactoring, and execute it if we want to apply the refactoring, or ask to check preconditions if we want to do so. In order to invoque these refactorings, we rely on the feature of SOUL that allows to manipulate objects from the underlying Smalltalk system inside logic clauses or terms.

For checking the preconditions, we rely on the separation of the refactoring from its preconditions, provided by the refactoring browser:

```
checkPreconditions(removeParameter,<?parameter,?class,?selector>)  if
    symbolAsString(?parameter,?parString),
    [(Refactory.Browser.RemoveParameterRefactoring
        removeParameter:?parString in:?class selector:?selector)
           preconditions check ]
```

By this means, we wrapped all the refactorings supported by the Refactoring Browser, mentioned in section 2.2.3. Most of them are simple to wrap as they receive parameters we are used to such as class names, variables names and selectors.

But a small group like *extractMethod* or *inlineTemporary*, are based on user selection from source code, and they receive as parameters the interval of the source code positions of the portion that needs to be extracted or inlined. This is not a problem at all, because the method representation of SOUL includes these intervals as part of all components of the parse tree. This means that the source code intervals are part of the nodes containing statements, message sends, assignments, literals or variables. For example the predicate for extracting code is the following:

```
extractMethod(<?from,?to>,?selector,?class)  if
    equals(?interval,[Interval from:?from to:?to]),
    [(Soul.ExtractMethodNoQuestionRefactoring extract:?interval
        from:?selector in:?class) execute]
```

Here we also mannipulate Smalltalk entities in order to produce the correct parameters for the refactoring. We use our representation of an interval that is a list with `?from` and `?to` values for generating a Smalltalk interval object that the refactoring needs as parameter.

One example of how we use the refactoring *extractMethod* by logic means is the predicate `extractMethodWithName`. This predicate receives the interval of the code that needs to be extracted, and the selector and the class containing this code. If the name the user wants to put on the extracted code is given, the predicate extracts the code and names it with that input. If the name is not given, the predicate asks it to the user and returns it in the term `?toSelector`.

3.4.2 Composite Refactorings

Many of the refactorings that solve the bad smells we detect, are not directly implemented by the Refactoring Browser. However, some of them can be defined by composing the the ones that are provided. Our approach relies on the definition of few atomic primitive refactorings, that we compose with logic predicates by stating the sequences of refactorings that we need to apply. These constructions are essential for the scalability of our approach, since it allow us to add and to manage composite refactorings in a clear and flexible way, without hard-coding any relation.

For defining composite refactorings, we determine which is the sequence of refactorings we need to perform, and which ones are prerequisites of the others. We state these relations by using the predicate `requires`:

```
requires(?context,?refactoring,?refactoringList)
```

Where `?refactoring` is a primitive refactoring that requires a number of refactorings `?refactoringList` to be performed before it. The term `?context` specifies for which context the requirement is valid.

For actually executing the composite refactoring, we invoque the predicate `applyRefactoring` that checks for nested refactorings that need to be performed before applying the desired one. The predicate is the following:

```
applyRefactoring(?context,<?refactoring,?arguments>) if
    requires(?context,<?refactoring,?arguments>,?listRequirements),
    checkRequirements(?context,?listRequirements),
    ?refactoring(?arguments)
```

This predicate retrieves the requirements of a refactoring, and check that they are fulfilled. If this is the case, it directly applies the refactoring that it receives as parameter. The predicate for checking the requirements is the following:

```
checkRequirements(?context,<>).

checkRequirements(?context,<?refactoring|?rest>) if
    applyRefactoring(?context,?refactoring),
    checkRequirements(?context,?rest)
```

The `checkRequirements` predicate has two cases: either the refactoring has no requisites, in which case the predicate returns true, or the requisites are a list of refactorings that also have to be checked. Actually, both the `applyRefactoring` predicate and the `requires`, uses an extra variable that stores the environments, for trespassing the variables while executing the composite refactorings.

Replace Method With Method Object

For example, the smell *Too Many Temporaries* can be solved with the refactoring *Replace Method With Method Object*. This refactoring can be composed in the following way:

- First we need to create a new class with *addClass*, and to add an instance variable to the class that contains the method with *addInstanceVariable*. Note that none of these refactorings have any prerequisites with respect to this context, and that the order in which they are applied does not matter.
- Then we are able to apply the refactoring *moveMethodToComponent*, which moves the method that has too many temporaries to the newly created class, leaving a reference in the created instance variable.
- The temporaries of the moved method can now be transformed into instance variables using the refactoring *temporaryToInstanceVariable*.

When the refactoring *replaceMethodWithMethodObject* is called with parameters `?class` and `?selector`, we trigger the predicate `applyRefactoring`. The prerequisites are logically stated as follows:

```
requires(tooManyTemporaries,
    replaceMethodWithMethodObject,<?class,?selector>,
    <allTemporariesToInstanceVariables,<?newClass,?newSelector>>)

requires(tooManyTemporaries,
    allTemporariesToInstanceVariables,<?newClass,?newSelector>,
    <moveMethodToComponent,<?selector,?class,?instanceVariableName,
        ?newClass,?newSelector>>)

requires(tooManyTemporaries,
    moveMethodToComponent,<?selector,?class,?instanceVariableName,
        ?newClass,?newSelector>,
    <addInstanceVariable,<?instanceVariableName,?class>,
        addClass,<?newClass,?superclass,?subclasses>>)

requires(tooManyTemporaries,
    addInstanceVariable,<?instanceVariableName,?class>,
    <>)
```

```
requires(tooManyTemporaries,
    addClass,<?newClass,?superclass,?subclasses>,
    <>)
```

Note that the refactoring *Replace Method With Method Object* does not need to check for pre-conditions, as the method is moved to a newly created class, where no name clashes can occur. The only checking have to be made when the user finally applies the refactoring and provides the names for the class and the instance variable.

Duplicated code

The refactorings we perform for unifying and removing duplicated code are quite complex, and depend on the places where the duplicated code was found.

We contemplate three cases of duplicated code: inside the same class, between a superclass and its subclasses and between sibling classes. In all these cases, we have to take into account that the duplicated code statements we find can be either code segments or whole methods. With the refactoring *extractCode*, duplicated segments are extracted and whole methods are extracted depending on if they are overriding or overridden methods or if there are other methods with the same name.

In the case of *Duplicated Code in Class*, this process ends up with no duplicated code, because of the characteristics of the *extractMethod* refactoring in the Refactoring Browser. If we extract code that is already present in the class, the Refactoring Browser automatically replaces the duplicated code. An improbable case that can occur is that we find two exactly duplicated methods in the same class but with different names. In this case, after applying the refactoring *extractCode*, we propose to the user to remove one of the methods with *removeMethod*, or the replace the body of one of them with a call to the other by using *extractMethod*, as shown in figure 3.8.

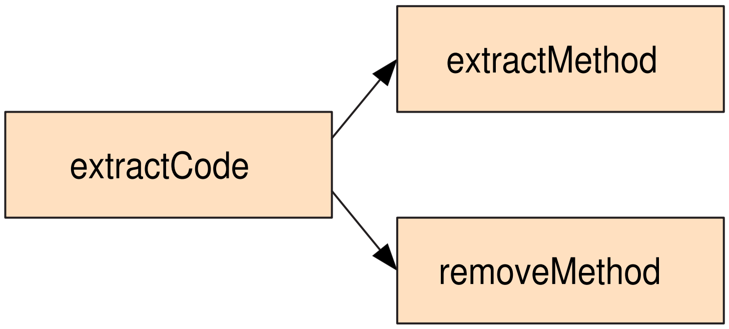


Figure 3.8: The primitive refactorings that compose the *refactoringDuplicationInClass* refactoring, for solving *Duplicated Code in Class*

The detection of *Duplicated Code in Subclasses* and *Duplicated Code in Siblings* requires several primitive refactorings, whose dependences are depicted in figures 3.9 and 3.10 respectively. In these cases we allow coherent differences related to literals and message sends. For making the code equivalent to the parameters of the Refactoring Browser, the first prerequisite is to apply the refactoring *extractDifferences*. This refactoring traverses the parse trees of all duplicated segments of code, and when a different literal or message send is found, it is extracted and replaced by a message call in all segments, with the same name. This generates duplicated code, now with identical structure, except for the temporary variables and parameters, which are equivalent to the Refactoring Browser, as long as they are handled consistently. Afterwards we apply the refactoring *extractCode*, as we did with the *Duplicated Code in Class* case. The following step is to unify the names of all the methods and extracted code. The refactoring *unifySelectors* uses the existing refactoring *renameMethod*, for ensuring that all duplicated methods end up having the same name.

For the case of *Duplicated Code in Subclasses* the last step is to apply the refactoring *pushUpMethod* which will automatically remove the code duplication, after all the other primitive refactorings have been applied. This is shown in figure 3.9.

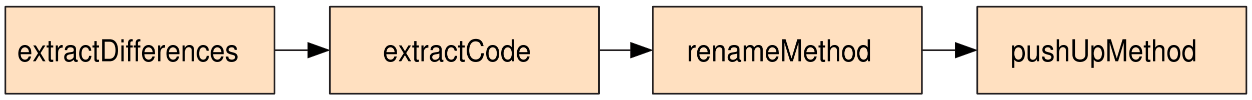


Figure 3.9: The primitive refactorings that compose the *refactoringDuplicationInSubclasses* refactoring, for solving *Duplicated Code in Subclasses*

For *Duplicated code in Siblings* we need to analyze the following:

- If the selected method is already defined in the superclass, or the code is not duplicated in every subclass, propose *addClass* to create an intermediate class and *pushUpMethod*.
- If the method is defined in the superclass propose *replaceSuperclassDefinition*.

The sequence for proposing these refactorings is shown in figure 3.10.

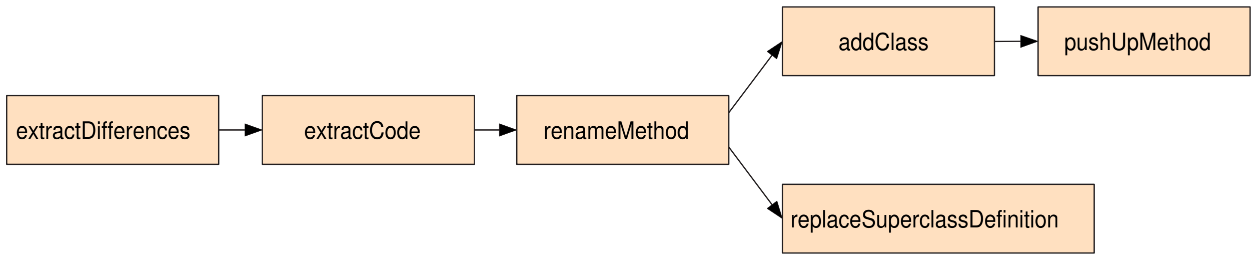


Figure 3.10: The primitive refactorings that compose the *refactoringDuplicationInSiblings* refactoring, for solving *Duplicated Code in Siblings*

Other Refactorings

Other refactorings that are easy to implement are the following:

Replace Temp with Query. This refactoring replaces temporary variable assignments with query methods. For generating it, we first extract what is assigned to the temporary using *extractMethodWithName*, for asking the user the name of the method that will be created. Then we apply *inlineTemporary* to the temporary variable.

