Language Independent Metric Support towards Refactoring Inference

Yania Crespo[1], Carlos López[2], Esperanza Manso[1], and Raúl Marticorena[2]

[1] University of Valladolid
Department of Computer Science, Valladolid (Spain)
{yania, manso}@infor.uva.es
[2] University of Burgos
Area of Languages and Informatic Systems, Burgos (Spain)
{clopezno, rmartico}@ubu.es

Abstract. One of the current trends in refactoring is *when* and *where* we should refactor. Until now, most of the proposals establish that the refactoring process starts from the programmer intuition and experience. From the bad smell concept, and using metrics, it is possible to discover refactoring opportunities, not only from a subjective point of view but also from an objective point of view.
The following work presents an exploratory case study on the use of metrics in the detection of bad smells.This leads to related refactorings in order to improve underlying design. The process is achieved in a language independent manner. In this sense, it is briefly described a framework support for collecting metrics that allows to reuse the effort on a wide family of object-oriented languages. Framework solution is based on the use of metamodels describing family of languages. In addition to this, it is also described how to use the approach and its support, with other metamodels.

Key Words: metrics, bad smells, refactoring, automation of metrics collection, metric-based reengineering, frameworks.

1 Initial Context

One of the key subject in code refactoring process is: *when* and *where* do we perform refactorings? In [6] Fowler proposes a list of clues or symptoms that suggest refactoring opportunities. These symptoms or stinks are named "Bad Smells" and their detection should be achieved from *"the programmer intuition and experience"*.

Currently, there are a big number of integrated development environments (Eclipse, NetBeans, Visual Studio .NET, Refactoring Browser, etc.) which include refactoring capabilities. These environments also contain or allow to add plug-ins for obtaining metrics. The programmer is also able to customize the warning messages and corrections for every metric over the threshold.

However, there are common points between these concepts not connected until now. Although we have metrics, they are not used to determine refactorings.

There is not a direct connection among these metrics, the usual flaws that could be suggested by them, and the required corrective actions to reduce or erase these flaws. The usual flaws can be described in terms of bad smells and the corrective actions, in terms of the refactoring operations suggested by each bad smell. Thus, we can go from metrics to bad smells and from these, to refactoring.

On the other hand, metrics should be implemented for each object-oriented environment/language that we use. Nevertheless, one of the intrinsic properties of most of them, specially in object-oriented metrics, is their language independence.

One of the current trends in refactoring research is to achieve language independence. Therefore, starting from the current state of the question, we can go forward in two directions:

– Use metrics as clues of bad smells, to hint or suggest the suitable refactorings.
– Define a language independent metric collection support. The main issue when defining this support should be to fit solution for reuse in most of integrated development environments or in a multi-language environment.

The remainder of this work continues as follows: Section 2 presents the state of the art and current problems, from metric and refactoring studies to the connection of both lines. In section 3, we describe a case study about the relations between metrics and symptoms, suggesting code flaws. In Section 4, it is introduced tool support based on frameworks to use metrics in a language independent refactoring process. Finally, in Section 5, we finish with conclusions and future works.

2 State of the Art and Current Problems

The starting point to this matter is the Fowler's classical book on refactoring [6]. There are defined 22 bad smells from a non formal manner, with a simple description and a set of suggested refactorings to improve the code.

On the basis of this work, in [15] the author collected the opinion of programmers about the presence of bad smells in a set of program codes. He compared these results with metric results on the same codes. He also proposed a taxonomy for bad smells: **Bloaters**, **Object-Oriented Abusers**, **Dispensables**, **Encapsulators**, **Couplers** and **Others**. For each bad smell, he suggested a set of related metrics. This assignment is partially subjective.

However this work just confronts the collected opinions with correlations between metrics. In particular, the author only takes three bad smells in **Bloaters** category: *Large Class*, *Long Parameter List* and *Duplicated Code*. For the two former cases, we can use classic metrics related to the code size [14], but in the case of *Duplicated Code*, it is necessary to use a duplicated code detection tool. There are not metrics which collect this flaw. The author used the number of duplicated lines which not seems to be the most accurate method to detect clones in code. Duplicate code detection, although necessary sometimes, goes beyond the scope of this paper.

