Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 117–136
www.elsevier.com/locate/entcs

Adaptive Detection of Design Flaws

Jochen Kreimer[1,2]

Research Group Programming Languages and Compilers
Department of Computer Science
University of Paderborn, Germany

Abstract

Criteria for software quality measurement depend on the application area. In large software systems criteria like maintainability, comprehensibility and extensibility play an important role.
My aim is to identify design flaws in software systems automatically and thus to avoid "bad" — incomprehensible, hardly expandable and changeable — program structures.
Depending on the perception and experience of the searching engineer, design flaws are interpreted in a different way. I propose to combine known methods for finding design flaws on the basis of metrics with machine learning mechanisms, such that design flaw detection is adaptable to different views.
This paper presents the underlying method, describes an analysis tool for Java programs and shows results of an initial case study.

Keywords: Design flaw, code smell, object-oriented design, software quality, refactoring, program analysis, and machine learning.

1 Introduction

The object-oriented programming paradigm promises clearly structured, reusable and easily maintainable software. In practice, only very experienced developers achieve this.

"All data should be hidden within its class" [36] is but one of the numerous helpful pieces of advice of known mentors and successful practitioners of the

[1] Email: jotte@uni-paderborn.de
[2] Thank to M. Thies, P. Pfahler, U. Kastens, and the anonymous reviewers for their valuable comments.

doi:10.1016/j.entcs.2005.02.059

object-oriented paradigm that should lead to a critical review of one's own program structures.

One valuable technique to improve software quality is manual software inspection [10] [9]. It incorporates sifting source code, design and documentation. This plays an important role for quality assurance in modern agile development processes like *Extreme Programming*.

With inspection techniques, errors might be found before testing which means that they are found in early development stages. Tool support is recommended for this time consuming task. Using a tool that analyzes software automatically and repeatedly should achieve a constant high level of quality.

It is my aim to find errors in the design of software systems automatically and therefore to avoid program structures that can not easily be extended and changed.

Section 2 describes and classifies the notion of design flaws. Afterwards section 3 introduces a method for adaptive detection of design flaws. Section 4 describes the prototype tool *It's Your Code* (IYC) which implements that method. The applied program analysis techniques are outlined in section 5. The remaining sections describe results of an initial case study and provide an overview of related work, as well as a conclusion and plans for future work.

2 Design Flaws

Design flaws are program properties that point out a potentially erroneous design of a software system.

In the literature these are also referred to as *"Design Heuristics"* [36], *"Design Characteristics"* [46] or *"Bad Smells"* [13]. The authors denote design flaws normally using metaphors and explain to software developers and architects how to recognize and correct such erroneous software structures.

In addition Fowler [13] describes *refactoring* transformations which change the internal program structure, but do not alter the observable behavior of the program. These transformations lead to revised and simplified programs. Fowler's *"Bad Smells"* are design flaws that describe which program locations may get improved by *refactoring* transformations. Examples of *"Bad Smells"* are long methods, multipurpose classes, too many parameters or local variables in a method, violation of data encapsulation, overuse of delegation or misuse of inheritance.

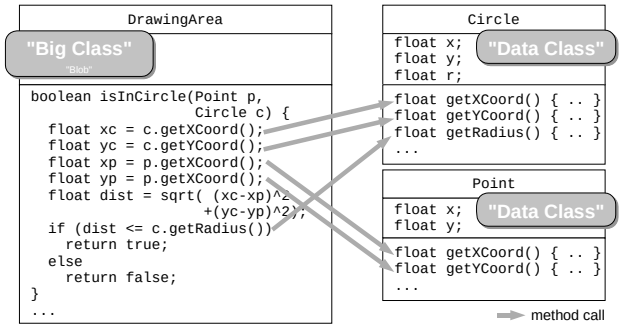

Fig. 1. Example of design flaws "Big Class" and "Data Class".

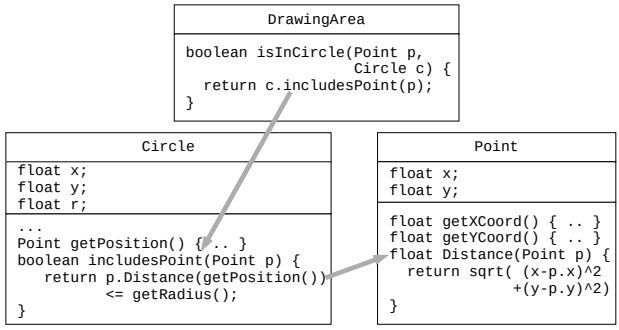

Fig. 2. Example from fig. 1 after *refactoring* transformations.

2.1 Example: "Big Class"

Figure 1 shows an excerpt from a class diagram. One could imagine that the classes *DrawingArea*, *Circle* and *Point* form a drawing program. The class *Circle* implements, among other things, the method *isInCircle* which checks whether a given point is inside a given circle.

It is remarkable that *isInCircle* does not even use a single field of its own class. Only values from other classes are used to compute the result.

