Automatic Detection of Design Problems in Object-Oriented Reengineering

Oliver Ciupke
FZI Forschungszentrum Informatik
ciupke@fzi.de

Abstract

The evolution of software systems over many years often leads to unnecessarily complex and inflexible designs which in turn lead to a huge amount of effort for enhancements and maintenance. Thus, the reengineering of object-oriented software becomes more and more important as the number, age and size of such legacy systems grow. A key issue during reengineering is the identification and location of design problems which prevent the efficient further development of a system. Up to now this problem area has not been sufficiently supported, either by methods, or by tools.

In this paper, we present a technique for analyzing legacy code, specifying frequent design problems as queries and locating the occurrences of these problems in a model derived from source code. We present our experiences with a tool set which we implemented to support this task by automatically analyzing a given system and detecting the specified problems.

We applied our tools to check violations of a number of well-known design rules in existing source code taken from several case studies, both from industrial and academic fields. These experiments showed that the task of problem detection in reengineering can be automated to a large degree, and that the technique presented can be efficiently applied to real-world code.

Keywords: *Object-oriented reengineering, design problems, tool support for reengineering, model capture, problem detection*

1: Introduction

The evolution of software over many years often leads to unnecessarily complex and inflexible systems. It is hard to predict the effects of changes or even to determine what has to be changed and thus enhancements and maintenance of these systems are becoming more and more expensive. This is already true for many object-oriented systems. Sometimes they are first-generation oo systems when methods were less mature, sometimes only an object-oriented language was used but the developers were not trained in object-oriented design, but often the system is simply very large and the class structure itself is not as flexible and simple as it could be.

In order to allow further efficient evolution, the structure of such a system has to be improved. In order to do this, we need to find out which parts of the structure prohibit further enhancements to the system and what sort of problem we are facing. We want to know which classes, subsystems or methods have to be changed and what kind of changes we must make. We call this task *problem detection*. It is one phase in the life cycle of

reengineering as defined by [4]. Problem detection is hard to do manually. Some of the reasons for this are:

- Programs which have to be reengineered tend to be very large.
- Systems are developed by different developers or teams. Design problems can affect several different subsystems and thus cannot be detected locally.
- It is often unclear *what* exactly to search for.

For all these reasons we need *automatic support* for the task of problem detection, i.e., tools which accept source code[1] as input and point to design fragments which are candidates for design problems. The result is a list of locations where problems have been found together with their classification. It can provide a starting point for reorganizing a system.

In this paper, we describe a method for the automatic detection of design problems in legacy code. We implemented tools to support this method and applied these to a set of case-studies. The following criteria are considered important for our method to fulfill:

- Method and tools work on the design level. In our opinion, the quality of the overall structure determines the flexibility of a system, i.e., how expensive it is to extend or maintain it.
  In the context of our reengineering work, we use the term "design" not only for the corresponding activity, but also for the "actual" structure, as we can observe it in a given system – even if it was never planned to be like it is.
- All of the required information is extracted from source code. We cannot rely on design documentation, since it often does not exist or is in most cases inconsistent. We cannot rely on information that has to be collected dynamically during run time either. First, there is no practically applicable method to ensure that all the necessary parts and paths of a program are executed. Second and even more important, many systems, especially the huge ones which need reengineering the most, simply cannot be run apart from the dedicated hardware or environment for which they are built.[2] Famous examples are systems in the telecommunications area and other embedded or real-time systems. Similarly, systems in the commercial area often need to have most of their data stored in databases to get meaningful program runs.
- To cope with large amounts of code, problem detections should run fully automatically. It may be necessary to check the results to determine their relevance, but this set of results should first be produced without any user intervention.
- A problem detection method should point us directly to the problems we are searching for. Software metrics, as a counter-example, only deliver numbers which require further interpretation. Only together with a set of thresholds can they point us towards potential problems.
- The approach (and associated tools) should be language independent (as much as possible). Since the design of a system is to some extent independent of the implementation language, this should be reflected in the chosen method as well. Nevertheless, it should be possible to add issues that are typical for a certain language.
- An approach to problem detection in reengineering should cover a wide range of possible problems.

[1] or a representation of it for languages such as Smalltalk

[2] And you hardly want to build your reengineering tools on that environment.

This paper is organized as follows. Section 2 describes our approach to automatic problem detection by querying design. Section 3 presents a tool set which implements the approach. Section 4 describes an experiment where we applied our tool set to detect violations of design guidelines in several case studies. Section 5 discusses related work and how our approach differs. Section 6 concludes.