His conclusions hint that there are not correlations between smell detection based on human intuition (subjective) and metrics (objective). The surveys confirm that refactoring symptom detection without objective method leads to different results depending on the programmer (according to their experiences, their involvement in the project, etc.)

Relations between metrics and refactoring has been studied from other point of views. In [3], change metrics are used among different versions of a system to hint which refactorings have been applied at each evolution step. The work is based on the definition of some heuristics in this sense. Other similar works are [8] and [20].

By other side, in [4, 11], it is presented the idea of the language independence collecting object-oriented metrics. To collect metrics, they relate available information in a metamodel for a family of languages. The kind of languages the metamodel describes, have led them to do not have information on complex characteristics of inheritance or generic classes. On the other hand, although they name the concept of independent metric engine, they do not explain its design and the relations between metrics, bad smells and refactorings are not defined either.

In [19], Tourwé and Mens propose the detection of refactoring opportunities using queries on a logic meta-programming environment. They define queries to suggest the corrective actions to do. This solution, in its current state, is not directly reusable due to the use of the base language as repository. This means that their "Prolog" is reasoning about the real Smalltalk code base, and not about an imported logic representation of the code.

From these previous works, this paper describes a case study on the relation metrics-bad_smell-refactoring and presents the definition of a framework to calculate metrics in order to support the detection of refactoring opportunities in a language independent manner. It is presented an object-oriented design of the framework and its instantiation to use our own metamodel describing a family of languages which takes into account complex inheritance and genericity issues.

3 Bad Smell and Metric Relations

From the work [15], where relation between metrics and bad smells is not closed, we define a case study to show the usefulness of the detection of bad smells based on metrics.

Refactoring a big scale project, over which we have source codes but not experience about the project domain, brings us a problem: *where* do we begin to refactor? We choose to define a case study with an open source project. We do not know anything about this project however using metrics we are able to propose refactorings.

We select the open source project JFreeChart (www.jfree.org). It is a Java library for graphics drawing. The metrics are calculated on the last version (`1.0.0-pre2`). The study is constrained to bad smells that can be found with widely accepted metrics and language independence.

We take four bad smells: *Data Class* and *Lazy Class* in the **Dispensable** category with a strong correlation [15], *Switch Statements* in the **Object Oriented Abusers** category and finally, *Parallel Inheritance Hierarchy* in the **Change Preventers** category . The goal is to determine which classes in this library present these symptoms.

The case study focuses on class and method metrics. We have selected size metrics as NOA, NOM, etc. [14] and other object-oriented metrics as [2]: WMC, NOC, DIT, LCOM, CBO and RFC. Also we use method metrics as V(G) [16], LOC (Lines Of Code) and NBD (Nested Block Depth) [14].

3.1 Bad Smell: Data Class

"These are classes that have fields, getting and setting methods for fields, and nothing else" [6]. Thus, we select the metrics Number Of Attributes (NOA) and Methods (NOM). Complexity and cohesion of the selected classes should be low.

Using this filter and taking the five classes with higher values using clusters[3], we are able to detect: `AbstractRenderer`, `ChartPanel`, `PiePlot`, `XYPlot` and `CategoryPlot`. If we observe their codes, all of them are classes with a big number of accessor (`get`) and mutator (`set`) methods.

Refactoring to be applied [6]: **Move Method** to add more functionality to these classes.

3.2 Bad Smell: Lazy Class

"A class that isn't doing enough to pay for itself should be eliminated" [6]. These classes have a low number of attributes (NOA) and methods (NOM). Their complexities are low. Their DIT values are also low, so they do not add any functionality directly or indirectly by inheritance. Cohesion among methods is usually low.

Using this filter we obtain:

– if we set a DIT value of 1: we find classes merely functional, without state, which do not accomplish any tasks (i.e. `CountourPlotUtilities`, `DataSetReader`). Some of the selected classes by this filter also implement the **Factory Method** design pattern [7] (i.e. `ChartFactory`).
– DIT higher values are in classes with low functionality. The class names begin with `Default` which it suggests a default behavior (i.e. `DefaultKeyedValues2DDataSet`, `DefaultKeyedValuesDataSet`)

All these classes, although have been discovered with different filters, are grouped into the same set (cluster). Their role in the system is to provide low functionality so they should be refactored.