Classes *Circle* and *Point* own some data fields with associated *get*-methods. These classes are typical examples for "Data Classes", because they do not implement any behavior on their own. They exhibit high cohesion between fields and methods as well as low coupling to other classes.

The class *DrawingArea* may be called a "Big Class" (also known as "Blob"). It violates the basic principle of data encapsulation, because data is situated in the data classes and all behavior is implemented in the class itself. One could imagine that this class implements several more similar methods, such that this class seems to be responsible for more than one task. A relatively large number of statements in every method, low cohesion and high coupling to other classes are criteria that point towards a "Big Class".

The result of restructuring these classes is shown in figure 2. Cohesion has

Design flaw *"Long Method"*:

"The longest living program is that with the shortest methods, . . . "

"the real key . . . are good names."

". . . write a new method, whenever you want to comment something."

"Conditions and loops give hints for extractions."

$\Longrightarrow$ *Refactoring suggestion:* extract method.

Design flaw *"Big Class"*:

"Class tries to do too much, . . . "

". . . too many instance variables, . . . "

". . . duplicated code . . . "

"Somtimes a class does not use all its instance variables . . . "

$\Longrightarrow$ *Refactoring suggestion:* extract class, move method.

Fig. 3. Excerpts from Fowler's descriptions of design flaws [13].

increased and coupling has decreased. Classes mostly operate on their own data and hide details from the outside world.

2.2 Classification

Well-known program inspection tools (e. g. the Lint tool [17]) are searching for programming errors. They typically can detect uninitialized variables, violation of array bounds, dereferencing of null pointers or division by zero and similar frequent programming errors by means of static program analysis.

The concept of *Design Patterns* [14] is complemented by *Anti Patterns* [4]. *Design Patterns* propose schemes for solving frequent problems; whereas *Anti Patterns* represent schemes for solutions that cause more problems than they solve. The authors distinguish levels of software development, architecture and management. Problems, symptoms and consequences are exemplified.

But when searching for design flaws, it is interesting to identify program structures that hint at an erroneous design. This kind of design flaws resides somewhere between *Anti Patterns* and programming errors.

2.3 Depiction of design flaws

Design flaws are described informally. Concise and metaphoric names ease acquisition. Examples of conspicuous program fragments illustrate each problem. In many cases the authors argue with indications and hints on program properties. Some authors provide rules of thumb to ease identification. Figure 3 shows excerpts from Fowler's description.

The established descriptions of design flaws aim at intuitive comprehension. They communicate the knowledge of "good" object-oriented design in manageable chunks. Nevertheless, to recognize design flaws remains difficult for an unexperienced developer or designer. It depends on the perception and

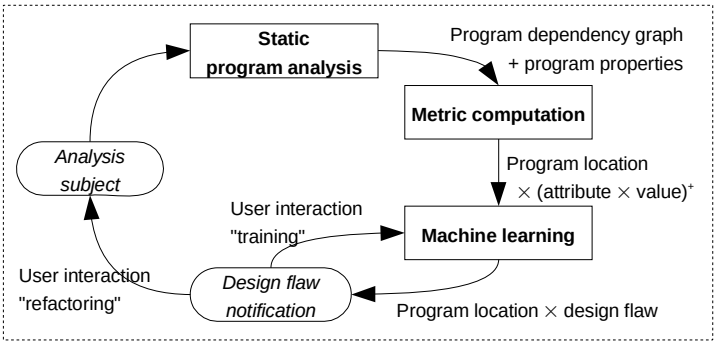

Fig. 4. Basic concept for adaptive detection of design flaws.

expert knowledge of the individual which flaws are recognized and how this is done. Different people often have very different opinions concerning the same flaw [24].

The application area of a program under examination is forming the view on a design flaw, too. One could imagine that e. g. a mathematical program contains implementations of complex algorithms. One would allow longer methods here than within a program that implements an event driven user interface and basically relies on delegation to library functions.

3 Adaptive detection of design flaws

In order to detect design flaws automatically, I combine object-oriented metrics with machine learning techniques.

Figure 4 shows the basic concept for automatic detection of design flaws.

I follow the approach in [25] and assign a set of program properties to every design flaw. Each program property is expressed by object-oriented metrics. Often size, complexity, coupling and cohesion metrics are used. This forms a model that provides a strategy for detecting every design flaw.

Metric values are computed by static program analysis with classic approaches like control and data flow analysis as well as abstract interpretation. The analysis results lead, among other things, to a program dependency graph, which works as an abstract model of the analysis subject and is used in computation of the metrics. [3]

According to the model, measured values of program locations are passed to a machine learning mechanism. The machine learning mechanism has been trained with examples of known design flaws and afterwards is able to detect design flaws based on specific values of the metrics.

[3] Read more in section 5.

3.1 Modeling design flaws

My approach starts from a mental model of design flaws. Descriptions of design flaws provide hints on how to recognize each flaw. Often blurry indications or vaguely described program structures hint at a flaw. In addition, suggestions on how to remove a design flaw, using *refactoring* transformations for example, are provided. Hence, it also seems worthwhile to consider program locations where such transformations are applicable.