Replace Magic Number with Query. This refactoring replaces all occurrences of a given symbol by a call to a query method. This is accomplished by extracting all the occurrences of the symbol in the class using *extractMethod*. The first time this refactoring will ask the user for the name of the extracted query method, which will only return the value of the symbol. The following times the refactoring will detect that there already exists a method that returns the value of the symbol, and it will automatically replace the symbol by a call to the query method.

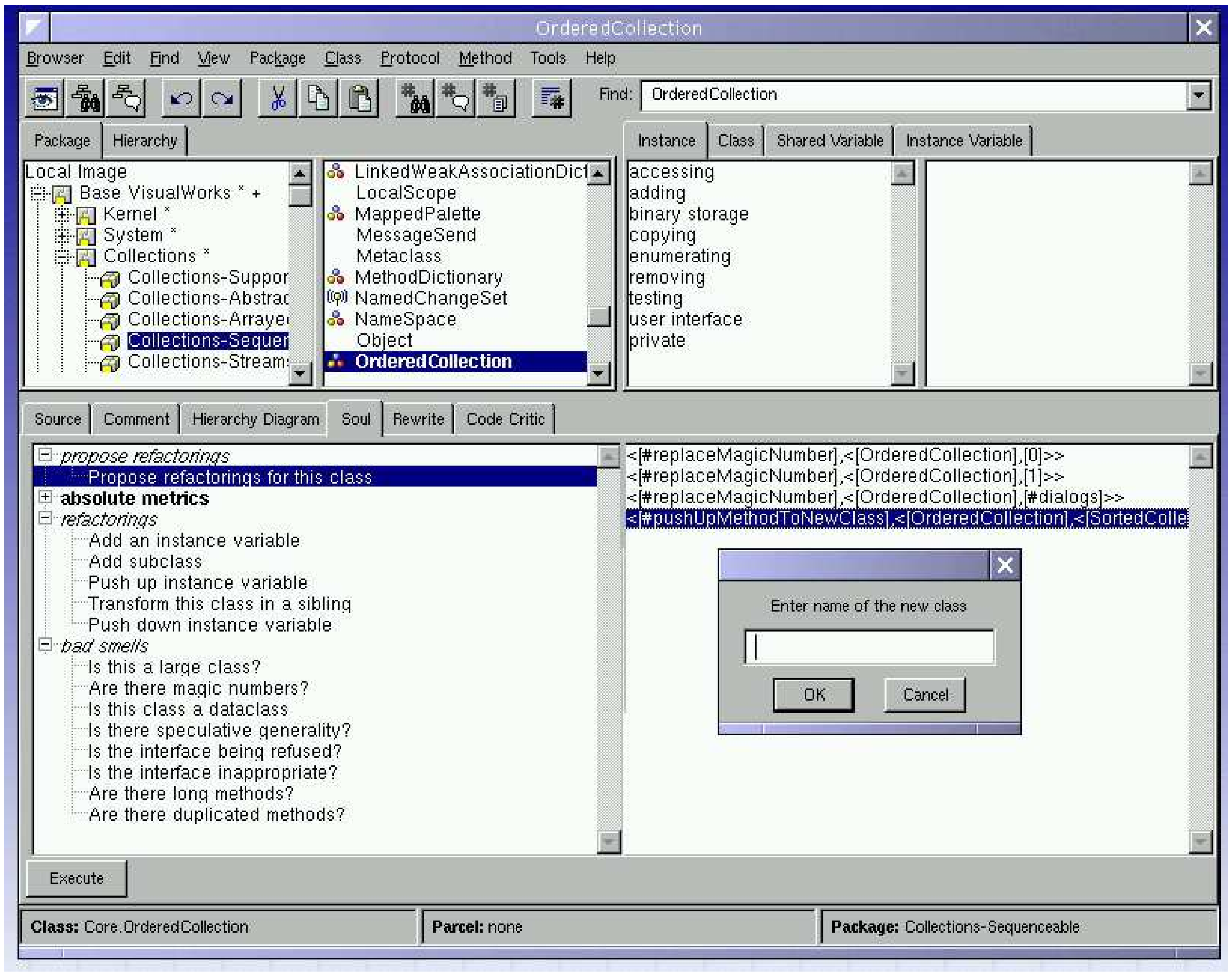
Remove Method In Hierarchy. This refactoring does not use the Refactoring Browser's *Remove Method* refactoring, as it can only be applied if the method is not called anywhere in the system. We are interested in removing the method without checkings, assuming that some detected smell needs it this way, like the *Abstract Method not Implemented*. We use a predicate provided by SOUL called `removeMethod` that does not do any kind of checking.

Add class With Interface. This refactoring adds a class with empty method definitions, that should be filled in by the user. The names of the methods to add are received as parameter.

3.5 Tool Support

Providing tool support for our approach is indispensable, for hiding the logic programming environment from the user, and to provide him with a straightforward and easy to use interface. The developer can select any class of the system to be analyzed for bad smells and possibilities of refactoring. The analysis should be performed automatically without user intervention, resulting in a list of refactorings that can be applied. The developer must then be able to pick out the refactoring(s) he wants to apply, and the tool makes the necessary transformation with a minimal amount of user input.

To provide this kind of tool support, we integrated our logic meta programming approach with the interface of the *Refactoring Browser*, which is the standard browser for the VisualWorks Smalltalk integrated development environment. This browser was augmented with a SOUL tab (see figure 3.11), which exists next to the other tabs already available (such as the *Source, Comment, Code Critic* and *Hierarchy Diagram* tabs). The SOUL tab contains a list of logic queries that can directly be invoked by the user from within the Smalltalk browser. Upon selection of a class in the upper left pane, the developer can select a number of logic queries (or



```
<[#replaceMagicNumber],<[OrderedCollection],[0]>>
<[#replaceMagicNumber],<[OrderedCollection],[1]>>
<[#replaceMagicNumber],<[OrderedCollection],[#dialogs]>>
<[#pushUpMethodToNewClass],<[OrderedCollection],<[SortedColle
```

Class: Core.OrderedCollection **Parcel:** none **Package:** Collections-Sequenceable

Figure 3.11: Tool support for detecting bad smells and proposing refactorings in Smalltalk

all of them) and click on the *Execute* button. As a result, the logic query will be executed and the results will be shown in the lower right pane.

As can be seen in figure 3.11, we have used five categories of logic queries in our experiments, which will be discussed in the following sections: *Bad Smells*, *Refactorings*, *Propose Refactorings*, *Configure Thresholds* and *Metrics*.

3.5.1 Bad Smells

This category contains all the queries that we implemented for detecting particular bad smells, defined in section **??**. For example, *Is the interface inappropriate?* invokes the logic predicate `inappropriateInterface`. The result of applying a bad smell query is a list of smell results, that includes the entities related to the smell, and the most relevant indicators such as the number of duplicated statements, or the length of a *Message Chain*, that allow the user to realize which smells are the strongest ones. It should be noted that we explicitly allow the user to select a number of bad smells (or even one single bad smell) to be checked, instead of automatically checking for all of them. This is because the user may already know that a specific bad smell does not occur in a particular class hierarchy, or he does not consider that the smell is important enough. A more technical reason is that the current implementation of SOUL is not optimized, and as a consequence, checking for all bad smells on large software systems may take quite some time.

3.5.2 Refactorings

This category provides an alternative way to access the refactorings, that are otherwise proposed to the user after the detection of a bad smell. For example, *Add subclass* executes a logic predicate that invokes the built-in refactoring that creates a new subclass of the current class, and allows the user to select which of the current subclasses should become children of the new subclass.

3.5.3 Propose Refactorings

This category contains only one query *Propose refactorings for this class.* When executed, it invokes the predicate `proposeRefactoring`. In other words, it computes all bad smells in the context of the selected class and proposes refactorings for these bad smells. In figure 3.11 we see that this returns 4 results for the `OrderedCollection` class: 3 instances of the user-defined refactoring *replaceMagicNumber*, and one instance of the composite refactoring *pushUpMethod-ToNewClass.* Note that in the list of proposed refactorings, only those refactorings are mentioned for which the preconditions are satisfied. The latter is needed to avoid proposing refactorings that will be refused by the refactoring engine. Therefore, we need to access the preconditions of each refactoring in the refactoring engine separately.

From the list of all proposed refactorings, the user can select a refactoring, that can be executed directly. For example, in Figure 3.11 we see that selecting the composite refactoring `pushUpMethodToNewClass` opens a new window in which a new superclass needs to be specified, to which the method will be pushed up.

3.5.4 Configure Thresholds

An important characteristic of smells is that many of them depend on the user's perception. A way to discriminate when a bad smell is considered a bad smell or not is to provide user configurable thresholds. We use these threshold values in many of our defined bad smells, both for tuning the smells and making the detection process more efficient. In this category, the user can configure all the threshold values, like the number of methods a class should have in order to be considered as a large class, the amount of keywords to be considered in a parameter clump, the amount of statements code should have or the ratios for the delegating methods of a middle man.

3.5.5 Metrics

This category contains a number of logic queries that compute object-oriented metrics, as explained in section 3.1.3. These metrics can be used to detect those places in the code that are worthy of further investigation because they are likely candidates for bad smells. Once these locations in the source code have been identified, we can use the bad smell queries to analyze these parts of the program in more detail.

3.6 Conclusions

In this chapter, we have explained how to support different steps of the refactoring process using logic meta programming. Thanks to powerful features as backtracking and unification, we can write complicated algorithms related to the structure of the code in a straightforward, understandable and concise way. This includes the detection of bad smells, the proposal of refactorings that can remove the bad smell, and the application of these refactorings.

By means of logic meta programming and the appropriate predicates, we have been able to define nine bad smells completely and five of them partially, most of them highly time-consuming to detect manually. There are four kinds of information that limit the number of smells we can detect: the lack of typing information, the lack of dynamic runtime information, fuzzy definitions and the lack evolutionary information.

Some of the bad smells that we do detect, like the different cases of *Measured Smells* or *Speculative Generality* are not hard to determine by means of basic auditing tools. However we also tackle the detection of more high level smells like *Inappropriate Interfaces* between a class and its subclasses, *Message Chains* or *Refused Interfaces*, which are not easy to detect with the basic tools.

In order to present the bad smell results to the user, we propose to weigh bad smells regarding the experience of the user and the characteristics of the presented smells in the system. This is highly necessary due the number of bad smells our automated tool can detect, so the user can get an overview of how strong the presented smells are.

The following step, consists on proposing useful refactorings to the user. For accomplishing this, we analyze the context of detected bad smells by using logic meta programming. We found that most of the bad smells are solved directly with one or more refactorings without the need of a deeper analysis. In this case all we need is to pass the entities related to the bad smell as parameters to specific refactorings. Other case are bad smells like *Inappropriate Interface* or *Lazy Class*, where we must perform some extra analysis over the related entities, like the amount of subclasses involved in the smell, in order to propose certain refactorings. However, for some of the bad smells we detect, we cannot propose proper refactorings because this requires to analyze information related to dynamic typing or the refactoring possibillities are to many and depend highly on the users decision like the refactorings for the smells *Large Class* or *Long Method*.