Refactorings to be applied [6]: **Move Method**, **Remove Class**, **Collapse Hierarchy** and **Inline Class** to increase or decrease the class complexity.

[3] Using Weka tool, available at http://www.cs.waikato.ac.nz/ml/weka/

3.3 Bad Smell: Switch Statements

"Most times you see a switch statement you should consider polymorphism" [6]. Applying the McCabe Ciclomatic Complexity metric (V(G)) [16], we find the `executeQuery` method in `JDBCXYDataset` class with a value of V(G) = 54. Usually, this value should not be greater than 10. The other two values over threshold are LOC = 153 and NBD = 7. Inspecting the source code of this method, we observe three `switch` control statements, with 12, 3 and 13 `case` clauses. The remainder of the methods in these classes maintain the metrics in the recommended thresholds.

Refactorings to be applied: **Replace Conditionals with Polymorphism** and **Replace Type Code with Subclass** or **Replace Type Code with State/Strategy**. Besides, we should apply **Extract Method** refactoring to reduce the complexity of long methods and high density of statements.

3.4 Bad Smell: Parallel Inheritance Hierarchy

"Every time you make a subclass of one class, you also have to make a subclass of another" [6]. We establish the use of the metrics (DIT and NOC) to detect this bad smell. Depending on the depth of inheritance tree and the number of children, we use these values as indicators of parallel inheritance hierarchies existence. More concretely, we choose classes with a number of children greater than 1, so the inheritance hierarchies are obviously complex. Collecting the metric values and clustering, we found four clusters (see Table 1). Studying the different mean values and standard deviations for each cluster, we only focus on classes taking into account the mean values of DIT and NOC. We are looking for classes at the top of the inheritance hierarchy (DIT between 1 to 3) with a medium number of children (NOC greater than 4 in this case).

Table 1. `JFreeChart-1.0.0_pre2` - Clusters

Cluster	Num.Classes	%	Mean DIT	St.Dev	Mean NOC	St.Dev
0	410	65%	2.8592	0.5164	0	1.4839
1	64	10%	5.1989	0.7940	0.1642	0.3704
2	128	20%	1.0478	0.2198	0.0921	0.3133
3	**27**	**4%**	**1.9991**	**0.9162**	**4.0688**	**3.4295**

The rest of the clusters contain classes with high depth and without children (Cluster 0), very deep with few children (Cluster 1) or low depth with few children (Cluster 2). These three last clusters do not seem suitable in order to find parallel hierarchies. Therefore, we take Cluster 3 with its 27 classes. To find parallel inheritance hierarchies we establish that classes must have values of DIT and NOC very similar. Also we added the criteria that class names must have similar prefixes as [6] suggests. By means of this process, we have detected three parallel hierarchies. We show the root classes and their metric values:

- Hierarchy 1
  - `Tick` (DIT=1, NOC=2)
  - `TickUnit` (DIT=1, NOC=2)
- Hierarchy 2
  - `AbstractCategoryItemLabelGenerator` (DIT=1, NOC=4)
  - `AbstractPieItemLabelGenerator` (DIT=1, NOC=2)
  - `AbstractXYItemLabelGenerator` (DIT=1, NOC=2)
- Hierarchy 3
  - `RenderederState` (DIT=1, NOC=3)
  - `Plot` (DIT=1, NOC=12)

Hierarchy 1 does not need any explanation about the metric values. In **Hierarchy 2**, the NOC value includes two inner classes that must not be considered to find the bad smell. In **Hierarchy 3**, similarity has been obtained by similar prefixes. Besides, the other nine child classes of `Plot` have not descendants, the other three classes have an association one to one with descendants of the `RendererState` class.

Through this process, we point out a set of parallel inheritance hierarchies that follow a similar pattern. They must be observed manually by the programmer to decide the suitable refactoring set to apply.

Refactorings to be applied: **Move Method** and **Move Field**.

4 Metric Calculation Support Based On Frameworks

In this section we present a language independent solution to the metric calculation support in the aim of obtaining an assisted refactoring process. Current solutions to metric calculation are proposed to work on particular languages.

Although they are correct solutions, there is an outstanding issue to be taken into account. Most of metrics, specially object-oriented metrics, are language independent. Even if it seems to be worth the trouble, in practice, it is not taken advantage of this opportunity. The same definition and implementation effort is achieved from the scratch to obtain metrics, for each development environment and programming language.