This newly built mental model of the design flaw is then described as a set of metrics. Every criterion will be mapped to one or more measurable program properties. The result is a set of object-oriented metrics that characterize each design flaw. At this point there is only a vague idea, about what values are expected for each metric and in what combination they would indicate the presence of a flaw. Furthermore, the relevance of each metric to the design flaw is not known. Thus initially, this is a hypothetical model.

An unmanageable amount of metrics is mentioned in literature, [5] [27] [11] [3] [12] among others. I have selected a reasonable subset or created new, similar metrics for my purpose.

Probably all design flaws can be described by metrics. An exception are flaws that rely on information from version history (e. g. divergent changes); these are not treated here. I have modeled five design flaws so far: "Long Method", "Big Class", "Feature Envy", "Delegator", and "Lazy Class", from which I show two example models in the following.

3.1.1 Model for a "Long Method".

Fowler arguments that programs live long if they consist of very short methods. Fowler adds that a key aspect of good methods are well-chosen method names. One indication for splitting a method into smaller ones is the presence of comments. Whenever something should be commented, Fowler suggests to build a separate method of that program fragment.

A long method is not a method with many statements, but a method with a complex control structure. A complexity measure should clearly take this into account.

Fowler argues that many parameters and local variables hinder method extraction. He suggests additional refactoring transformations that ease later decomposition.

In summary, these criteria correspond to the following metrics:

- The size measure *"Number of statements of a method"*.
- The complexity measue *"Complexity of a method"*.

The classical complexity measure from McCabe [27] is used.

- The size measure *"Number of parameters of a method"*.
- The size measure *"Number of local variables of a method"*.

3.1.2 Model for a "Big Class".

Classes should suit just one single purpose. It is often tried to overburden a class with multiple tasks. Such classes emerge over time, when "unobtrusively" implementing small changes or enhancements. Such classes also arise, when procedural design is converted to an object-oriented design or experienced procedural developers implement a main class with associated helper classes [36, p. 32ff.].

Fowler argues that multipurpose classes often own too many instance variables. Typically methods use only a fraction of these variables. Thus, the class decomposes naturally into multiple constituent classes. Fowler further claims that a class containing a large amount of code is suspicious of containing duplicated code. Redundancies should be eliminated by extracting short reusable methods out of long methods. Finally, Fowler looks at the usage of a class. When different users employ disjoint sets of methods, decomposition of the class might be sensible.

These criteria correspond to the following metrics:

- The size measure: *"Number of instance variables of a class"*.
- The cohesion measure *"Number of internal connected components"*.
  Figure 5-I shows methods of a class and their usage of fields, as well as method calls within that class. Presumably it is possible to partition the class into four connected components: $\{f_1, f_2, f_3, m_1, m_2\}$, $\{m_3\}$, $\{m_4, m_m, f_4\}$ and $\{f_n\}$.
- The complexity measure *"Median of complexities of all methods of a class"*.
- The size measure *"Median of the number of statements of all methods of a class"*.
- The coupling measure *"Number of external connected components"*.
  Figure 5-II shows a class C, whose methods are called from different parts of the program. With regard to the level of granularity, two or three disjoint subsets of the set of methods arise. The subsets provide hints for class extraction.

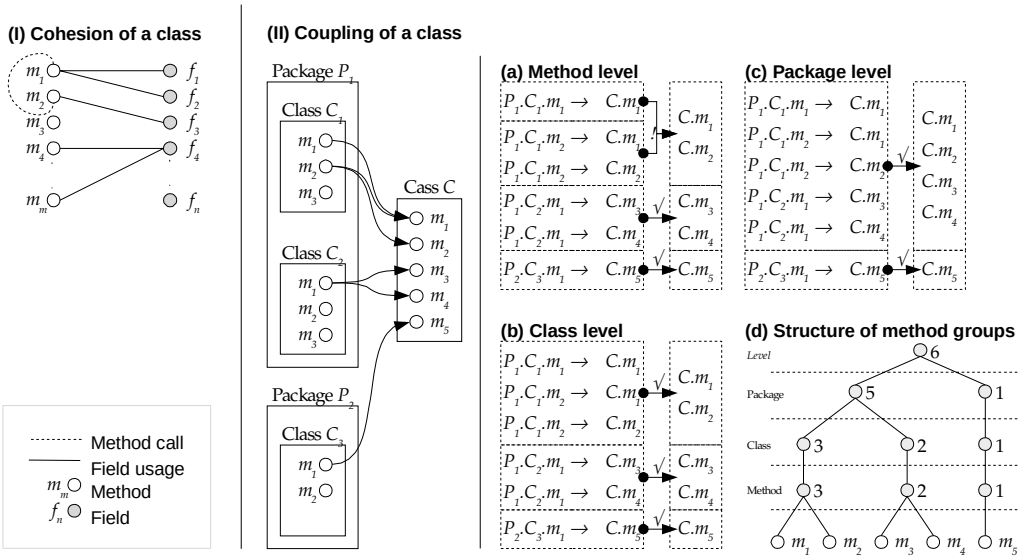

Fig. 5. Calculation of cohesion and coupling measures of a class.