The work described in this paper was done within the joint project FAMOOS which is funded by the European Commission as ESPRIT project 21975 within the software technologies domain. The goal of FAMOOS is the development of methods and tools for the reengineering of object-oriented legacy systems.

2: Automatic problem detection

We assume that legacy systems are given in the form of *source code* or an equivalent representation. In order to search for problems in the overall software structure, we must leave the concrete implementation behind and move towards a higher level of abstraction at which specifications of those problems are given. We do this by parsing the source code and producing high-level *design information*. What we then gain is a description of the *actual design* of the system, which often differs from what can be found in design documents produced in earlier phases of the development process.

To be able to express and interpret the information gathered from source-code, we have set up a *meta-model* for object-oriented systems. This meta-model defines the different entities and relations that may occur in the design of an object-oriented program. A model of a legacy system which conforms with the meta-model can be stored as graph, as entities and relations, or as predicates. This makes it possible to query and manipulate the model using different query languages.

2.1: Querying design

For detecting problems in the design of a system, we have to search for certain patterns representing those problems in the model built of the system. This means we have to be able to *specify* problems and to *query* the model about the existence of a specified problem. The result of such a query is a piece of design specifying the location of the problem in the system. Such a piece of design in a formal model is often referred to as a *design fragment*.

Several ways for specifying queries on a design model exist. A model can be understood as *typed graph* and the queries become algorithms working on this graph. A model can be specified by *sets and relations*. Queries then take the form of relational algebraic expressions. A model can also be expressed by logical propositions and be queried using *predicate calculus*, e.g., using a logic programming language.

The three above-mentioned approaches represent different viewpoints about the same meta-model. They are equivalently powerful[3] and the corresponding models and queries can be converted into each other. Each of them has its advantages in certain tasks and all three of them are used in our tools for different purposes. During the experiment described in the next section, we primarily used predicates implemented as Prolog clauses [7, 31], since we felt they were the easiest to map to the guidelines given in natural language. We will also concentrate on this way to query designs in the remainder of this paper.

[3]Given not full predicate calculus, but only horn-clauses are used and we only consider predicates concerning our (finite) meta-model.

```
% Base classes should not have knowledge about their descendants
knowsOfDerived (Class, DerivedClass) :-
   % Both Class and DerivedClass must be classes
   class (Class), class (DerivedClass),
   % DerivedClass is a direct or transitive descendant of Class
   trans (inheritsFrom, DerivedClass, Class),
   % The base class knows its heir
   knows (Class, DerivedClass).

% A class 'knows' another class, if
knows (Class1, Class2) :-
   % it inherits from that class, or
   class (Class1), class (Class2),
     inheritsFrom (Class1, Class2);
   % it has an attribute of that type, or
   hasAttribute (Class1, Attr), hasType(Attr, Class2);
   % it has a method which returns an object of that type, or
   hasMethod (Class1, Meth1), returns (Meth1, Class2);
   % it has a method which calls a method of that class, or
   hasMethod (Class1, Meth1), calls (Meth1, Meth2),
     hasMethod (Class2, Meth2);
   % it has a method containing a parameter with type of that class.
   hasType(Param, Class2).
```

Figure 1. Formalization of a design rule in Prolog

2.2: Example

Here we provide an example of how a design problem and a query which detects this problem may look like. We have chosen a problem related to a *design guideline*. If design guidelines (or heuristics) make a statement about good design, then a violation of such a guideline may indicate a design problem. The following is taken from [27]:

> *"Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes."* (Heuristic 5.2)

If base classes depend on their descendants, then these descendants cannot be altered independently of their ancestors. But exactly this functionality is often needed during the evolution of an object-oriented system. So a violation of this guideline points to a spot in a software structure, where this structure is hard to change or maintain.

In order to be checked automatically, this rule must now be formalized. This can be done in different ways, e.g., using different query languages. An elegant formalization of this example can be done using Prolog, which we also used for many other queries in our prototypes. A base class would violate this heuristic, if it had knowledge of one of its direct or indirect heirs. *Knowing* a class means being dependent on the interface or the implementation of this class. In order to also consider the indirect heirs the transitive closure of the inheritance relation is also required. These two remarks lead us to the Prolog clause shown in Figure 1 which is satisfied by the entities setting up the design problem (for definition of `trans` see Appendix B).