The last step we perform in the refactoring process is applying automatically the refactorings selected by the user. This is accomplished by relying on existing safe and efficient implementations, in this case provided by the Refactoring Browser. The refactorings of the Refactoring Browser were wrapped into logic predicates and applied directly or by means of composition. Composite refactorings can be defined using logical predicates that state which are the prerequisite refactorings that need to be performed before applying a certain one. The composition of refactorings is also necessary for the scalability of our approach. Relaying on the definition of a few powerful primitive refactorings, our framework allows to compose them in a clear and flexible way by stating their dependencies with logic meta-programming predicates. With this approach we were able to implement some of the refactorings proposed by Martin Fowler, but not all that we would have liked to, as many of them depend on human intuition or on typing characteristics.

Our approach defines the bad smells, the proposition of refactorings and the composite refactorings in a declarative way, making them very readable, scalable and easy to maintain. For adding more smells, refactoring propositions or composite refactorings, a few predicates have to be defined that state how the code entities will interact.

The tool that supports the detection of bad smell detection with user configurable thresholds, propose proper refactorings and apply the ones selected by the user is integrated in the *Smalltalk* development environment, and allows the user to apply our logic queries in a user-friendly environment.

Chapter 4

Experiments

This chapter discusses the experiments we have conducted to validate our approach over three case studies with different characteristics: the Collection hierarchy from Smalltalk Visualworks, the HotDraw Framework and SOUL, the application we use for reasoning about code. The experiments consist of detecting the bad smells on the cases, analyzing the most interesting results and checking whether the proposed refactorings are useful or not. These results also led us to discussions like possible refactorings that were not proposed by our tool, the scalability of our approach and the opportunities for cascade refactorings.

4.1 Case studies

To perform our experiments, we selected three different applications on which we detected bad smells and refactoring opportunities using the tool described in the previous chapter.

SOUL Application. As a first application, we chose the SOUL application itself [56, 57], one of which we have close contact with the developers. This enables us to assess the correctness of the identified bad smells and the usefulness of the proposed refactorings. We used version 3.0 of SOUL, which was the latest version at the time we started experimenting. In this version, the implementation consists of 126 classes and 1627 different method implementations, which makes it a small to medium-sized application.

Collection Hierarchy. The second selected application is the Smalltalk *Collection* hierarchy. It is an essential part of the Smalltalk programming development environment, and as such it is heavily optimized, and not much subject to changes. Therefore, it is interesting to analyze this hierarchy to find out whether there are still any remaining bad smells and opportunities for refactoring. We consider the classes in the hierarchy of collection present in the bundle *Base Visualworks*, from Visualworks version 7.1. In total there are 94 classes and 1842 methods.

HotDraw Framework. A final application is *HotDraw* [9], a small-scale application framework in the domain of structured drawing editors. HotDraw has undergone many evolutions, and many different versions of it exist for different programming languages. Moreover, it is a very popular, successful, well-documented and well studied framework. As a consequence, it is very interesting to verify whether we can still identify some design flaws in this framework. We are using version 4.5.1 which contains 69 classes and 886 method implementations.

4.2 Detected Bad Smells

This section described the results of the experiments we performed over the three case studies. This includes two tables with a list of all the occurrences of bad smells present and their respective indicators of how strong the smell is.

Of all the presented results, in the following sections we will focus on the occurrences of the bad smells explained in detail in section 3.2: *Unused Parameter*, *Parameter Clump*, *Inappropriate Interface* and *Duplicated Code*. We also discuss the refactorings that were proposed and which ones were effectively applied to remove the bad smell.

4.2.1 Summary of Results

Tables 4.1 and 4.2 show the results of detecting bad smells on the three cases. The first column contains the name of the detected bad smell. The bold numbers correspond to the amount of bad smells found, and the italics are the average of the indicators of how strong the smell is. For example, in the case of *Too Many Instance Variables*, the Collection hierarchy presented *7* cases of classes with this characteristic. Five of the classes have five instance variables, one has six and one has seven. This gives an average of *5.4* instance variables, which is the number in italics. Note that we only take into account the detected cases for calculating the average of the indicators.

The threshold values for the bad smells were subjectively determined. Here we give the values and explain the indicators of the ones we will treat in more detail (the definition of the indicators of the other smells can be found in appendix A):

- *Unused Parameter*. This smell has no threshold values. The indicator is the number of selectors that receive the parameter but do not use it.
- *Parameter Clump*. Considers parameter clumps with more than *4* parameters in more than *4* methods of a class. The indicators are the amount of parameters that conform the clump, and the amount of methods in the system where the clump was found.
- *Duplicated Code*. Considers duplicated segments of *2* sentences or more. The indicators are the amount of statements that present duplicated code, and the places where it was

	Collection		HotDraw		SOUL	
Too Many Instance Variables	**7**	*variables:5.4*	**8**	*variables:5.8*	**11**	*variables:7.6*
Too Many Methods	**21**	*methods:57.9*	**8**	*methods:54.8*	**25**	*methods:58*
Too Many Statements	**10**	*statements:14.1*	**4**	*statements:13.25*	**6**	*statements:17.5*
Too Many Temporaries	**32**	*temporaries:6.6*	**6**	*temporaries:6*	**5**	*temporaries:5.6*
Long Parameter List	**18**	*parameters:9.8*	**0**	-	**9**	*parameters:6.8*
Redundant Parameter	**27**	*calls:2.6*	**5**	*calls:2*	**10**	*calls:2.2*
Magic Number	**34**	*appears:7.4*	**16**	*appears:4.7*	**26**	*appears:7*
Duplicated Code in Class	**40**	*statements:2.6* *places:1.4*	**7**	*statements:2.1* *places:1*	**17**	*statements:2.5* *places:1.2*
Duplicated Code in Subclasses	**40**	*statements:2.5* *places:1.5*	**6**	*statements:2.5* *places:2*	**15**	*statements:2.8* *places:2.1*
Duplicated Code in Siblings	**14**	*statements:2.6* *places:1.9*	**8**	*statements:2.1* *places:2.3*	**17**	*statements:2.5* *places:1.4*
Data Class	**0**	-	**1**	*methods:6*	**0**	-
Parameter Clump	**2**	*parameters:7.5* *places:8.5*	**0**	-	**1**	*parameters:4* *places:96*
Refused Interface	**1**	*ratio:0.3* *methods:9*	**0**	-	**0**	-
Refused Inheritance	**22**	*ratio:0.9* *self sends:30.1*	**11**	*ratio:0.9* *self sends:20.3*	**11**	*ratio:0.9* *self sends:40*
Inappropriate Interface	**9**	*interface:3.6* *subclasses:2.1*	**1**	*interface:2* *subclasses:2*	**1**	*interface:3* *subclasses:2*

Table 4.1: (table 1 of 2). Experimental results for detecting the bad smells in the following categories: Measured Smells, Duplication, Data and Interfaces

	Collection		HotDraw		SOUL	
Feature Envy	**8**	*ratio:0.9 accessors:7.6*	**2**	*ratio:0.8 accessors:7*	**5**	*ratio:0.9 accessors:11.4*
Message Chain	**1**	*length:5*	**1**	*length:5*	**2**	*length:5*
Middle Man	**1**	*ratio:0.3 delegators:4*	**1**	*ratio:0.5 delegators:8*	**1**	*ratio:0.6 delegators:6*
Lazy Class	**4**	*methods:1 inst. variables:0 method size:0.75*	**0**	-	**12**	*methods:1 inst. variables:0 method size:0.39*
Unused Parameter	**6**	*not used:1.3*	**6**	*not used:2.2*	**48**	*not used:1.7*
Unused Instance Variable	**0**	-	**0**	-	**5**	*not used:1*
Abstract Method Not Implemented	**0**	-	**0**	-	**7**	*not implemented:1*
Odd Name	**52**	*odd words:1*	**23**	*odd words:1*	**97**	*odd words:1.3*
Parameter Conditional	**53**	-	**12**	-	**20**	-
Nil Checking	**13**	*nil checks:5.7*	**2**	*nil checks:3.5*	**3**	*nil checks:7*

Table 4.2: (table 2 of 2). Experimental results for detecting the bad smells in the following categories: Responsibility, Unnecessary Code and Conditional Logic

found. For example, in the case of siblings, in how many siblings we found the same duplicated code.

- *Inappropriate Interface.* Considers interfaces larger than *2* that are present in at least *2* subclasses and not in the superclass.

Regarding the first group of smells from table 4.1, that we consider in the larger classification of measured smells, or highly related to metrics, we see that the Collection hierachy contains much more occurrences than the other case studies. The amount of *Too Many Temporaries* (with a threshold value of five temporaries) is very high, as well as the occurrences of *Redundant Parameters*. SOUL has an intermediate number of bad smells of this kind, and HotDraw is has definitively less bad smells of this kind. Moreover, HotDraw does not contain a single occurrence of a long parameter list, considering a threshold of six parameters.

For duplicated code, the Collection hierarchy also has more occurrences than the others, even though the indicators for all cases are low. In the three case studies the average size of duplicated code is around two or three statements, and the average of places where the duplicated code is found is around two.

In the data related smells, we observe that the only detected data class belongs to the HotDraw framework. On the other hand, this framework doe not have any parameter clump, what can be related to the fact that it does not have long parameter lists either. The other two classes have parameter clumps, that are discussed later in this chapter.

Regarding last group, the interface smells, we only found one *Refused Interface* smell, and even this occurrence has a very low refusal ratio (30% of canceled methods from the superclass). This is a good sign of the status of the code we analyzed, as this smell is considered as very serious by Fowler. The lighter smell *Refused Inheritance* based on the analysis of the `self` sends, has more occurrences. Here the amount of `self` sends is very high (30, 20 and 40 for Collection, HotDraw and SOUL respectively) and the ratios of refusenes are around 90%. However, relative importance of this smell is low.

In first group of smells from table 4.2, we see that there are no many occurrences related to responsibility. We found a number of classes that contain methods with *Feature Envy*, but in practice we need more information than the ratio of usages of accessors in order to decide to move a method from a class to another. There were very few occurrences of *Message Chains* and *Middle Mans*, and the occurrences that were found had low indicators.

Regarding the speculative generality smells, the one that contains the largest amount of occurrences if SOUL. It is the only case studty that presents *Unused Instance Variables* and *Abstract Methods not Implemented*. These smells were recognized as serious, and were all removed afer their detection. SOUL also has more occurrences of the *Lazy Class* smell, which in some cases corresponds to classes that group others but does not contain any behaviour, and in other cases are subclasses with low amount specialization, but that this specialization is really needed.

The *Parameter Conditional* occurrences are many in the Collection hierarchy. In some cases the detection of this smell indicates places with clear possibilities for applying polymorphism, like the example shown in A.19.

In the following sections we discuss in more detail the results for the smells *Unused Parameters*, *Parameter Clump*, *Inappropriate Interface* and *Duplicated Code*.