3.2 Machine learning

The research area *machine learning* focuses on methodologies that allow programs to "learn". A program that learns fulfills a certain task and increases its performance by accumulating experience [28] [47] [16].

Machine learning techniques have been used successfully for speech recognition, driving vehicles, board gaming and further learning goals. Among others, the subjects statistics, psychology, philosophy and artificial intelligence contribute to machine learning.

Adopted techniques include concept learning, construction of decision trees, neural networks, bayesian learning, instance-based learning, genetic algorithms, and reinforcement learning.

When designing a learning method, it is first decided what task the learning program should fulfill. After that, it is necessary to model the experience that the program should draw from and the output it should provide. At last a performance measure is defined to check if the system improves over time.

For detection of design flaws, I have concentrated on learning decision trees.

Decision trees are used for classification. Their input consists of a set of criteria or attributes respectively. An attribute is a name/value pair, whose values may be nominal or numeric. The so-called target attribute describes the desired classification of a set of attributes with concrete values. A set of attributes with concrete values is called an instance.

A set of instances, including a value for the target attribute, works as a

training set. From this a decision tree is constructed recursively in a top-down fashion. According to a special entropy measure the information gain of each attribute is judged. The attribute with the highest information gain becomes the root of the next recursively constructed subtree. Refer to [28, chapter 3] for a detailed description. The *C4.5* method, which is used here, has been introduced by Quinlan [33][34].

An example of a set of instances, including a derived decision tree, is shown in figure 6. The decision tree should classify all training examples correctly, if possible, but this is not guaranteed. Furthermore not all attributes might be used. In this example all training examples are classified correctly without using the *"#methods"* attribute.

Instances not classified so far, namely instances without a value for the target attribute, get classified by interpreting the decision tree. Thereby inner nodes of the decision tree model the decision points based on a single attribute. Starting from the root node, a branch is selected that fits the actual value of the attribute. Leave nodes represent the result of the classification.

3.3 Adaptive detection

The detection of design flaws is a classification problem, too. It has to be decided whether a questionable program location really constitutes a design flaw.

Every design flaw gets modeled by a set of object-oriented metrics. These are used as attributes for the decision tree algorithm. The target attribute states, whether a design flaw exists. Figure 6 shows an example of a set of measured values for different classes of a program and the classification as a design flaw. The constructed decision tree for this design flaw will be used to check if an unknown program location could represent that design flaw.

For this example the number of training examples and the set of attributes used have been abridged. Only the number of fields, methods, and statements of every method in the class and just 12 training instances are shown. Learning mechanisms of this kind construct decision trees with sufficient accuracy from about 100 training examples.

It is remarkable that the number of fields of a class is the strongest criterion in this model. The number of methods is not used in the classification. These decisions, as made by the learning mechanism, allow first conclusions on the suitability of the chosen model for detecting "big classes".

If a proper training set is available, the constructed decision tree can be used for classifying future program locations. But often it is not evident, if the chosen attributes are suitable for characterizing the associated design flaw.

Nr.	*#fields*	*#methods*	*#statements*	Target attribute *Big Class*
1.	33	5	100	Yes
2.	28	29	87	Yes
3.	45	24	67	Yes
4.	21	27	95	No
5.	18	13	104	No
6.	13	5	83	No
7.	18	5	272	Yes
8.	5	2	301	Yes
9.	29	23	125	Yes
10.	67	23	125	Yes
11.	27	23	93	No
12.	32	8	113	Yes

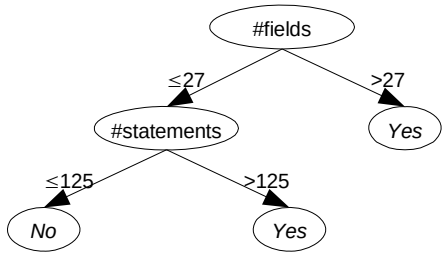

Fig. 6. Abridged example of measured values for design flaw "Big Class" and constructed decision tree [23].

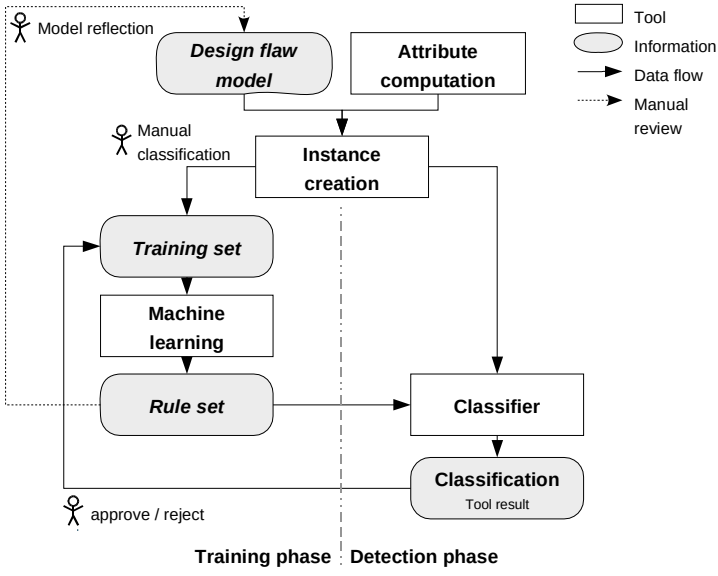

Fig. 7. Learning and detection phase with adaption of decision trees and design flaw models.