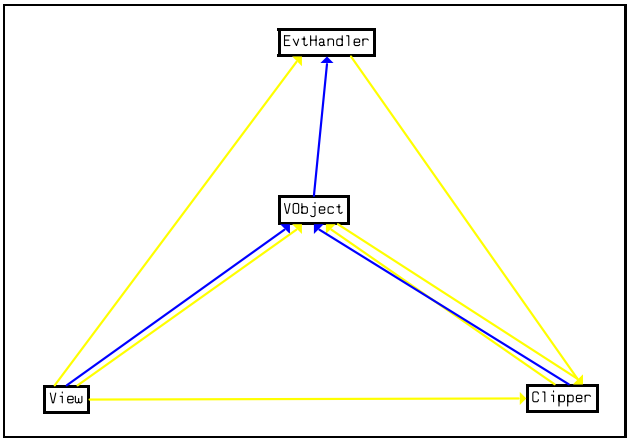

Figure 2. Visualization of a detected problem

In the 2.2 version of the object-oriented framework ET++ for example, the preceding Prolog query detected 23 violations. One of these is shown in Figure 2.[4] Boxes represent classes, dark arrows represent inheritance relationships, light arrows the existence of method calls between the methods of these classes. Here we even have two overlapping violations at once. The class "Clipper" inherits "VObject" directly and "EvtHandler" transitively. Both EvtlHandler and VObject contain calls to a method of Clipper. This means that changes to Clipper may require EvtlHandler and VObject to be changed as well. Both can be found at a high position in the inheritance hierarchy, where changes affect many other classes in turn. Actually, Figure 2 does not depict the full degree of complexity of this design fragment. In the complete graph of this ET++ version, each of the classes is connected to at least 50 other classes, VObject even to 376. What results is that we have found a design fragment that is resistant to change and difficult to maintain.

Possible reorganizations which could simplify this situation would be to introduce abstract base classes or migrating methods or attributes between classes. Which of the solutions is suitable depends on the overall structure and on the particular goals and requirements of the reengineering of a specific system.

ET++ does of course not resemble a typical example of a legacy system, since all in all it has a well designed structure. Nevertheless we used it here, because it is well-known in the object community and we wanted to show an example of a problem in a "real" object-oriented design.

3: Tool support

A tool set named GOOSE[5] intended for various reengineering tasks was developed at the Forschungszentrum Informatik in the framework of the FAMOOS project. It supports software structure visualization, metrics calculation, querying design and automatic reorganization. The structure of the tools relevant for the task of problem detection is shown in Figure 3. Boxes represent tools, ovals represent persistent data. Arrows denote the data

[4] The figure shows no methods, but only classes. To get a more abstract view, methods and relations between methods have been collapsed into their classes using our tool REVIEW to produce this visualization.

[5] The name originates from a preceding concept named "Moose" and the fact that it deals with graphs.