4.2.2 Refactoring Unused Parameters

As we can see in table 4.2, all three case studies present unused parameters. We discuss them here in more detail.

Collection

The Collection hierarchy contains five instances of the unused parameter bad smell:

- The `intersectsFromY:toY:` method in class `SPActiveLines` does not use its first formal parameter.
- The `firstIntersectionBetween:and:` method in class `SPActiveLines` does not use its first formal parameter.
- The `errorSubscriptsBounds:` method in class `List` does not use its single formal parameter.
- The `asNumberFromFormatString:` method in class `CharacterArray` does not use its single formal parameter.
- The `extraAttributesForDefinitionOf:` method in class `GeneralNamespace` does not use its single formal parameter.

Five *removeParameter* refactorings are proposed based on these bad smells. A closer investigation of the source code revealed that they should all be applied to make the code more consistent.

HotDraw

The unused parameter bad smell occurred six times in the *HotDraw* framework:

- The `ToolState` class defines a number of methods that deal with events in the user interface, such as `pseudoEvent:`, `immediateEvent:`, `boundsEvent:` and `keyReleasedEvent:`. Each of these methods defines a parameter that represents the event that occurs. This event parameter is not used in 4 methods of the `ToolState` class nor in any of the other implementations with the same selectors.

- The `canConnectFromPoint:` and `canConnectToPoint:` methods in class `Figure` do not use their single formal parameter.

Based on these 6 bad smells, 6 *removeParameter* refactorings are proposed. The proposed refactorings for `canConnectFromPoint:` and `canConnectToPoint:` were effectively applied to remove the parameters of these methods in class `Figure`.

Because the other 4 bad smells and proposed refactorings all occurred in the same `ToolState` class, we decided to investigate this class in more detail. A closer look to the class reveals that it defines 24 methods for handling events, which are also defined in class `Controller` from the Smalltalk's User Interface. Curiously, none of these selectors use the parameters that are given to them. If `ToolState` becomes a subclass of `Controller`, all unused parameter occurrences simply disappear. Therefore, the actual refactoring that needs to be applied is *changeSuperclass*.

SOUL

48 occurrences of the unused parameter bad smell were identified in the SOUL application. We explain the classes which presents the highest amount of unused parameters.

- A number of reduceAction selectors like `reduceActionForLiteral3:` or `reduceActionFor-Temporaries2:` are defined in the Parser classes `SoulParser`, `SmalltalkTermParser` and `QuotedCodeParser`. The parameter is always of type `nodes`. There are 300 selectors with the `reduceAction` prefix, and in 25 of these cases, the methods does not use the `nodes` parameters they receive.
- The `SymbioticMessageTerm` class is the only one that defines the selector `smalltalk-MessageResolveWithUnknownReceiverIn:startAt:`, and does not use any of its parameters.
- Both classes `AbstractTerm` and `PosVariable` defines the selector `unifyWithUnderscore-Variable:inEnv:myIndex:hisIndex:inSource:` without using any of its parameters.

Based on these occurrences, 48 *removeParameter* refactorings are proposed to remove the unused parameters.

The 25 occurrences found in the selectors with prefix `reduceAction`, correspond to generated code, based on a given grammar. Any refactoring is impractical in this kind of code, in particular the *removeParameter* refactoring.

In the case the selectors from the `SymbioticMessageTerm` class contains the message “self no-tYetImplemented”. So this occurrence really corresponds to the *Speculative Generality* smell, where functionality is being planned but is not yet implemented.

Regarding the `AbstractTerm`, after consulting with the developers we decided not to apply the refactorings. The reason is that the keywords `inEnv:myIndex:hisIndex:inSource:` are considered a coding convention for all selectors related to unifying code. We can apply the *removeParameter*, but this would confuse the usage of the selectors and deteriorate the code.

For the other 13 cases we decided to apply the *removeParameter* refactoring, as in this case there is no reason for the parameters remain unused.

4.2.3 Refactoring Parameter Clumps

When detecting the smell *Parameter Clump*, we found one occurrence in the Collection hierarchy, and one in the SOUL application. The threshold values considered that the clump should consist of at least four parameters, present in at least four selectors in the class.

Collection

We found the clump `private:constant:category:initializer:` in four selectors of class `GeneralNameSpace`, which are also defined in the unrelated class `Class`. In both cases, the selectors where the clump is present are related to the creation of static and shared variables, and the parameters are treated consistently in all the implementations.

The proposed refactoring is *introduceParameterObject*, that receives as parameters the classes `GeneralNameSpace` and `Class`, which contains the clump, and the list of parameters that compose the the clump: <`private:, constant:,category:,initializer:`>. We apply the refactoring, as conceptually all the parameters are attributes related to the defined variables, so they can be transformed into instance variables of a coherent object.

SOUL

The clump `inEnv:myIndex:hisIndex:inSource:` was present in 96 selectors of 15 classes, the majority in the `AbstractTerm` hierarchy as shown in figure 3.3. All selectors containing the clump are related to the unification of terms.

The proposed refactoring is *introduceParameterObject*, which we do not apply, even if the parameters are treated consistently in all selectors. Due the fact that this smell was present in 96 selector, it had already been recognized by the developement team. However all these methods are heavily used during the evaluation of logic queries, so the introduction of an object that needs to be created several times in this process is not acceptable in terms of performance.

4.2.4 Refactoring Inappropriate Interfaces

For the *Inappropriate Interface* smell, we found occurrences in the three case studies. The threshold values involved inappropriate interfaces with at least two subclasses sharing at least two selectors that are not present in the superclass nor understood by it.

Collection

Collection contains 10 inappropriate interfaces with different characteristics:

- There are three cases of inappropriate interfaces in the Collection hierarchy where all the subclasses share an interface that is not present in the superclass. Class `CharacterArray` has two subclasses `Text` and `String` which contain the selectors `asParagraph` and `hasChangeOfEmphasis` that are not defined in `CharacterArray`. The second case is the class `GeneralNameSpace`, which does not declare the methods `reorganize` and `absoluteReference` defined in its two subclasses `NameSpaceOfClass` and `NameSpace`. The third case are the subclasses of `ByteArray`, `BinaryStorageBytes` and `BOSSBytes` that implement the methods `longAt:put`, `replaceBytesFrom:with:`, `unsignedLongAt:`, `swapBytesWidth:`, `shortAt:`, `unsignedShortAt:put:`, `shortObjectAt:from:baseIndex:`, `swapColumn:with:`, `objectAt:-from:baseIndex:`, `unsignedLongAt:put:`, `unsignedShortAt:` and `longAt:`, which are not implemented in the superclass. Moreover, `longAt:put`, `replaceBytesFrom:with:`, `unsigned-LongAt:`, `swapBytesWidth:`, `shortAt:`, `unsignedShortAt:put:`, `swapColumn:with:`, `unsigned-LongAt:put:`, `unsignedShortAt:` and `longAt:` are duplicated completely in the two classes.
- Interfaces shared by some subclasses but not all of them. For example the `ArrayedCollection` class contains two inappropriate interfaces which do not overlap. The subclasses `RunArray` and `List` contain the selectors `addFirst:` and `addLast:`, while the subclasses `TableAdaptor` and `TwoDList` contain eleven selectors that are not defined in `ArrayedCollection`.
- Interfaces shared by subclasses that overlap. The first occurrence consists in three subclasses `FourByte`, `ByteEncoded` and `TwoByteString` define the methods `asIntegerArray` and `sizeInBytes` which are not part of the superclass. The second occurrence consists on four selectors including the mentioned ones plus `byteAt:` and `byteAt:put:` which are defined in the subclasses `FourByte` and `TwoByteString`.

For interfaces present in all subclasses, three *addMethods* refactorings are proposed, containing the interface that is not present in the superclass. The three refactorings were applied, defining the methods as abstracts in the superclass. For the subclasses of *ByteArray*, we expect the detection of duplicated code to propose refactorings that remove the duplicated code, so we do not take measures against it here.

For the cases, where inappropriate interfaces occur in some of the subclasses only, seven *addMethods* refactorings and seven *addClass* refactorings are proposed. For the cases where the interfaces does not overlap we decide to apply the *addClass* refactorings.

For the case where there is overlap between the inappropriate interfaces results, we apply the *addClass* refactoring for the interface shared by the three subclasses. This reduces the amount of selectors involved in the inappropriate interface detected in the two subclasses *FourByte* and

TwoByteString. We can add an intermediate class by hand to solve this smell (as the conditions have changed, the *addClass* with four methods is no longer valid), or detect the bad smell again to obtain new proposals, now with the new scenario.

HotDraw

Hotdraw only contains one inappropriate interface, of two selectors `displayOutlineOn:` and `displayFilledOn:` that are defined in two subclasses of class `Figure`, `RectangleFigure` and `EllipseFigure`.

The proposed refactorings are one *addMethods* to the class `Figure`, and one *addClass* between `Figure` and the subclasses that share the interface.

A closer inspection of these methods reveals that they have segments of code in common. If we search for other refactoring opportunities, this code duplication will be detected, and an intermediate class will also be proposed for extracting the common code. In both cases the intermediate class will be proposed and will be usefull for containing the duplicated code or the missing interface. We decide to apply the refactoring, making it easier to extract common behaviour in future refactorings.

SOUL

SOUL contains only one inappropriate interface that consists of the three selectors `fixClean:`, `printForCompileOn:` and `prettyPrintOn:scope:` present in the two subclasses of `Concludable-Clause` class: `Fact` and `Rule`.

The proposed refactoring is an *addMethods* refactoring with the three selectors as parameters. A closer inspection made by the developers, showed that the method `printForCompileOn:` was actually “dead code”, remaining from a previous release but now obsolete. We decided to apply the refactoring *removeMehod* over the obsolete method, and the refactoring *addMethods* with the other two selectors as parameters. The methods added in the superclass are declared as abstract.

4.2.5 Refactoring Duplicated Code

Due the large amount of duplicated we found in our experiments, we will give one example of duplicated code in the same class, in subclasses and in siblings for each case study. The threshold considers duplicated code containing more than two statements.

Collection

Duplicated Code in Class The class `SequenceableCollection` contains duplicated code in the methods `=` and `isSameSequenceAs:`, where both of them receive a parameter named `otherCollection`. The difference is the first statement of the method `=`, that does some extra computations. The other three statements are exactly the same as the code of `isSameSequenceAs:`, as we can see in the method definitions:

```
isSameSequenceAs: otherCollection
    | size |
    (size := self size) = otherCollection size ifFalse: [^false].
    1 to: size do: [:index |
        (self at: index) = (otherCollection at: index) ifFalse: [^
           false]].
    ^true


= otherCollection
    | size |
    self species == otherCollection species ifFalse: [^false].
    (size := self size) = otherCollection size ifFalse: [^false].
    1 to: size do: [:index |
        (self at: index) = (otherCollection at: index) ifFalse: [^
           false]].
    ^true
```

The proposed refactoring for this case, when one duplicated code is a segment and the other is a whole method is to extract the segment with *extractMethod*, and as the extracted code already exists in the class, our extension of the Refactoring Browser's *extractMethod* refactoring will automatically replace the code with a call to the existing code instead of creating a new.