Furthermore the training set could have been chosen awkwardly in such a way that the intended spectrum of instances of that flaw is not fully covered yet. Therefore the mechanism should continue to learn from further examples.

Figure 7 shows the resulting approach that consists of two phases. In the

training phase, program locations are chosen and it is decided manually if the design flaw in question is actually present. The measured values of all these program locations form — together with the decision concerning the design flaw — the training set from which an initial decision tree is constructed.

In the detection phase the user states which parts of the system are to be analyzed. Here, all yet unknown program locations are measured and all decision trees of associated design flaws are applied to the measurements. Every flaw tainted program location is shown to the user.

Now the user verifies each individual case and approves the presumed design flaw or rejects the suggestion. In both cases the program location can be added to the training set as another example. If the user skips verification of a case completely, the training set remains unchanged.

Training and detection phase are independent from each other. Even during the detection phase, the user may add further suspicious program locations to the training set.

3.4 Explanation component

The user might find it difficult to discover why a program location has been detected as flaw tainted.

For explanation the decision tree can be shown. In combination with measured values of the relevant program location, the path through the decision tree can be visualized.

For a start the user compares his own idea of the nature of the design flaw with the decision tree. Possibly there are attributes the user rated as meaningful, but which do not appear in the decision tree at all or they appear only far away from the root of the tree. Attributes with a high impact on the decision would be located near the root by construction.

Quite often, an edge leads directly from a root node or some adjoining node to a leaf node. The associated attribute of the edge represents effectively a knock-out criterion. This clarifies the impact of the attribute for the decision to the user even more.

The user might notice that previous training examples do not cover the intended spectrum of the design flaw. Or the user's analysis might reveal that the model used is not appropriate.

3.5 Model reflection

Learning mechanisms, like the *C4.5* algorithm used here, achieve a very good adaptation if a sufficiently big training set is used. By means of standardized validation techniques the user gets an impression how confident the learning

mechanism is in its classifications.

When applying machine learning to design flaw detection, it is necessary to measure the performance in comparison to human intuition. That is, a well trained system should exhibit only few differences in comparison to a manual human analysis. If this is not achieved, the model of the design flaw probably does not match the user's intuition. The model must be refined.

For every training example the associated program location is known. Thus subsequent changes to the model can be made without loosing the training set. Provided that the program has not been changed, new attribute values for all program locations can be re-measured. In conjunction with the already known classifications, the original training set can be adapted to the refined model.

Future work will show if other learning techniques (like bayesian learning) achieve a better detection accuracy than decision tree techniques. But the main advantage of decision trees is that they can be inspected manually, such that the explanation component and model reflection are feasible at all. This would be lost with black box methods.

4 Tool for *Eclipse*

For evaluating the concept of section 3 the prototype tool *"It's Your Code"* (IYC) [4] has been developed as a plug-in for the Java development environment of the *Eclipse* platform [8].

Eclipse provides, like many other development environments, project, version and build management as well as source code editors and different views upon the current project. Views allow the developer to navigate the file system or to view the package and class structure in Java projects.

The *IYC* plug-in hooks into the user interface of such views. If a program object is selected, e. g. a package or class, the context menu provides options to start searching for design flaws. Results are presented as a list of potential design flaws within an own view.

Figure 8 shows a screen shot of *Eclipse* with a source code editor in the upper right area, a view for navigating the Java program structure in the left area and a list of presumed design flaws in the lower right area.

The configuration of IYC is omitted for brevity. Users may define their own models for new design flaws or change existing ones by assigning a set of

[4] The name has been chosen, because the tool just suggests potential design flaws. In the end the user has to decide whether a flaw exists or not.

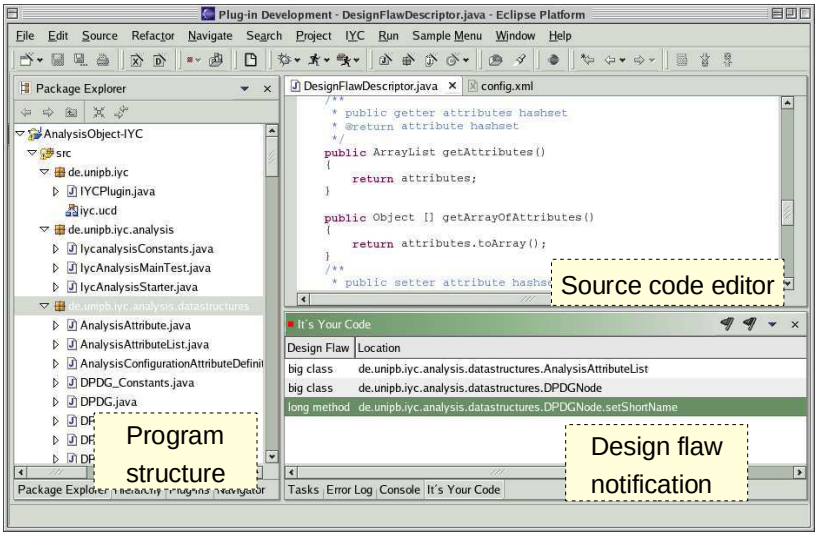

Fig. 8. Screen shot of IYC in *Eclipse*.