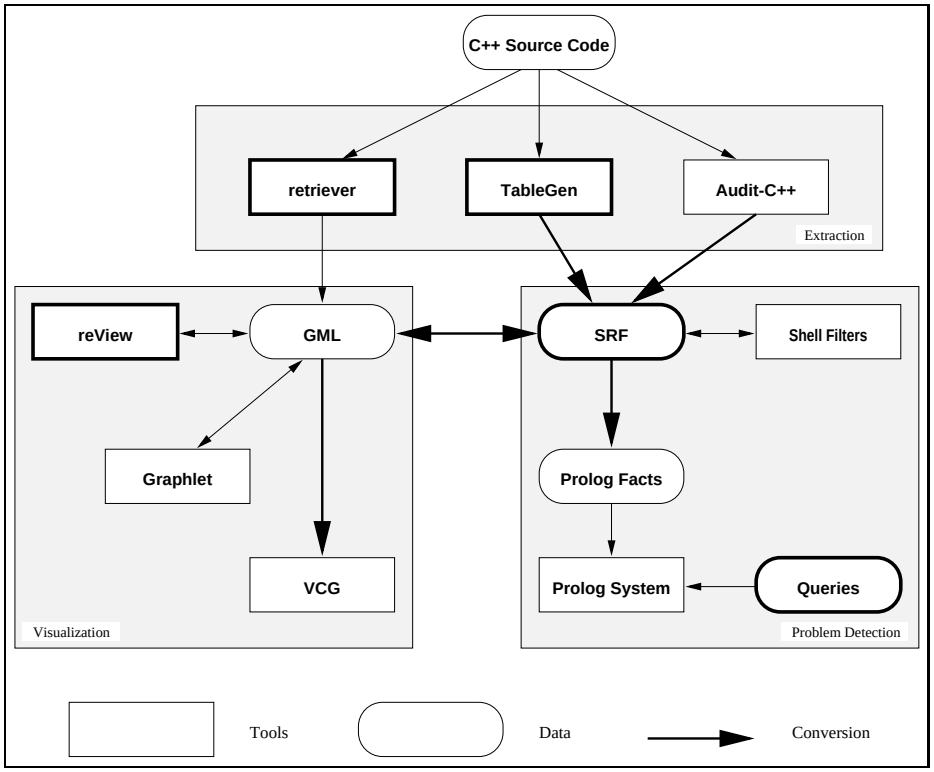

Figure 3. Overview of tool interaction

flow between tools or between different representations.

A large portion of the functionality was achieved by connecting already existing tools, rather than implementing them ourselves, e.g., for visualization and querying. Thickly highlighted borders or arrows in the picture are things we had to implement or define ourselves. Thick arrows represent conversions between different data formats, mostly implemented in the form of Perl scripts.

Detecting problems in a system is done as follows. First, the source code of the system is parsed. This can be done by different parsers depending on the environment. (See the upper grey box labeled "Extraction"). The system can now be visualized to provide a first view of the interdependencies. We use different publicly available visualization for this. If necessary, manipulations or simplifications can be made by hand (e.g., deleting undesired nodes) or done automatically (e.g., make common simplifications to generate a visualization on the class level). For the actual problem detection, the information is converted to Prolog facts. Queries formulated as Prolog rules examine this information and deliver the locations of problem candidates as result.

We now describe the individual tools and formats in more detail. The parsers for extracting design information from source code can be freely exchanged. So far we have made experiments with C++ and Chill [11]. Further extraction tools and parsers for Smalltalk, Ada and Java are available within the FAMOOS project.

retriever is an extraction tool which is connected to the API which comes with the UNIX version of the software development tool SNIFF+.

Audit: GOOSE can import tables produced by the commercial analysis tool AUDIT as well.

The information is mostly a subset of that exported by TableGen, but we maintain this as an easy way to integrate other languages such as Ada and Java.

TableGen [19] is a parser built using the parser library Fast. Compared to retriever and Audit it is more stable and delivers additional information about function calls and variable accesses.

The different data formats we use to represent and exchange information correspond to the different ways to query design models as described in section 2.1. All of them are used for different tasks within Goose:

Simple relational format (SRF): SRF is an intermediate format to facilitate conversion of data formats required by different third party tools and to provide an easy-to-use interface to a design model for interactive querying.

GML (Graph Modeling Language)[6] is an upcoming standard for storing graph data.

Prolog facts: Design information can be converted to Prolog facts to query it with a Prolog system.

Several tools can query, manipulate or visualize the extracted design information:

Prolog system: In our current work, we use the environment ECLiPSe[7] for Constraint-Prolog to run deductional queries.

reView supports a large number of manipulations including a grouping (abstraction) functionality for graphs given in GML format.

Shell filters: Combinations of common shell filters like `grep, sort, comm, join, cut, wc, uniq` and `awk` are used together with the SRF format for ad-hoc queries, in order to filter noise or to merge data from different sources.

Graphlet is the tool we currently use most for visualizing design models. It supports both automatic and manual layout and allows editing graphs.[8]

VCG Supports fewer layout algorithms than Graphlet and does not allow the interactive manipulation of layouts, but is still used for some purposes.[9]

4: An experiment: checking guidelines in legacy systems

During our first experiment, we used guideline violations as specifications of problems to be searched for. A lot of material often given in the form of rules or heuristics exists in the literature about what *good* object-oriented design should look like. Every rule corresponds to a potential problem in an object-oriented design, where the rule has *not* been followed. Originally these guidelines were meant to be followed by a human developer when creating a new design, rather than as input for an automatic tool testing whether they are met in the actual design of a legacy system. Nevertheless, we examined whether such rules could be used to automatically detect problems.