We apply this refactoring making the code clearer, increasing the possibility for reuse of the system. The resulting code is the following, where the `isSameSequenceAs:` method remains the same:

```
= otherCollection
    self species == otherCollection species ifFalse: [^false].
    ^self isSameSequenceAs: otherCollection
```

Duplicated Code in Subclasses Class `Dictionary` and its subclass `PoolDictionary` contain the same last four statements in the selectors `add:` and `simpleAddBinding:` respectively. The code contains assignments, conditionals and other messages and differs only in the name of the parameter they handle, as we can see in the code:

```
"add: selector from class Dictionary"
    index := self findKeyOrNil: key.
    element := self basicAt: index.
    element == nil
        ifTrue: [self atNewIndex: index put: anAssociation]
        ifFalse: [element value: anAssociation value].
    ^anAssociation

"simpleAddBinding: selector from class PoolDictionary"
    index := self findKeyOrNil: key.
    element := self basicAt: index.
    element == nil
        ifTrue: [self atNewIndex: index put: aVariableBinding]
        ifFalse: [element value: aVariableBinding value].
    ^aVariableBinding
```

The proposed refactoring is *extractMethod* in both classes, which ends up with the duplicated code replaced with a message call to a method defined in the superclass. In this case we named the extracted method `actionForAdding:key:`. The resulting code is the following, which replaces both duplicated segments:

```
ŝelf actionForAdding:  aVariableBinding key:  key
```

Duplicated Code in Siblings The siblings `BinaryStorageBytes` and `BOSSBytes` are the subclasses of `ByteArray`, and they contain several identical methods. The ones detected with the threshold values are `replaceBytesFrom:with:`, `shortAt:` and `swapColumn:with`, where all of them are defined in the superclass as consequence of the `addMethods` refactorings from the *Inappropriate Interface* smell. A closer look to the subclasses allows us to discover several duplicated methods of one statement.

The proposed refactoring are three *replaceDefinitionInSuperclass* and three *addClass* refactorings. We decide to apply the first one, and also to replace the definition in the superclass of the duplicated methods with one statement.

HotDraw

Duplicated Code in Class The class `CompositeFigure` contains three duplicated statements at the end of the selectors `add:` and `addAll:`. This is half of the size of the methods, and includes messages send to self, and a return at the end. A closer look to the selector `addLast:` reveals that its last three statements are equivalent to the ones detected as duplicated, but instead of assigning the value send to the messages before, as in the other cases, it calculates it in the self send itself. The equivalent codes of the three methods are the following:

```
"add: "
    self invalidateRectangle: figureBounds.
    self mergeBounds: figureBounds.
    ^aFigure

"addAll:"
    self invalidateRectangle: figureBounds.
    self mergeBounds: figureBounds.
    ^aCollection

"addLast:"
    self invalidateRectangle: aFigure bounds.
    self mergeBounds: aFigure bounds.
    ^aFigure
```

By applying the refactoring `extractToTemporary` applied to the message, we leave the code identical to the other two methods.

The proposed refactoring is to extract both segments into a common selector with *extractMethod*. We apply this refactoring, naming the new selector `handlingFigureBounds:for:`, and manually extract the code of the `addLast:` method in order to unify more duplicated code. This leaves the duplicated code replaced by a the call:

```
ŝelf handlingFigureBounds: figureBounds for: aFigure
```

Where `handlingFigureBounds:for:` is defined in the same class `CompositeFigure`.

Duplicated Code in Subclasses The class `Figure` and its subclass `TextFigure` contain three statements of equivalent code in the selector `menuAt:`. The problem is that our approach for detecting duplicated code allows different message sends whenever they are handled consistently in the code. In this case, the three statements contains three different messages to self:

```
"class Figure"
    self addLineColorMenuTo: mb.
    self addFillColorMenuTo: mb.
    self addLineWidthMenuTo: mb.

"class TextFigure"
    self addJustificationMenuTo: mb.
    self addVisibilityMenuTo: mb.
    self addStyleMenuTo: mb.
```

The proposed refactoring involves the extraction of these differences, generating three new methods that return different messages. This does not contribute to the clarity of the system, so we do not apply this refactoring.

Duplicated Code in Siblings The siblings `RectangleFigure` and `EllipseFigure` contains identical selectors `displayFigureOn:`, and identical bodies for the methods `rectangle:` and `ellipse:`. A previous refactoring had already created an intermediate class between these siblings and the superclass `Figure`, which contains several other subclasses.

There are two refactoring proposed: *extractMethodToSuperclass* with the selector `displayFigureOn:` as parameter, and a *renameMethod* composed with a *extractMethodToSuperclass* having the `rectangle:` and `ellipse:` selectors as parameters. We decide to apply both of them for reducing duplication and grouping more common behavior between these two siblings.

SOUL

Duplicated Code in Class The class `UppedObject` contains two methods with the same body of three statements. The selectors are `prettyPrintAsKeywordOn:scope:` and `prettyPrintAsFunctorOn:scope:`.

Since both methods are used in the system, the proposed refactoring does not include to remove one of the methods. The proposed refactorings maintain the interfaces of the methods, proposing two *extractMethod* for replacing the body of one of them with a message call to the other.

We applied the *extractMethod* to the body of the `prettyPrintAsFunctorOn:scope:`, replacing it with a call to `prettyPrintAsKeywordOn:scope:`.

Duplicated Code in Subclasses The class `Soul.ListTerm` and its subclass `PairTerm` contains two nearly dupplicated methods: `listPrintOn:` and `printOn:` respectively. The differences are that the method in the subclass contains one extra statement, and there is a different literal in the segment. While the superclass writes ',' at the beginning of a stream, the subclass writes '<':

```
"class ListTerm"
listPrintOn: aStream
    aStream write: ','.
    termsequence terms first printOn: aStream.
    termsequence terms last species == Factory current makeVariable
        ifTrue:
            [aStream write: '|'.
            termsequence terms last printOn: aStream]
        ifFalse: [termsequence terms last listPrintOn: aStream]
```

```
"class PairTerm"
printOn: aStream
    aStream write: '<'.
    termsequence terms first printOn: aStream.
    termsequence terms last species == Factory current makeVariable
        ifTrue:
            [aStream write: '|'.
            termsequence terms last printOn: aStream]
        ifFalse: [termsequence terms last listPrintOn: aStream].
    aStream write: '>'
```

The refactoring proposes to extract the differences (with the same name), defining the message `firstString` both in the superclass and overriding it in the subclass. Now the codes are equivalent, and the refactoring extracts all statements, except the last one, in the `printOn:` method, replacing the code with a call to the method `listPrintOn:` defined in the superclass. The resulting code is the following:

```
"class ListTerm"
listPrintOn: aStream
    aStream write: self firstString.
    termsequence terms first printOn: aStream.
    termsequence terms last species == Factory current makeVariable
        ifTrue:
            [aStream write: '|'.
            termsequence terms last printOn: aStream]
        ifFalse: [termsequence terms last listPrintOn: aStream]

"class PairTerm"
printOn: aStream
    self listPrintOn: aStream.
    aStream write: '>'
```

Duplicated Code in Siblings The siblings `TermSequence` and `MultiPartFunctor` are the only subclasses of class `AbstractTerm`, which has several more subclasses. They contain equivalent code in their selectors `printForCompileOn:`. The difference is the structures of the last line in each method. The rest of the body, including a complex block expression and several message sends, are equivalent, containing only one different literal, and one different message send, as can be seen in section 3.2.7.

The refactoring includes the extraction of both the literal and the message send defined in the siblings and the extraction of the common code. We decided to push up the extracted code to an intermediate class.

4.3 Discussion

These experiments provided a number of interesting results: We found all kind of bad smells present in the three case studies, and several of them pointed to situations that were worthy of restructurings. In this section we also discuss the possibilities of cascade refactorings and the scalability of our approach. This section is mainly based on the discussions we maintained in [53].

4.3.1 Cascaded Refactorings

During the analysis of the results, we observed that in some cases, the application of a particular refactoring opens up possibilities for performing other refactorings. For example, by adding an intermediate superclass for solving some of the *Inappropriate Interface* bad smells, it becomes possible to factor out common behaviour related to duplicated code by applying *pushUpMethod* refactorings, like the case present in HotDraw explained in 4.2.4 and 4.2.5. In this case, the application of one particular refactoring opens possibilities for other refactorings to be applied. This phenomenon is called *cascaded refactoring opportunities* [53]. The logic meta programming approach naturally allows to detect cascaded refactoring opportunities. The example we mention can be easily expressed as follows:

```
proposeRefactoring(?class, pushUpMethod, <?siblings,?selector>) if
     proposeRefactoring(?, addClass, <?class,?>),
     duplicatedCodeInSiblings(?class,?selector,?siblings)
```

where the `duplicatedCodeInSiblings` predicate determines which the `?siblings` of class `?class` define a duplicated method named `?selector`.

4.3.2 Scalability

In our experiments we considered three small to medium-scaled case studies, without experiencing problems. However, a different situation is to analyze large software systems. In this section we discuss how to adapt our approach to larger software systems in terms of efficiency and the amount of proposed refactorings.

Detecting Bad Smells more Efficiently

A problem of our approach is that some of the logic rules to detect bad smells are very intensive in computations. This makes it unfeasible to check them on large systems, as it would take too much time before all bad smells have been detected. Therefore, we suggest to combine our approach with more lightweight approaches that detect bad smells more efficiently.

We can use object-oriented metrics to detect those places in the code that are likely candidates for bad smells, as we saw in the *Related Work* section 2.3. Once these locations in the source code have been identified, we can use our logic rules to analyse these parts of the program in more detail.

For detecting code duplication, there are more efficient approaches available than with logic predicates, as we saw in the *Related Work* section. We can use some of these techiques for detecting duplicated code, and analyze the results with our logic predicates.

4.3.3 Managing the list of proposed refactorings

In practice, the result of our bad smell detection may lead to a large list of proposed refactorings. Considering the smells analyzed in detail, only for the Collection hierachy we obtain 117 proposed refactorings. This list is already difficult to handle, and if we consider more bad smells and larger software systems this number will certainly increase. Even if we configure our threshold values for mitigating this effect up to some degree, the amount of results we obtain is still unmanageable. Moreover, one particular bad smell can often be remedied by a multitude of refactorings, as shown by the refactorings proposed for the *inappropriate interface* or *duplicated code* bad smells. The list of proposed refactorings will increase even more if cascaded refactoring opportunities (as discussed previously) are taken into account.

We identify different ways to manage the size of this list:

- Induce an order on the proposed refactorings: the smell weighing predicate we defined in section 3.2.8 can be used to determine which of the proposed refactorings will produce a higher improvement on the code. The refactorings proposed for solving the strongest smells should be presented in the first place, so the user can be sure that their application will benefite the code more than the ones that follow.
- Present clearly the proposed refactorings that overlap, and eliminate the ones that become obsolete. For example a *removeMethod* refactoring overlaps with a number of *removeParameter* refactorings for the same method, since the application of the former refactoring makes the latter one obsolete. Whenever a refactoring is applied, the list is recomputed dynamically to remove obsolete refactorings that appear later in the list. To achieve this, we need to compute dependencies between different refactorings, which is considered future work.
- Check for refactoring opportunities and apply the corresponding refactorings regularly (as suggested in [5]). Checking for refactoring opportunities should happen at the same time as unit testing. The unit tests will then ensure the behavioural correctness of the application, whereas the bad smell tests will ensure its "structural" correctness.
- If a large list of bad smells and proposed refactorings exists, it might also be helpful to analyse the bad smells visually. This approach is suggested by van Emden and Moonen [54], who combine the detection of bad smells in *Java* with a visualisation mechanism.