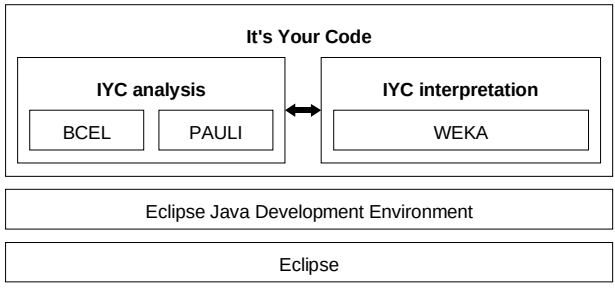

Fig. 9. Architecture of IYC.

predefined attributes. The current training state may be stored to disk and loaded separately.

Figure 9 shows the architecture of the tool. The *Eclipse* platform forms the basis by providing plug-ins for Java development. IYC consists of two parts: the analysis part *IYC analysis* computes object-oriented metrics, the interpretation part *IYC interpretation* implements the adaptive detection with user interaction.

The machine learning mechanism consists of the *J48* classifier — a specific *C4.5* implementation from the *WEKA* library [45].

The analysis part relies on the *Bytecode Engineering Library* (BCEL) [7] [2] for accessing the analyzed Java program. *PAULI* [41] is a library for program analysis of Java bytecode, which implements e. g. control and data flow analyses.

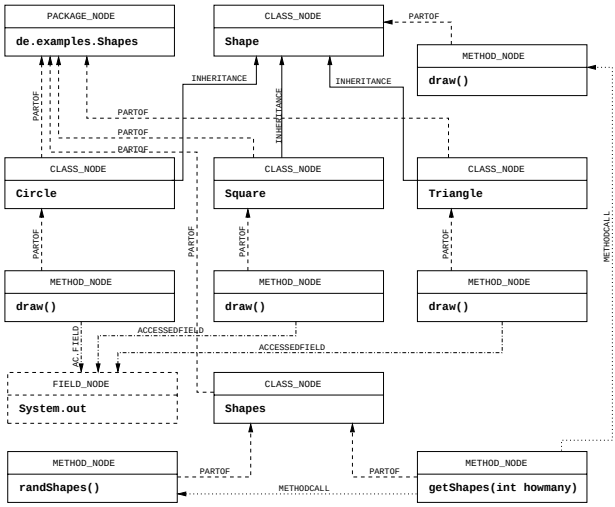

Fig. 10. Example of a *Design Program Dependence Graph* [21].

5 Rapid program analysis

For computing object-oriented metrics IYC uses a specific dependency graph, which is denoted *Design Program Dependence Graph* (DPDG). Node types of the graph correspond to types of the respective program objects: *"package"*, *"class"*, *"method"* and *"field"*. Edges represent relations between program objects: *"part of"*, *"inheritance"*, *"implements"*, *"association"*, *"accessed field"*, and *"method call"*. Figure 10 shows an example graph.

Building the program dependency graph requires significant amounts of memory and time. Therefore not the whole program, but only the context requested by the user is analyzed; e. g. a package or class. Within this context all program objects are analyzed and a node is created for each one. During analysis, relations to program objects may be discovered, that lie outside the context. These nodes are also added to the graph and are marked as external.

It is easy to derive metric values from the DPDG. Figure 11 shows some examples.

Many metrics are calculated by restricting the graph to certain kinds of relations and applying simple graph algorithms like depth-first and breadth-first search.

Often very similar metrics, which only differ in the chosen context, are computed. E. g. the number of fields used in a method with respect to fields defined within vs. outside its own class. This can be computed easily by widening the context along the *"part of"* relation.

Analysis speed is sufficient for an interactive tool; individual classes and packages are analyzed, depending on their size, in a few seconds. Memory

- *Number of fields of a class k* (size measure)
  Determine the cardinality of the set of all fields that are in *"part of"*-relationship to class k.
- *Number of internal connected components of a class k* (cohesion measure)
  (1.) Restrict the graph to nodes which are of type *method* or *field* and which are in *"part of"*-relationship to class k. (2.) Assume the relations *"method call"* and *"accessed field"* to be undirected and determine the number of connected components (q. v. figure 5-I).
- *Number of external connected components of a class k* (coupling measure)
  (1.) Determine the set of methods of class k using the *"part of"* relation. (2.) Restrict the graph to nodes that are in *"method call"*-relationship to this method set. (3.) Check if the method set from (1.) resolves into disjoint subsets regarding the methods from (2.) with same class membership (q. v. figure 5-II).