First, we investigated how many of the guidelines found can be formalized within our model. Though these design guidelines cannot cover every possible design problem, they

[6] `http://www.uni-passau.de/Graphlet/GML/index.html`
[7] `http://www.ie.utoronto.ca/EIL/ECLiPSe/eclipse.html`
[8] `http://www.uni-passau.de/Graphlet/`
[9] `http://www.cs.uni-sb.de:80/RW/users/sander/html/gsvcg1.html`

can determine how well the task of problem detection is done by our approach. Secondly, we implemented the formalized guidelines as queries and applied them to several case studies, to demonstrate the practicability of both method and tools.

During our study, we encountered about 280 guidelines originating from various authors ([27, 15, 14, 18, 12, 30, 16, 13, 20, 33, 29, 8, 3, 23, 22] and several others). Many of these guidelines were equivalent or at least very similar to each other. Some were very implementation specific and thus too far away from what we considered to be a design rule. In the end, this left us a list of 59 rules which in our opinion best covered our area of interest for further examination [2].

4.1: Formal testability

Nearly all design guidelines were originally intended to give a human developer hints for designing a new system or new parts of a system. They are normally given in natural language and state what he or she should do or should avoid. For our purpose in contrast we wanted the rules to be checked on existing systems within a reengineering process and we wanted them to be checked automatically by a tool. Being able to implement such rules implies having an *exact formal definition* first. Thus we determined to what extent such formalizations can be made. *Testability* gives an idea of how precise an automatic search for the violation of a given guideline can be. We use a classification similar to [12].

The guidelines can be classified according to how close a formal definition according to our model can get to the original definition. We categorized guidelines into five classes of testability based on the relationship between two associated sets: the set V of design fragments violating the guideline and the set I of design fragments identified by the given formalization.

Exactly testable: $I = V$
: An exact formalization can be given for this guideline.

Partially testable: $I \subset V$
: Only a subset of the violating fragments can be found, but every fragment found violates the guideline.

Vaguely testable: $V \subset I$
: Every violation is found, but some fragments found may not contain violations.

Symptomatically testable: I and V correlated, but neither $I \subset V$ nor $V \subset I$
: These guidelines combine the disadvantages of the partially and vaguely testable ones.

Not testable: I is undefined.
: We cannot give a satisfying formal definition for the guidelines in this class. The main characteristic of these guidelines is their lack of stringent demands. They may use information not extractable from source code, such as about dynamic behavior, or they may regard quantity and not include sharp limits to obey.

Table 1 shows the number of guidelines in each of the testability classes. Since the table is ordered from highly useful to (for our purpose) useless guidelines, one can see that 46% are exactly testable and nearly two thirds identify only actual violations. By setting thresholds to quantities or by slight changes to the definition, even more guidelines can be migrated into an other testability class and this way be turned into applicable rules.

Category	Quantity	Percentage
Exactly	27	46 %
Partially	10	17 %
Vaguely	1	2 %
Symptomatically	2	3 %
Not testable	19	32 %
Sum	59	100 %

Table 1. Size of the testability classes

4.2: Application to case studies

The rules implemented so far were applied to seven case studies written in C++: two versions of the well-known object-oriented framework ET++ [32], Rheingold, which is a development for a customer of our own group, and four industrial software systems or subsystems, named CS_4 to CS_7.[10] The systems run under different UNIX variants or under NT. To keep the results interpretable, we always measured only the systems themselves, i.e., without eventually underlying frameworks, such as the MFC.

The number of violations for a selection of the implemented rules in each case study is shown in Table 2. In the upper part, the table shows the sizes of the case studies in terms of lines of code, classes and methods. In the lower part, it shows the number of hits for each query.

As can be seen from the table, some of the queries do not report any violations at all (e.g., private methods). Others report huge numbers of violations for some of the systems (e.g., private attributes and unused inheritance). The reason for this is that many guidelines reflect a certain *style* of designing and programming. In some cases inheritance is only used for the sake of polymorphism, in others it is also used for the reuse of implementation. Thus, if we want to use the queries to point the reengineer to actual problems, we must make a selection according to the style of the given system.

4.3: Performance

By far most of the time for the whole analysis process was needed for parsing the source code to produce design information. Performance was measured for ET++-2.2 (40 kLOC). ET++ was not the biggest of our case studies in terms of lines of code, but needed the longest parsing times, though.[11] TableGen needed 27 minutes for this task. The other parser used, RETRIEVER, needed only about two minutes, but this one does not deliver some of the information we need for some of the queries. (All measured on a Sun Ultra 1, Processor: Model 140 UltraSPARC, 64MB RAM).