The solution to this problem is based on a metamodel. This metamodel must collect the basic elements of any object-oriented language: classes, attributes, methods, client-provider relations between classes, inheritance and genericity. In particular, it would be necessary to include information about flow-control instructions, assignment instructions, expressions, etc. All these instructions are needed to calculate metrics as V(G) [16], WMC [2], etc.

The UML metamodel [17] does not contain information about instructions. In our case, the used metamodel contains this information. Taking this metamodel, defined in previous works [12, 13], or other similar metamodels like FAMIX [4], the proposed solution defines a framework to run metrics with language independence. A framework is a set of abstract and concrete classes which defines an easily extensible behavior [5].

The main goal is to propose a framework that can be reused and extended with classic and new metrics on a wide set of metamodels to infer refactorings. Although in particular, we have validated this framework on a defined metamodel, there is an open line of work about incorporating it to other metamodels with small adaptations, following the advice given in next section. The framework design is proposed as an open guide to be implemented on any object-oriented language. We have implemented it on Java.

4.1 Metamodel Elements Traversal

To avoid modifications on every class representing metamodel instances which contain information to be collected when measuring, we apply the **Visitor** [7] design pattern. The aim of this design pattern is to avoid including a new method each time we need to make a new operation with all of them. In this particular case, the necessity emerges from measuring different element properties. This is also important in a metamodel solution based in order to preserve the metamodel definition.

The pattern indicates that `accept` methods must be introduced in each element to visit. In a metamodel with unique hierarchy, this is reduced to introduce an `accept` method in the root of hierarchy. By other side, we define a `Visitor` interface which must include `visit` methods for each one of the measurable elements (see Fig. 1).

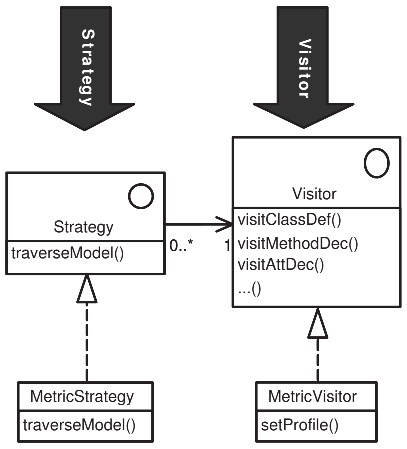

Fig. 1. Metric Engine Core

The traversal algorithm is defined independently of the visitor, allowing the use of the **Strategy** [7] design pattern. We choose dynamically the concrete algorithm to access to the metamodel instances.

4.2 Runnable Metric Hierarchy

Metrics have been classified in several ways. In [10] we find different taxonomies more or less complex. Particularly, we only focus on the granularity level: system, class and method. Metrics related on attributes are linked to classes as containers. For example, metric NOA (Number Of Attributes) [4] [14] measures the number of encapsulated attributes in the class context.

Depending on the information the metamodel describes, some metrics could have problems to be defined. Metamodels, as abstractions in general, lose some information from the elements they describe. In our metamodel, the loss is reduced to branching instructions (conditional and loop sentences). They are stored without semantic content.

This does not allow to define metrics related with McCabe ciclomatic complexity [16] in a language independent way. Nevertheless, the framework still supports their calculations. We calculate them using key words stored in concrete extensions (with language dependence).

The inheritance hierarchy of metrics is presented in Fig. 2, where `Metric` abstract class is playing the **Template Method** [7] role. Before running it, it is checked (`check` method) to verify that the metric is related with the type element to measure. If it is possible, the metric is calculated through `run` method. While checks are defined in the framework core, concrete executions are defined in the framework extension, following the **Command** [7] design pattern. Both methods, `check` and `run` build the template of `calculate` method.

To collect the measures, we use the **Collecting Parameters** design pattern defined in [1]. A `MetricResult` (see Fig. 3) implements the pattern. The object collects the measures each time the `calculate` method is called on a object which implements the `IMetric` interface. This solution is similar to a blackboard where everyone writes their results.

4.3 Profiles: Metric Customization

Metrics suggest certain problems for their application and interpretation. We observe that depending on the context applied, thresholds can change. Therefore, the framework must support the customization of these values.