4.3.4 More accurate proposed refactorings

Our current approach considers the bad smells that are detected and proposes a combination of mutually exclusive refactorings that can be applied to remove them. By taking into account more information on the context of the bad smells, we believe we can reduce the number of refactorings in this combination, and help the developer in choosing the appropriate one. The *duplicated method* bad smell for example, yields different proposed refactorings that take into account the characteristics of the class that can be determined with metrics or deeper logic analysis. This contextual information can also involve the simultaneous occurrence of the same or other bad smells in the same or related software entities. For example, the *inappropriate interface* and *duplicated method* bad smells in the Collection hierarchy overlap to some extent: the `replaceBytesFrom:with:`, `shortAt:` and `swapColumn:with:` methods are duplicated and are not part of the interface of the `ByteArray` superclass. If we refactor the duplicated code first, the other smells disappear.

4.4 Conclusion

As we can see in tables 4.1 and 4.2, the detection of bad smells gave a number of occurrences in the three case studies. The kind of bad smells present in each of them are consistent with the characteristics of the application. For example the Collection hierarchy has more bad smells related to the size of its code entities than the other cases. This can be related to the fact that this hierarchy is a large library of container classes with a lot of funcionality. The high performance provided by the Collection hierarchy may also have an influence in the larger amount of duplicated code and interface related smells. On the other hand SOUL is an application which is still being developed, so it is reasonable that it presents more smells related to speculative generality for example. The HotDraw framework, smaller in size than the other two cases presents a considerably less smells in almost every category that the other cases. This is probably related to the fact that HotDraw has been carefully designed with not much concern about performance requirements.

From the cases that were studied in deepness, we learned that the proposed refactorings were not always the ones that were needed for solving the bad smells. In some cases, the user needed to perform some extra analysis to identify which refactoring was necessary. This was for example the case for the `ToolState` class in HotDraw that gave rise to 4 proposed *removeParameter* refactorings, while it sufficed to apply a single *changeSuperclass* refactoring to solve the problem. This confirms the idea that refactorings should not be performed automatically, and that they are complementary to human inspection. We can try to express the reasoning that made us propose the *changeSuperclass* with logic rules, but we will never be able to cover all the cases a human mind can imagine. On the other cases, the proposed refactorings were quite appropriate, and after inspecting the code, we decided to apply most of them.

Based on our results, we can conclude that logic meta programming has proved to be a really good method for detecting bad smells and proposing refactorings based on statical analysis.

Chapter 5

Conclusion

5.1 Summary and Conclusion

Applying refactorings in order to improve the internal structure of software is becoming a common practice in diverse object oriented software development environments, and there are several tools that perform refactorings in an automated way. However there are much more decisions involved than just applying the refactoring to the code. For example deciding *when* to apply a refactoring, detecting *where* to apply it, *which* refactoring should be applied, assessing the contribution of possible refactorings or evaluating the effect of the applied refactorings.

Some of these steps are difficult to automate, like assessing the effect of the refactorings before applying them. This involves having very accurate definitions of the postconditions of the refactorings, a situation that is not yet achieved by refactoring tools. The decision of *when* to apply a refactoring also depends a lot on management decisions, and the common advice is to refactor whenever it is necessary or when performing unit testing.

The contributions of this thesis are the following:

- The implementation of a logic framework for refactoring, that supports important steps in the refactoring process.
- Our approach based on logics, is language independent to a large extent.
- Our approach integrates existing refactorings provided by the Refactoring Browser, but can easily be redefined for other existing refactoring tools.
- The experiments we performed over the case studies proved that our approach is useful in practice.

In this work we present a tool integrated with VisualWorks Smalltalk that detects bad smells in the code for recognizing places where refactorings could be useful. The smells are analyzed in order to propose appropriate refactorings with the necessary parameters. This complements the user's intuition and helps him to apply refactorings in a more informed way, by detecting automatically a number of refactoring opportunities that would be very hard to recognize by manual inspection. In our approach we did not implement the refactorings themselves, but we rely on existing behavior preserving transformations provided by the Refactoring Browser. To evaluate the quality of applied refactorings, we can run the bad smells again and see the bad smell is no longer present, and that other bad smells were not introduced.

The technique we used for querying and analyzing the code at the level of structural relations is logic meta programming, a variant of logic programming. By reasoning with logic predicates over source code entities, we managed to define a number of bad smells, propose refactorings for them and we even defined composite refactorings in order to apply more complex ones than the ones defined in the Refactoring Browser.

Our approach is language independent up to some degree. Even if most of the predicates SOUL provides can handle object oriented entities from any language, some of them rely on the dynamically typed characteristics of Smalltalk. The relation with the refactorings of the Refactoring Browser is also relevant, but the refactorings we apply can easily be redefined over other tools with similar characteristics.

In order to validate our approach, we performed experiments over three case studies of medium-sized object-oriented applications in Smalltalk: The *SOUL* application itself, the *Collection* hierarchy of Smalltalk Visualworks and the framework *HotDraw*. We detected a number of smells, some of more relevance than others, but all of them indicating sections of code that were worthy of attention. Even more, many of the proposed refactorings were quite useful in order to solve the bad smells and improve the legibility of the code. The experiments also reflected some characteristics of the selected case studies. We related many of the smells present in the Collection hierarchy with efficiency reasons and the fact that it is a large library of utility classes with a lot of functionality. In the SOUL application we found more smells related to speculative generality than in the other applications, as it is an application still in development process. The HotDraw framework, containing noticeably less smells than the others, still had an interesting number of situations that could be improved.

5.2 Future Work

Some possible directions of future work in the areas of bad smell detection and refactorings are analyzed in this section:

- Investigate the relation with design patterns: It is possible to detect deteriorated design pattern implementations and propose adequate refactorings to correct them.

- More complex cases that were not taken into account for proposing refactorings can also be stated as logic rules. For example, one *changeSuperclass* was a more suitable refactoring than four *removeParameter* in one of the experiments we performed (section 4.2.2). This refactoring was determined by using human intuition, but we are convinced that this kind of information can be encoded explicitly in logic rules as well, allowing us to further increase the accuracy or usefulness of the proposed refactorings.

- Integrate our approach with dynamic analysis, could help for type detection and smells that need run time information [23], such as *Incomplete Library* smell or the *Semantic Duplication* smell mentioned in section 3.2.2.
- Part of our bad smell definitions can be directly ported to the version of SOUL that analyzes Java code. In this statically typed environment, all bad smells that we did not considered because the lack of typing information could be detected.
- To weight the detected bad smells is an important step for presenting the results properly to the user. These results can be displayed either in the form of a list, as we do with our tool, or in the form of graphic results as proposed in [54]. The predicate we presented in section 3.2.8 for weighing smells is very simple, but it is possible to define a more precise weighing algorithm.

Refactoring Related Future Work

- How to attach quality characteristics to the software (using metrics), determine whether a refactoring improves the quality, and check whether the refactored software has improved its quality characteristics.
- State the preconditions and postconditions of every refactoring with logical predicates, in order to ease the composition of refactorings. The usage of logic predicates enforces the language Independence of the approach and it results very natural in this context, as the assertions are usually declared in logical manners and the translated to the language of interest.
- Investigate what happens with composite refactorings if refactorings are changed or introduced. What is the relation when adding or changing preconditions.
- Compute the dependencies between refactorings before presenting the possibilities to the user. This allows to separate refactorings that overlap (if one is applied, the other one is no longer valid) and to identify more clearly which refactoring opportunities can or will give rise to other opportunities. This means to investigate which refactorings depend upon other ones, and how and why this is the case.
- Investigate the efficiency of the application after applying refactorings to the code. Sometimes bad design smells are preserved in the code because the developers think that performance of the application will be affected if they refactor them. A deeper investigation of the effect on efficiency of different refactorings would help to dissipate these doubts.

Bibliography

[1] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proceedings of the 13th Australian Software Engineering Conference, ASWEC 2001*. IEEE Computer Society, August 2001.

[2] Dave Astels. Refactoring with UML. In *Proceedings of the 3rd International Conference eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002. Alghero, Sardinia, Italy.

[3] Kevin Atkinson. 12dicts. http://wordlist.sourceforge.net/12dicts-readme.html.

[4] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[6] Keith H. Bennett and Václav Rajlich. A staged model for the software lifecycle. *IEEE Computer*, 33(7):66–71, July 2000.

[7] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 77–81, 2002. Alghero, Sardinia, Italy.

[8] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Coordinated distributed diagram transformation for software evolution. *Electronic Notes in Theoretical Computer Science*, 72(4), 2002.

[9] John M. Brant. Hotdraw. Master's thesis, University of Illinois at Urbana Champaign, 1995.

[10] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In *on Generative Programming and Component Engineering (GPCE)*, 2002.

[11] Gerardo Canfora, Aniello Cimitile, Maria Tortorella, and Malcolm Munro. A precise method for identifying reusable abstract data types in code. In *Proceedings of the Proceedings of the International Conference on Software Maintenance 1994*, pages 404–413. IEEE Computer Society Press, September 1994.

[12] Eduardo Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings of the European Conference on Object-Oriented Programming, ECOOP 1992*, pages 114–132. Springer, 1992.

[13] Cincom. Smalltalk VisualWorks. http://www.cincomsmalltalk.com.

[14] Mariano P. Consens, Isabel F. Cruz, and Alberto O Mendelzon. Visualizing queries and querying visualizations. *SIGMOD Record*, 21(1):39–46, 1992.

[15] Borland Software Corporation. Together control center. http://www.borland.com/together/controlcenter.

[16] IntelliJ Corporation. Intellij idea. http://www.intellij.com/idea.

[17] João G. Del Valle. Towards round-trip engineering using logic metaprogramming. Master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, 2003.

[18] Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog: The Standard Reference Manual.* Springer-Verlag, 1996.

[19] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, may 2001.

[20] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of the International Symposium on Software Architectures and Component Technology 2000*, pages 207–224, 2000.

[21] Bart Du Bois and Tom Mens. Describing the impact of refactorings on internal program quality, 2003.

[22] Thomas Dudziak and Jan Wloka. Tool-supported discovery and refactoring of structural weaknesses in code. Master's thesis, Technical University of Berlin, 2002.

[23] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *Transactions on Software Engineering*, 27(2):1–25, February 2001.

[24] Johan Fabry. Supporting development of enterprise javabeans through declarative meta programming. In *Object-Oriented Information Systems.* Springer, 2002.

[25] Richard Fanta and Václav Rajlich. Reengineering an object oriented code. In *Proceedings of the IEEE International Conference On Software Maintenance*, pages 238–246. IEEE Computer Society, 1988.