Since parsing only has to take place once for every case study, the performance turned out to be fast enough, since further analysis work can always re-use already prepared design information. All other steps took less than half a minute. The problem detection itself running on ECLiPSe was amazingly fast. All queries returned their results immediately, or in at most 25 seconds in cases where a transitive closure was needed.

[10] Since these are subject to non disclosure agreements, we do not use their full names here.

[11] Goose was already being successfully used on a 5 MLOC system, but mainly for visualization purposes and less for automatic problem detection [28].

Case Study	**ET++-2.2**	**ET++-3.a1**	**Rhein-gold**	**CS_4**	**CS_5**	**CS_6**	**CS_7**
Size:							
Lines of Code	40167	48747	14165	68638	35180	53370	276000
Classes	352	543	136	87	17	53	806
Methods	3738	4981	607	1285	192	32	4536
Free Functions	425	323	63	37	4	130	n.k.
Query:							
A class should not contain more than six objects	8	12	2	19	0	1	92
Inheritance hierarchy too deep (> 6)	27	6	0	0	0	0	1
Avoid multiple inheritance	0	0	0	6	0	0	12
Inheriting the same class twice	0	0	0	0	0	0	1
Attributes should be private	224	402	260	12	15	29	560
Do not turn an operation into a class	103	197	23	10	7	3	172
Base classes should not have knowledge about derived classes	23	14	0	0	0	0	0
Divide large classes (> 12 methods)	96	118	15	39	3	0	91
Reduce number of arguments (> 4 args.)	0	0	0	0	0	0	0
Unused inheritance	293	421	22	14	10	0	425
Containment implies violation of use	154	249	7	58	0	7	591
Do not declare private methods	0	0	0	0	0	0	0

Table 2. Number of violations in different case-studies

5: Related work

CCEL [10] is a meta language in C++ to define rules for the entities of a C++ design. It is possible to check these rules for given C++ sources. CCEL defines an object-oriented meta-model, which only focuses on the *entities* of a design. Furthermore CCEL works exclusively with C++. A lot of the design rules we investigated in our work required a meta-model including relations. Thus our method deals with both entities and relations. It is also not restricted to one language.

Law-governed architecture [24, 25] is a general approach to ensure rules during software development. It distinguishes two kinds of rules: those concerning the development process and those concerning system structure. The approach observes the development of a *new system* and aims at the avoidance of problems there. It introduces mechanisms to detect design problems, but does not define rules for their detection. In contrast, our method examines already existing systems and we define queries to do that.

MeTHOOD [12] improves object-oriented designs at the level of a meta-model. As before, MeTHOOD can be applied during the design phase of *forward engineering*. A design model can be built using a special editor and can then be checked for the violation of rules.

GOOSE can extract the design model from source code and comes with a catalogue of formally defined rules. On the other hand, MeTHOOD is more general in the sense that it provides concepts for transforming designs and for resolving problems.

Object-oriented software metrics [6, 5] map pieces of design or implementation to a certain, normally *numerical value*. Most object-oriented metrics measure the different kinds of complexity of classes or the cohesion between classes [19].

If metrics are used for problem detection, their results must first be interpreted. For example one must add thresholds to finally get statements about what has to be considered as a problem. Up to now, this has been a difficult task. First, because of lack of experience in measuring large systems of real-world code; second, because the limits vary with the languages, programming style, details in the definition of a metric, etc. Furthermore, metrics can only detect problems related to magnitudes of values.

Our collection of queries for problem detections contains as well some which are based on quantities. GOOSE can also be (and has been) used for measuring software metrics. And of course, the number of occurrences of a certain problem in a system defines a metric, so both subjects are strongly related. But our main focus in the work presented here was on problems based on the more structural properties of a design, which are hard to define in terms of quantities and thus needed a different approach to automatic detection.

Software reflexion models [26] can be used to find *mismatches* between high-level designs and the corresponding source code. The approach requires the reference model, to which a system can be compared, to be given in advance. In our approach, arbitrary problems that can be formulated as queries can be detected with only the source code as a given.

Conceptual module querying [1] is a method for supporting queries about the relationships between fragments of source code. Conceptual module querying does not

aim directly at the task of problem detection and thus only supports certain types of queries. Furthermore it focuses on program entities that basically belong to the level of source code. Our approach aims at as much as possible arbitrary reasoning about entities and relations of the design level.