Initially, metrics are instantiated with recommended default values. These values are subjective, and should be fitted by means of empiric observations, tuning and adjusting the values.

With this aim, we define a wrapper class `MetricConfiguration` (see Fig. 3), that allows to change the initial metric definition, rewriting the default values and adjusting the metric threshold to a particular context or domain.

[4] Also known as NOF (Number of Fields)

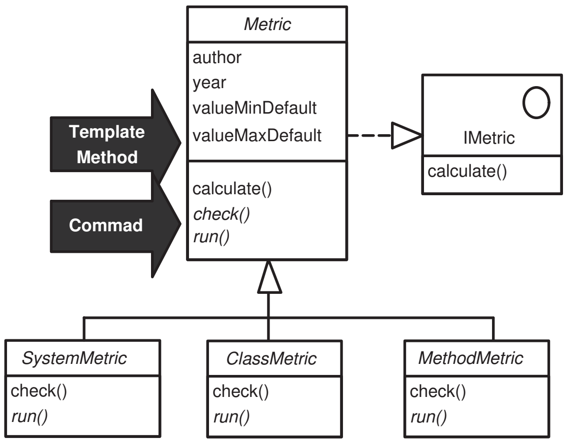

Fig. 2. Metric Framework Core

By means of a configuration profile in `MetricProfile` class (see Fig. 3), programmers can define different profiles. They can tune the values, on the base of previous observations or depending on the domain. We do not focus, for the moment, on aspects related to profile persistence and recovery.

4.4 Measure Calculation

Bringing together all the pieces, the concrete process begins visiting the elements, and obtaining the metrics, following a concrete strategy. It uses a visit method for each of the metamodel elements to be visited. A metric profile is linked to the visitor (`setProfile` method in Fig. 1).

On each element, we apply metric pre-configurated in the current profile. Results for each metamodel object are collected as a measure that allows navigability to the metric using `MetricConfiguration` class, on one side, and to the object where the metric has been calculated, on the other side. Measures are grouped in a "metric result" (`MetricResult` class) to allow the result analysis and later presentation.

4.5 Framework Validation: An Example

The framework has been implemented on an existent metamodel. This metamodel supports concepts as class and inheritance. We have implemented some metrics as DIT (Depth Inheritance Tree), NOC (Number Of Children) [9] among others.

Both metrics are defined as class metrics, so we can define them as extensions of `ClassMetric` class (see Fig. 4). Body of `run` method is redefined using

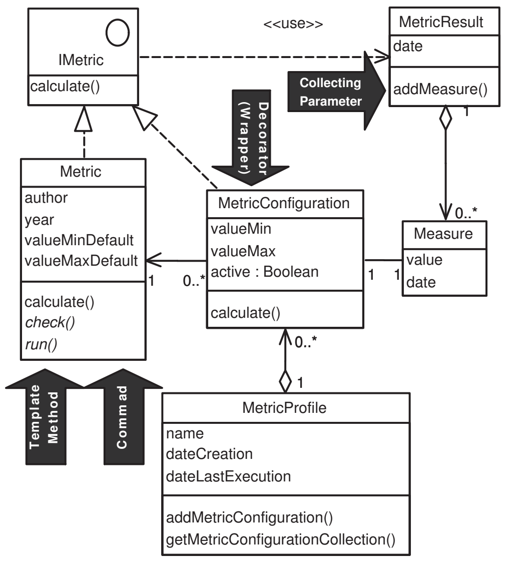

Fig. 3. Customization using Profiles

the information extracted from the metamodel and represented as instances of `Measure`. The entry point to calculate the metric value is in this case a class of the analyzed code. From this class, we navigate through its inheritance relations to achieve its depth and number of children.

To run on a particular example, all the effort falls on the framework. To achieve the metric collection, programmer must simply include these metrics in a profile. The implementation of the metamodel and the metric framework has been performed on Java, but the open design allows to be implemented on any other object-oriented language.

4.6 Strengths and Weaknesses

The main benefit of this approach is the reuse of the framework. We have a tool to obtain metrics for a wide set of objet-oriented languages such as implemented parsers from language to metamodel. In our work, we have implemented a metamodel (MOON [12]) and have also implemented parsers for Java and Eiffel.