[26] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[27] Erich Gamma, Richard Heml, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[28] William G. Griswold. *Program restructuring as an aid to software maintenance*. PhD thesis, University of Washington, 1991.

[29] Instantiations. jFactor. http://www.instantiations.com/jfactor.

[30] Object Technology International. The eclipse platform. http://www.eclipse.org.

[31] Stephen Johnson. Lint, a c program checker, 1978.

[32] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proceedings of the International Conference on Software Maintenance ICSM 2001*, pages 736–743, 2001.

[33] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the IEEE International Conference on Software Maintenance ICSM*, pages 576–585. IEEE, 2002.

[34] Mikael Lindvall and Kristian Sandahl. How well do experienced software developers predict software change? *The Journal of Systems and Software*, 43(1):19–27, 1998.

[35] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253, 1996.

[36] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.

[37] Tom Mens, Tom Tourwé, and Francisca Muñoz. Beyond the refactoring browser: Advanced tool support for software refactoring. In *Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2003*. ACM, 2003.

[38] Ivan Moore. Guru - A tool for automatic restructuring of self inheritance hierarchies. In *Proceedings of TOOLS-USA'95, Santa Barbara, (CA), USA*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[39] Mel Ó Cinnéide and Paddy Nixon. Composite refactorings for java programs. Technical report, Department of Computer Science, University College Dublin, 2000.

[40] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana Champaign, 1992.

[41] Santanu Paul and Atul Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, September 1994.

[42] .Net Refactoring. C# refactoring tool. http://dotnetrefactoring.com.

[43] Matthias Rieger, Stéphane Ducasse, and Georges Golomingi. Tool Support for Refactoring Duplicated OO Code. In *Object-Oriented Technology (ECOOP'99 Workshop Reader)*. Springer-Verlag, 1999.

[44] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company, April 1996.

[45] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 1997.

[46] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana Champaign, 1999.

[47] Chris Seguin. jRefactory. http://jrefactory.sourceforge.net.

[48] Frank Simon, Frank Steinbruückner, and Clause Lewerent. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, 2001.

[49] Xtreme Simplicity. C# refactory. http://www.xtreme-simplicity.net/CSharpRefactory.html.

[50] Sourceforge. Bicycle repair man, a refactoring browser for python. http://bicyclerepair.sourceforge.net.

[51] Gerson Sunyé, Damien Pollet, Ives Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *Proceedings of UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–138. Springer-Verlag, 2001.

[52] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of the International Symposium on Principles of Software Evolution ISPSE 2000*, pages 157–167. IEEE, 2000.

[53] Tom Tourwé, Tom Mens, and Francisca Muñoz. Detecting bad smells and refactoring opportunities with logic meta programming. In *Proceedings of the International Conference of Software Maintenance and Re-engineering*. CWI, 2003.

[54] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2002.

[55] William C. Wake. *Refactoring Workbook*. Addison-Wesley, August 2003.

[56] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.

[57] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[58] Xref-Tech. Xrefactory. http://xref-tech.com/xrefactory.

Appendix A

Bad Smell Definitions

The concrete examples given in this appendix are taken from the experiments erformed over the Smalltalk `Collection` library, the `SOUL` application and the `HotDraw` framework. More details are given in chapter 4 Experiments.

A.1 Large Class

For recognizing a large class, we have to detect two independent cases. *Too Many Instance Variables* and *Too Many Methods*. The detection is metric-based and uses the metrics Number of Instance Variables (NOIV) and Number of Implemented Methods (NOIM).

The bad smell predicates are the following:

```
badSmell(<largeClass,tooManyInstanceVariables>,
?class, <?class>, <?numberInstanceVariables>)
badSmell(<largeClass,tooManyMethods>, ?class,
<?class>, <?numberMethods>)
```

This predicates checks that the metric value present in the class is larger than the threshold value configurable by the user.

A.2 Long Method

Like *Large Class*, this is a highly metric-related smell. We take into account the amount of temporaries and statements of the method. We use the metrics Number of Statements (NOS) and Number Of Temporaries of Method (NOTM).

The bad smell predicates are:

```
badSmell(<longMethod,tooManyStatements>, ?class,
<?class>, <?numberStatements>)
badSmell(<longMethod,tooManyMethods>, ?class,
<?class>, <?numberMethods>)
```

A.3 Long Parameter List

As shown in table 3.1, we address two cases of the smell *Long Parameter List* defined by Fowler: *Too Many Parameters* and *Redundant Parameter*.

The long parameter lists are detected by using the metric Number Of Parameters of Method (NOPM).

For detecting that a parameter is being redundant, we check if all the callers are sending the same literal as parameter. This means we can use that literal value in the method body instead of receiving it as a parameter. The more general case mentioned by Fowler is that the result of the parameter can be requested through a message to an object we know. The difficulty is to know what the result of the parameter is, without type information. Note that the checking for literals is already pretty slow, as we have to check that all the senders in the system use the same arguments.

In figure A.1 we see that the selector `innerScanUpTo:ignore:` is defined in three classes. The selector is called from each class with the same literal arguments for both keywords `innerScanUpTo:` and `ignore:`. In this case, the parameters are not needed and they can be inlined in the code.

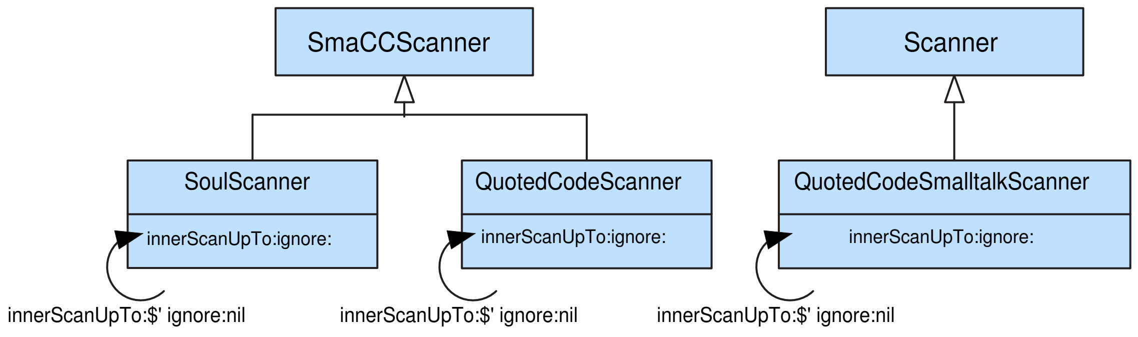


Figure A.1: *Redundant Parameters* in the selector `innerScanUpTo:ignore:`

The bad smell predicates for detecting these smells are:

```
badSmell(<longParameterList,longParameterList>,
?class, <?class,?selector>, <?numberParameters>)
badSmell(<longParameterList,redundantParameter>,
?class, <?class,?selector,?keyword>, <?numberCalls>)
```

When a parameter is redundant, we return the number of calls that are made to the selector that contains the parameter, as a sign of how bad the smell is. A method that has ten calls with the same literal is worse than one that has only one or two.

A.4 Magic Numbers

The *Magic numbers* smell is part of the *Duplicated Code* smell, and consists on a special number, string or other value that recurs many times in the code. For example, the class `ByteArray` in the `Collection` hierarchy of Smalltalk uses 21 times the magic number 32. Of course 32 has a meaning in the context of a `ByteArray`, but the code could be cleaner if the number were obtained from a method call.

The bad smell predicate has the form:

```
badSmell(<duplicatedCode,magicNumber>, ?class,
<?class,?magicNumber>, <?appears>)
```

This predicate retrieves the corresponding threshold value, and uses an the auxiliary `classWithMagicNumber` for retrieving the magic numbers of the class. This predicate takes all the methods from the given class, collect strings and numbers that are used by the methods, and count the total occurrences of magic numbers. This predicate returns one result per magic number found.

A.5 Code Duplication

This smell is explained in detail in section 3.2.7.

A.6 Data Class

A data class contains just fields, getters, setters and nothing else. Fowler proposes that a class with this characteristics should get more behavior with refactorings. An example of data class was found in the *HotDraw* framework, the `FigureAttributes` class, which contains 3 instance variables with its corresponding getters and setters.

The specific predicate for detecting a data class is the following:

```
dataClass(?class, ?numberOfMethods) if
    class(?class),
    one(instanceVariableInClass(?,?class)),
    not(
        one( and( or(methodInClass(?method,?class),
                      classMethodInClass(?method,?class)),
                    not(accessingMethod(?class,?method,?)) ))),
    metric(NOIM,class(?class),?numberOfMethods)
```

This predicate states that the class must have at least one instance variable, and there can be no method that is not an accessor or mutator. The generic bad smell predicate that uses the `dataClass` predicate is:

```
badSmell(<dataClass>, ?class, <?class>,
<?numberMethods>)
```

In this case, the indicator of how strong the smell is, is the amount of methods, this is accessors and mutators that the data class defines.

A.7 Parameter Clump

This smell is explained in detail in section 3.2.5.

A.8 Refused Inheritance

This case of the *Refused Bequest* smell states that a subclass is not using enough methods from the ones provided by the superclass. This is a minor smell considered by Fowler, but still it gives indications of a possible hierarchy disorder.

For detecting this smell, we consider the methods of the higher hierachy that the subclass actually *is* using. We want to know if the class is using methods understood by the superclass, or only using overridden versions. For this we count the occurrences of the following messages in the class:

- The calls made to the `self` variable, where the call is a method that is being overridden in the class, but contains a `super` call.
- The calls made to the `self` variable, where the call is a non-overriding method that is present in the superclass, or understood by it.

With this information, we calculate the ratio of calls made to `self`, that are not using the information provided by the superclass.

For obtaining more interesting results, we use user-defined threshold values for the allowd ratio and for the minimum amount of `self` calls.

For example in the class `TwoDlist` of the Collection hierarchy, there are 14 messages sent to `self`, where none of them is understood by the superclass.

The bad smell predicate is the following:

```
badSmell(<refusedBequest,refusedInheritance>,
?class, <?class>, <?refusedRatio,?numberSelfSends >)
```

A.9 Refused Interface

A subclass that refuses its interface is considered a much worse smell than a subclass that refuses its inheritance by a low usage of methods. The smell *Refused Interface* consists on a subclass that cancels methods of the superclass. This cancellation can have several forms: empty bodies, "should not implement" messages, error messages or abstracted methods. For example the class `SortedCollection` in figure A.2 cancels nine methods inherited from superclass `OrderedCollection`.