Style checker: Several tools exist for checking rules in the given source code at the *implementation level*, e. g., searching for the improper use of references. The best known among these is probably LINT [9] which analyses C programs. GOOSE checks rules on the *design level* and is to a large extent language independent.

6: Summary and future work

Automatic problem detection reduces the effort of browsing through large amounts of code during the reengineering process. It can also assist developers in finding weak parts in the design of a piece of software, which they would otherwise not be aware of.

We presented an approach for detecting problems in the designs of object oriented systems. Problems are specified as queries on a design model. A problem can be detected using our method, if it can be formalized in terms of the query languages we used. In this way, we can provide a satisfying formalization for about two thirds of the guidelines we collected from literature.

Our method is supported by a tool set that we implemented with moderate effort, by connecting well-understood techniques and existing tools. The tools can cope with regular industrial code up for very large systems (MLOC) and proved to be fast enough to be practically applicable. These tools can check a catalogue of problem detections fully automatically and deliver results directly pointing the user to places in the source code which contain potential design problems. Applying our tools to a further programming language only required adding a parser (or extractor) for this language, which makes our approach highly language-independent.

The catalogue of queries as implemented so far is mainly constituted by design heuristics taken from the literature and some of them from our own experience. Our experiments showed that these give good hints about where to start reorganizing a system, but that legacy systems often suffer from further problems not addressed by such common rules. In the future, we want to collect such problems, to formalize them and to implement them as queries, as well.

The tools only run on certain platforms so far, require other software to be installed before, and require relatively highly skilled users. Hence, in the current phase of the FAMOOS project, a problem detection functionality is being built into the quality assurance and metrics tool AUDIT by the Sema Group. This will provide a solution that can be applicable in everyday software projects and for a large variety of environments.

References

[1] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software engineering. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 64–73. IEEE Comput. Soc, Los Alamitos, CA, USA, 1998.

[2] Holger Bär and Oliver Ciupke. Exploiting design heuristics for automatic problem detection. In Stéphane Ducasse and Joachim Weisbrod, editors, *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, number 6/7/98 in FZI Report, June 1998.

[3] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, 2 edition, 1994.

[4] Eduardo Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, pages 45–52, January 1998.

[5] S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[6] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, pages 197–211, November 1991. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

[7] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin and New York, 1981.

[8] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice Hall, London, 2 edition, 1991.

[9] Ian F. Darwin. *Checking C programs with* `lint`. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, October 1998.

[10] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A metalanguage for C++. Technical Report CS-92-51, Department of Computer Science, Brown University, October 1992. Sun, 13 Jul 1997 18:30:16 GMT.

[11] Thomas Genßler and Oliver Ciupke. Toolgestützte Codeanalyse von Telekommunikationssoftware in Chill. Systemdokumentation und Projektbericht, Forschungszentrum Informatik, 1997.

[12] Thomas Grotehen and Klaus R. Dittrich. The MeTHOOD approach: Measures, transformation rules, and heuristics for object-oriented design. Technical Report ifi-97.09, University of Zürich, Switzerland, August 27, 1997.

[13] Walter L. Hürsch. Should superclasses be abstract? In Mario Tokoro and Remo Pareschi, editors, *Object-Oriented Programming, Proceedings of the 8th European Conference ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 12–31, Bologna, Italy, 4—8 July 1994. Springer.

[14] Ralph E. Johnson and Brian Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[15] John Lakos. *Large-Scale C++ Software Design.* Addison-Wesley, 1996.

[16] Wilf LaLonde and John Pugh. Subclassing =/ subtyping =/ is-a. *Journal of Object-Oriented Programming*, 3(5):57–59,62, January 1991.

[17] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programming. *IEEE Software*, pages 38–48, September 1989.

[18] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-oriented programming: An objective sense of style. In *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 11, pages 323–334, San Diego, CA, September 1988. A short version of this paper appears in *IEEE Computer Magazine*, June 1988, Open Channel section, pages 78-79.

[19] Radu Marinescu. The use of software metrics in the design of object oriented systems. Master's thesis, Universitatea Polytehnica din Timişoara, 1997.

[20] R. C. Martin. Object oriented design quality metrics. *Report on Object Analysis & Design*, September 1995.

[21] R. C. Martin. The Dependency Inversion Principle. *C++ Report*, 8(6):61–66, June 1996.

[22] Scott Meyers. *More Effective C++.* Addison-Wesley, 1996.

[23] Scott Meyers. *Effective C++.* Addison-Wesley, second edition, 1998.