The future improvements of the framework are:

– include the **Observer** [7] to update and recalculate metrics only associated to modified elements.

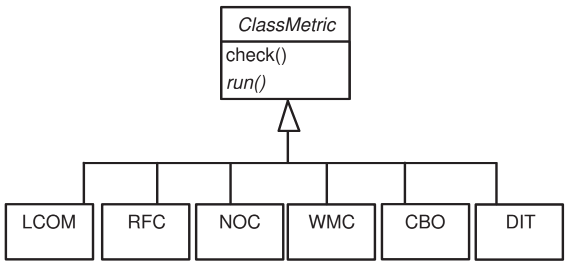

Fig. 4. Example of Framework Extension

- include additional filters to appoint that certain metrics must be customized. By example, in those languages that define constructors, NOP (Number Of Parameter) [18] metric could be relaxed in its maximum values.
- add a graphic representation tier to help to the interpretation of the metrics.

5 Conclusions and Future Works

Assisted calculation of metrics avoids a manual inspection of code, locating with a low effort those classes to be refactored.

The main goal is to provide a support to metric calculation in order to infer refactorings, with a certain language independence. We want to reuse the relation between bad smells and metrics with the refactorings to be applied. The final aim is to integrate the metamodel and framework in several environments, or multi-language environments.

There are still many open lines, but we claim benefits from a reuse approach in the metric and refactoring support.

Other lines of work are:

- Continue with the empirical validation of the metric framework to detect bad smells and to infer refactorings.
- Face the implementation problems of metrics which need particular features of each language.

References

1. Kent Beck. *Smalltalk: best practice patterns.* Prentice-Hall, Inc., 1997.
2. Shyam R. Chimdaber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions On Software Engineering*, 20:476–493, 1994.
3. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'2000*, pages 166–177. ACM Press, 2000.

4. Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, Institute of Computer Science and Applied Mathematic. University of Bern, 1999.
5. Mohamed E. Fayad, Douglas C. Schmidt, and Ralph Johnson. *Building Applications Frameworks. Object-Oriented Foundations of Framework Design.* Wiley, 1999.
6. Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison Wesley, 2000.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.
8. Tudor Gîrba, Stéphane Ducasse, Radu Marinescu, and Daniel Ratiu. Identifying entities that change together. In *Ninth IEEE Workshop on Empirical Studies of Software Maintenance*, 2004.
9. Martin Hitz and Behzad Montazeri. Chidamber and kemerer's metrics suite: A measurement theory perspective. *Software Engineering*, 22(4):267–271, 1996.
10. David A. Lamb and Joe R. Abounader. Data model for object-oriented design metrics. Technical report, Deparment of Computing and Information Science. Queens's University., 1997.
11. Michele Lanza and Stéphane Ducasse. Beyond language independent object-oriented metrics: Model independent metrics. In *QAOOSE 2002*, pages 77–84, 2002.
12. Carlos López and Yania Crespo. Definición de un soporte estructural para abordar el problema de la indepedencia del lenguaje en la definición de refactorizaciones. Technical Report DI-2003-03, Departamento de Informática. Universidad de Valladolid, septiembre 2003. Available at http://giro.infor.uva.es/docpub/lopeznozal-tr2003-03.pdf.
13. Carlos López, Raúl Marticorena, and Yania Crespo. Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje. In Jaime Gómez Ernesto Pimentel, Nieves R. Brisaboa, editor, *Actas JISBD'03, Alicante, Spain ISBN: 84-688-3836-5*, pages 251–262, November 2003.
14. Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
15. Mika Mäntylä. Developing new approaches for software design quality improvement based on subjective evaluations. In *ICSE*, pages 48–50. IEEE Computer Society, 2004.
16. Tomas McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
17. OMG. Unified modeling language: Superstructure version 2.0. http://www.uml.org, 2004.
18. Meilir Page-Jones. *The practical guide to structured systems design: 2nd edition.* Yourdon Press, Upper Saddle River, NJ, USA, 1988.
19. Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proc. 7th European Conf. on Software Maintenance and Reengineering*, pages 91 – 100, Benvento, Italy, 2003. IEEE Computer Society.
20. Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *16th International Conference on Software Engineering and Knowledge Engineering*, pages 123–128. Banff, Alberta, Canada, June 20-24, 2004.