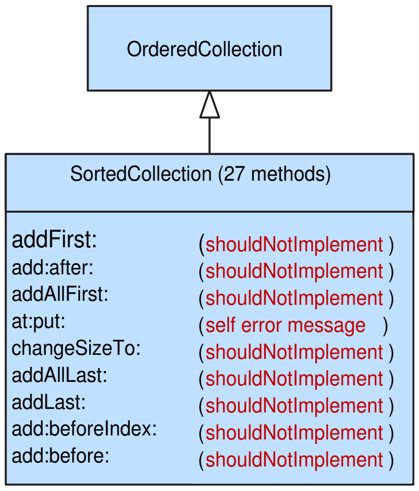

OrderedCollection
SortedCollection (27 methods)
addFirst: (shouldNotImplement)
add:after: (shouldNotImplement)
addAllFirst: (shouldNotImplement)
at:put: (self error message)
changeSizeTo: (shouldNotImplement)
addAllLast: (shouldNotImplement)
addLast: (shouldNotImplement)
add:beforeIndex: (shouldNotImplement)
add:before: (shouldNotImplement)

Figure A.2: Refused interface from the class SortedCollection

The code of the auxiliary predicate that calculates the refused interface of a class is the following:

```
refusedInterface(?class,?methodList,?ratio,+?thresholdRatio) if
```

```
    findall(?method,and(methodInClass(?method,?class),
        refusedMethod(?method)),?methodList),
    length(?methodList,?numberRefusedMethods),
    metric(NOIM,class(?class),?numberOfMethods),
    ratio(?numberRefusedMethods,?numberOfMethods,?ratio),
    greaterOrEqual(?ratio,?thresholdRatio)
```

This predicate finds all the methods in the class that are being refused. It also returns the list of the methods that were canceled, for future usage in refactorings.

The auxiliary predicate `refusedMethod` receives a method and checks that the method has a canceled form (empty body, should not implement, error message or abstract). Then it checks that at least one class in the upper hierarchy implements the method in a proper way (this means, it is not canceled) and that no class in the lower hierarchy implements it properly.

```
badSmell(<refusedBequest,refusedInterface>,
?class, <?class,selectorList>,
<?refusedNumber,?refusedRatio>)
```

A.10 Inappropriate Interface

This smell is explained in detail in section 3.2.6.

A.11 Feature Envy

Feature Envy is when a method of one class seems more interested in the attributes (usually data) of another class than in its own class.

Logically, we state this as follows: A given method uses a high ratio of accessor messages from a certain class, over all the accessors that are used in the class. For detecting which of the messages used by the class are accessors, we select the messages sent that does not include any argument, and check if they are accessors to an instance variable using the `accessor` predicate. In the case of accessors that are only defined in one class, this approach works fine. If the accessor is defined for more than one class, we only take into account one of them. This is related to the lack of typing information, as we do not know statically exactly which accessor is being used.

For obtaining more interesting results, we use user-defined threshold values for the minimum amount of accessors we should consider, and for the envy ratio of accessors sent towards a specific class.

The bad smell predicate for detecting this smell is:

```
badSmell(<featureEnvy>, ?class,
<?class,?selector,?enviedClass,?accessorList>>,
<?numberOfAccessors,?ratioEnvy>)
```

In this smell, the related entities are the class that is being envious, the class that is being envied, and the list of accessors of the second class that the first one is using.

The indicators are the ratio of envy, or usage of a specific class and the number of accessors.

A.12 Message Chains

This smell states that too many messages in a chain are present. This makes the code complicated to understand and to maintain. For example, this assignment was found in the `Collection` class in the `?numberOfMessages` method:
`article := self class name first isVowel ifTrue: ['an '] ifFalse: ['a '].`
It contains five messages sent in a row, which in this case is readable because many times Smalltalk messages looks like natural language, but in the general case is a situation we should avoid.

The code for detecting message chains is the following:

```
methodWithMessageChain(?method,?receiver,?msg,?args,?chainLength,+?
   thresholdNumber) if
    methodWithOuterSend(?method,?receiver,?msg,?args),
    nestedChain(?receiver,1,?chainLength),
    greaterOrEqual(?chainLength,?thresholdNumber).

nestedChain(?receiver,?num,?finalChainLength) if
    equals(?receiver,send(?nestedReceiver,?msg,<>)),
    add(?num,1,?chainLength),
    nestedChain(?nestedReceiver,?chainLength,?finalChainLength).

nestedChain(?receiver,?num,?num) if
    not(equals(?receiver,send(?nestedReceiver,?msg,<>)))
```

For detecting message chains, we first seek for messages that are not contained in any message send. Then we recursively traverse the parse tree while checking that the receivers of the message sends are also message sends. We return the chain and its length if it is larger than the user-defined threshold.

```
badSmell(<messageChain>, method(?method),
<?class,?selectorm,?chain,?chainLength>,
<?numberOfMessages,?ratioEnvy>)
```

The `?chain` value contains all the messages and receivers of the traversed chain. The problem is that these receivers can not directly be associated with classes, so we are not able to identify the other entities related to this smell.

A.13 Middle Man

A class is acting as a delegate, without performing useful extra work.

Logically we can say that more methods of a class are delegated to another object than handled by itself. By delegation we mean one of the following:

- The method has only one statement where the parameters are sent as arguments to another receiver.
- The method has only one statement, that returns the result of a sent message with the parameters as arguments.
- The receivers of the sent messages are neither the class or its superclass.

For example the class `Bag` contains four methods that are only asking results to other object:

```
valuesAndCountsDo: aBlock
    contents keysAndValuesDo: aBlock

removeAllOccurrencesOf: anObject ifAbsent: aBlock
    ^contents removeKey: anObject ifAbsent: aBlock

asSet
    ^contents keys asSet

includes: anObject
    ^contents includesKey: anObject
```

The predicate for detecting this smell is:

```
badSmell(<middleMan>, ?class,
<?class,?delegators>,
<numberDelegators,?ratioDelegators>)
```

A.14 Lazy Class

A class that is not doing enough work should be eliminated. It can be a class that was downsized during refactorings, or a class that was added because of changes, that were planned but not made. Like with the measured smell category, here we use only metrics to detect if a class is too light. We use the metrics Number of Instance Variables (NOIV), Number of Implemented Methods (NOIM) and Number of Statements (NOS).

For detecting the smell, we check that the amount of instance variables is smaller than threshold, that the amount of methods is smaller than threshold and that the size of each of these methods is smaller than threshold. All these threshold are user-defined values.

An extreme example we detected is the class `SignalCollection` from the Collection hierarchy that has no methods or instance variables at all. Other case is `LargeArray`, that has no instance variables and only one method that returns the class `Array`.

The bad smell predicate is the following:

```
badSmell(<lazyClass>, ?class, <?class>,
<?numberMethods,?numberInstanceVariables,?averageSizeMethods>)
```

A.15 Unused Parameter

This smell is explained in detail in section 3.2.4

A.16 Unused Instance Variable

This version of the *Speculative Generality* smell checks if there are unused instance variables present in the code. Logically, we retrieve the instance variables of a given class, and we check that it is not used in any methods of the whole hierarchy. The key auxiliary predicate in this case is `methodUsesVariable`, like in the *Unused Parameter* smell.

We found a few cases of unused instance variables in the SOUL application, but not more than for the same class.

The bad smell predicate is:

```
badSmell(<speculativeGenerality,unusedInstanceVariable>,
?class, <?class,?instanceVariable>,
<?numberNotUsed>)
```

In this case, the indicator is the number of all subclasses in the hierarchy of the class, including the class itself, that are not using the instance variable.

A.17 Abstract Method not Implemented

This smell detects methods declared as abstracts, which are not implemented (i.e. made concrete) anywhere in the hierarchy. It is a special case for the smell *Speculative Generality*, where probably a developer was thinking on extending the functionality, which never happened. These abstract methods should either be implemented somewhere in the hierarchy or simply be deleted.

Class `Factory` in the SOUL application has four abstract methods which are not implemented in its only subclass `StandardFactory`.

The code for detecting this smell was given as an example of using logic meta programming for detecting design flaws in section 2.1.2 of the Context chapter.

The bad smell predicate is:

```
badSmell(<speculativeGenerality,abstractMethodNotImplemented>,
?class, <?highestClass,?abstractSelector>,
<?numberNotImplemented>)
```

Like in other cases, we return a common result for all occurrences of the smell in the hierarchy. `?highestClass` corresponds to he highest class in the hierarchy that declares the method as abstract.

The indicator in this case is the number of times that the method is redeclared in an abstract way in the hierarchym this is, the number of times that the method is redefined but not implemented.

A.18 Odd Name

It is difficult if not impossible to determine when a name is odd or not appropriate for the context.

A heuristic for detecting uncommon word abbreviations or unclear names that does not mean anything is to check the words that are present in the names of classes or methods. We check if they are present in a large word list of common english words. We use the *2of12inf* list provided by [3], that contains english words taken from variate dictionaries, plus their inflections, amounting 81.520 words.

Even if this approach is not based on the structure of the system, it is easy to implement and fits in our logical framework.

For example, in the Collection hierarchy we detected classes with names like `Ephemeron` or `SPActive`, and also methods with names like `aisDisable` or `subrequisites`, which are not present in our word list.

In order to detect this bad smell for a given entity, we split the name by the capital letters (a common delimiter between words used in object oriented programming) and check the words in a valid dictionary list.

The generic bad smell predicates are the following:

```
badSmell(<speculativeGenerality,oddClassName>,
?class, <?class,?oddWordList>, <?numberOddWords>)

badSmell(<speculativeGenerality,oddMethodName>,
?class, <?class,?selector,?oddWordList>,
<?numberOddWords>)
```

Where `?oddWordList` is the list of odd words found in the entity.

A.19 Switch Statement Smells

Switch statements in the code usually indicate procedural behavior. In Smalltalk there are no switch messages, but there are still conditionals.

Using the auxiliary `methodWithConditional`, we detect *Conditionals send to Parameters* and *Nil Checking*.

The predicate `methodWithNilCheck` checks that there is a conditional in the method, and the receiver is a nil checking send. This smell indicates that there is an object receiving a number of nil checking in a class, and different code is performed according to the result. Fowler proposes that this kind of object could be refactored and replaced by a *null object*.

For example, in the classes `List`, `CharacterArray`, `TableAdaptor` and `DependentList` the variable `dependents` receives eight, twelve and eight checks of wether the variable is `nil` or not, in the context of a conditional call. The threshold value of the amount of nil checks over an object is defined by the user.

On the other hand, the predicate `methodWithParameterConditional` retrieves the parameters of a given method, and for each one of them, checks whether in the method there is a parameter that is compared to a literal, and depending on the result, different code is executed. According to Fowler, this behaviour can be replaced with polymorphism.

For example we found the following code in the class `GeneralNameSpace`, in selector `mapClassType:`.

```
mapClassType: typeName

    typeName == #none
        ifTrue: [^#beFixed].
    typeName == #objects
        ifTrue: [^#beVariable].
    typeName == #bytes
        ifTrue: [^#beBytes].
    typeName == #immediate
        ifTrue: [^#beImmediate].
    typeName == #fixedSize
        ifTrue: [^#beFixed].
    typeName == #variable
        ifTrue: [^#beVariable].
```

Note that we do not take into account nil checking over the parameter as this is a smell detected by the previous predicate.

The generic predicates for detecting *Switch Statements* are the following:

```
badSmell(<switchStatements,nilCheck>, method(?method),
<?class,?selector,?nilCheckList>, <?numberNilChecks>)

badSmell(<switchStatements,parameterConditional>,
method(?method), <?class,?selector,?parameterCheckList>,
<?numberParameterChecks>)
```