Fig. 11. Examples for computing metrics by means of a DPDG.

(a) Sizes of analysis subjects.

Measure	*IYC*	*WEKA*
#lines	4274	92615
#classes	91	597
#methods	765	7193
#fields	283	3431

(b) Number of wrong automatic classifications.

Design flaw	*IYC*	*WEKA*
"Long Method"	2 of 20	4 of 20
"Big Class"	1 of 20	4 of 20

Fig. 12. Sizes of analysis subjects and results of the initial case study.

requirements are moderate for a software development computer. Additional metrics may be easily implemented and therefore made available to the learning process.

6 Evaluation

The applicability of the proposed method has been proven in a first case study [23] using a prototype implementation.

The analysis subject consisted of two software systems that were well known to the test person, such that design flaws could be judged manually. These are the *IYC* system and the *WEKA* package. Figure 12-a gives sizes of both systems. Both implement a user interface and work with complex data structures. But *WEKA* implements more complex algorithms and *IYC* delegates many tasks to libraries.

Even though both design flaws "Long Method" and "Big Class" have been trained with just 20 examples the validation of the learning mechanism, using the *leave-one-out* method, led to an accuracy of 95 % and 100 % respectively.

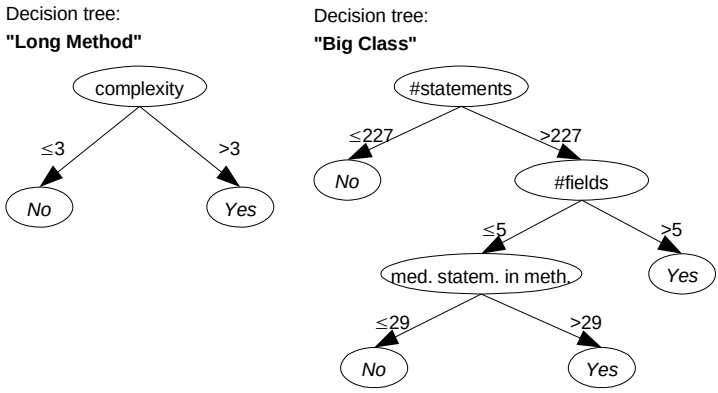

Fig. 13. Examples for decision trees.

Figure 13 shows the resulting decision trees. To detect a "Long Method" it was sufficient to use just McCabe's complexity measure. The learning mechanism decided to ignore further metrics. For detecting the "Big Class" flaw three out of five criteria were used.

The automatic analysis of both systems led to similar results. About 15 % of all methods have been classified as "Long Method" and about 20 % of all classes haven been marked as a "Big Class".

For comparison with the human intuition, 20 new program locations have been chosen randomly and evaluated manually. Figure 12-b shows that in each case only a few automatic classifications did not match the intuitive judgment of the test person.

7 Related work

As most related work has been discussed already, further related work is just summarized here.

Design flaw detection. This work relies on metrics to detect design flaws [25][26]. Adaption to requirements of individual users by means of machine learning techniques has been added.

Design flaws are heavily discussed in the *refactoring* community [35]. Among others, interpreting approaches for detecting design flaws have been proposed, e. g., in [42].

Visualizing program structures. Tools like *"jCosmo"* [43], *"CodeCrawler"* [22] and *"Crocodile/CrocoCosmos"* [39] visualize quantitative and structural properties of a subject program and provide different views for analyzing large programs.

It is my aim though, to relieve the user from interpreting and analyzing program properties and to automatically provide indications of presumed

design flaws instead. This is accomplished by a flexible interpretation mechanism that adapts itself.

Machine learning application. Machine learning techniques have further uses within software engineering [37]. They predict faults [19] [18] [29] or assess maintenance efforts [32] [6] [38].

Program abstraction models. In the *reengineering* domain [30] [20] models are used to abstract from concrete programs and programming languages respectively. Model instances reflect basic program properties which allow one to derive higher level properties, including object-oriented metrics. In the literature this model instantiation is termed as *fact extraction*. The well-known *Rigi* system [40] follows this approach. To my knowledge only complete programs are analyzed. This leads to hardly practicable memory and time consumption. Whereas my proposed rapid analysis examines only those parts of a program the user is really interested in.

Program dependency graphs, as employed here, have been introduced in [31]. They are most primarily used for *program slicing*. Approaches specific to Java appear in [15] [44].

8 Summary and prospects

I have introduced a method for detecting design flaws in object-oriented software. As design flaws are interpreted in very different ways, this method adapts itself to specific usage scenarios. I have combined well-known approaches, based on object-oriented metrics, with machine learning techniques and proposed an adaptive and learning approach.

In principle the methodology is applicable beyond object-oriented programs. The introduced tool is admittedly specialized for Java programs, but extractors for program properties and metrics of other languages could be integrated instead.

Based on a prototype implementation an initial case study has been conducted. To reach a final conclusion an empiric survey is needed. Therefore the prototype tool will be improved and will be made publicly available. An on-line component will allow collection of models and decision trees from a diverse user base for further design flaws. A user profile in the form of an inquiry about programming style and experience, size of development group and constitution, as well as characteristics of the software system built should give insight into factors that are reflected in different instances of a design flaw.

Based on this, a “learning group” in the form of a public catalog of design

flaws, consisting of models and decision trees, should emerge. [5]

Typically, when developing large software systems, programming guidelines are defined. The absence of design flaws could be one such guideline. When automatically detecting design flaws, this guideline could be verified and maintained. For this reason models of design flaws and decision trees are a suitable way to store and propagate knowledge.

References

[1] *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 2004.

[2] The Apache Jakarta Projekt, http://jakarta.apache.org/bcel. *Byte Code Engineering Library (BCEL)*, 2004.

[3] Dirk Beyer, Claus Lewerentz, and Frank Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. *Lecture Notes in Computer Science*, 2006:1–??, 2001.

[4] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley, 1998.

[5] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[6] Melis Dagpinar and Jens H. Jahnke. Predicting maintainability with object-oriented metrics - an empirical comparison. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *WCRE*, pages 155–164. IEEE Computer Society, 2003.

[7] Markus Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[8] Eclipse.org Consortium, http://www.eclipse.org. *Eclipse.org Main Page*, 2003.

[9] Micahel E. Fagan. Advances in software inspections. In *IEEE Trans. On Softw. Eng., 7 (12)*, pages 744–751, 1986.

[10] Michael E. Fagan. Design and code inspections and process control in the development of programs. Technical Report 00.2763, IBM, June 1976.

[11] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[12] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2 edition, 1996.

[13] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[14] Erich Gamma, Richard Helm, and Ralph Johnson und John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986. ISBN: 0–201–63361–2.

[15] Christian Hammer and Gregor Snelting. An improved slicer for java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22. ACM Press, 2004.

[5] Watch "http://ag-kastens.upb.de/iyc".

[16] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Stats. Springer, 2001.

[17] S. Johnson. Lint, a c program checker, 1978.

[18] Taghi M. Khoshgoftaar and Naeem Seliya. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Software Engineering*, 8(3):255–283, 2003.

[19] Taghi M. Khoshgoftaar and Naeem Seliya. Software quality classification modeling using the sprint decision tree algorithm. *International Journal on Artificial Intelligence Tools*, 12(3):207–225, 2003.

[20] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *10th Working Conference on Reverse Engineering*, page 36. IEEE, 2003.

[21] Carsten Lachmann. Statische Programmanalyse zur Erkennung von Entwurfsmängeln in Java-Anwendungen. Master's thesis, Universität Paderborn, 2004.

[22] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[23] Mike Liebrecht. Adaptive Erkennung von Entwurfsmängeln in Java-Anwendungen. Master's thesis, Universität Paderborn, 2004.

[24] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. Bad smells - humans as code critics. In *ICSM* [1], pages 399–408.

[25] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, 2002.

[26] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM* [1], pages 350–359.

[27] Thomas J. McCabe. A complexity measure. In *Proceedings: 2nd International Conference on Software Engineering*, page 407. IEEE Computer Society Press, 1976. Abstract only.

[28] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[29] Sanjay Mohapatra and B. Mohanty. Defect prevention through defect prediction: A case study at infosys. In *ICSM*, pages 260–272, 2001.

[30] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28. ACM Press, 1995.

[31] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, May 1984.

[32] Macario Polo, Mario Piattini, and Francisco Ruiz. Using code metrics to predict maintenance of legacy programs: A case study. In *ICSM*, pages 202–208, 2001.

[33] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[34] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[35] Refactoring Home Page, http://www.refactoring.com/.

[36] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[37] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[38] Jelber Sayyad-Shirabad, Timothy Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *ICSM*, pages 95–104. IEEE Computer Society, 2003.

[39] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics Based Refactoring. In *CSMR*, pages 30–38, 2001.

[40] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Muller. Rigi: A visualization environment for reverse engineering. In *International Conference on Software Engineering*, pages 606–607, 1997.

[41] Michael Thies. *Combining Static Analysis of Java Libraries with Dynamic Optimization*. Dissertation. Shaker Verlag, ISBN: 3-8322-0177-7, April 2001.

[42] T. Tourwe and T. Mens. Identifying refactoring opportunities using logic meta programming, 2003.

[43] 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, October 2002.

[44] N. Walkinshaw, M. Roper, and M. Wood. The java system dependence graph. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation*, September 2003.

[45] Weka 3 — Data Mining with Open Source Machine Learning Software in Java, http://www.cs.waikato.ac.nz/~ml/weka/. 2003.

[46] Scott A. Whitmire. *Object Oriented Design Measurement*. John Wiley & Sons, Inc., 1997.

[47] Ian H. Witten and Eibe Frank. *Data Mining*. Morgan Kaufmann, Los Altos, US, 2000.