[24] Naftaly H. Minsky. Law-governed regularities in object systems, part 1: An abstract model. *Theory and Practice of Object Sytems*, 2(4):283–301, 1996.

[25] Naftaly H. Minsky and Partha Pratim Pal. Law-governed regularities in object systems, part 2: A concrete implementation. *Theory and Practice of Object Sytems*, 3(2):87–101, 1997.

[26] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, October 1995.

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

[28] Fabian Ritzmann. Reverse Engineering of Large Scale Software Systems. Diplomarbeit, Universität Karlsruhe, June 1998.

[29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[30] Ed Seidewitz. Controlling inheritance. *Journal of Object-Oriented Programming*, 08(08):36–42, 1996.

[31] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.

[32] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[33] R. Wirfs-Brock and B. Wilkerson. Variables Limit Reusability. *Journal of Object-Oriented Programming*, pages 34–40, May/June 1989.

A: List of examined guidelines

The following table contains the names and references to all design guidelines we have examined so far. The reference column has to be interpreted as follows: references of the form *x.y* can be found in [27] under the stated number; *Chapter x* can be found in [15] in the stated chapter; *Rule x* can be found in [14]; for other sources a reference to the literature is given. We used names that are close to the original description, but sometimes were not very clear out of this context. Please refer to the cited literature for definitions or more detailed descriptions.

Name	**Reference**
exactly testable	
All data should be hidden within its class	2.1
Classes should only use the public interface of other classes	2.7
Spin off non-related information into another class	2.10
No illegal dependencies	3.5
Remove unused components of a class	4.5
A class should not contain more than six objects	4.7
All data in a base class should be private	5.3
Depth of inheritance hierarchies	5.4, 5.5
All base classes should be abstract classes	5.6, 5.7
Avoid multiple inheritance	6.1, 6.2
Check for accidental multiple inheritance	6.3
Keep class data members private	Chapter 2
Avoid data with external linkage at file scope	Chapter 2
Declaration and definition constraints	Chapter 2
Avoid hiding a base-class function in a derived class	Chapter 9
Use virtual destructors with virtual methods	Chapter 9
A component should only include the header files it depends on	Chapter 3
Include the required header files directly	Chapter 3
Avoid granting long-distance friendship	Chapter 3
Cyclic dependencies only within components	Chapter 4&7
Eliminate explicit case analysis on object types	Rule 2, 5.12
The top of the class hierarchy should be abstract	Rule 6
Separate methods that do not communicate	Rule 11
Reduce implicit parameter passing	Rule 13
Law of Demeter—class form	[17]
Law of Demeter—object form	[17]
Define abstract interfaces to used classes	[21]
partially testable	
A class should not be dependent on its users	2.2
Implement a minimal public interface that all classes understand	2.4

Common-code private functions should be hidden within their class	2.5
Eliminate irrelevant classes	3.7, 3.8, 3.10
Do not turn an operation into a class	3.9
Base classes should not know anything about their derived classes	5.2
Avoid explicit case analysis on attribute values	5.13
Avoid global data	8.1
Subclasses should be specializations	Rule 8
Avoid inheritance to achieve code reuse	home-grown
vaguely testable	
Do not override a base class method with a NOP method	5.17
symptomatically testable	
Do not clutter the public interface of a class with useless things	2.6
A class should capture one and only one key abstraction	2.8
not testable	
Minimize the number of messages in the protocol of a class	2.3
Keep related data and behavior in one place	2.9
Only object roles requiring different behavior should be modeled as classes	2.11
Avoid centralized control	3.1–3.4
Minimize coupling between classes	4.1–4.4
Distribute system intelligence vertically within containment hierarchies	4.8
Constraints in the uses relation of contained objects	4.13, 4.14
Factor the commonality of classes	5.8, 5.10
Common attributes of classes should be placed in a class contained by each	5.9
Attribute values versus inheritance	5.15, 5.18
Recursion introduction	Rule 1
Reduce the number of arguments	Rule 3
Reduce the size of methods	Rule 4
Class hierarchies should be deep and narrow	Rule 5
Minimize accesses to variables	Rule 7
Split large classes	Rule 9
Factor implementation differences into subcomponents	Rule 10
Send messages to components instead to self	Rule 12
Avoid large subsystem interfaces	home-grown

B: Transitive closure with Prolog

We implemented a Prolog rule `trans` giving the transitive closure of any relation. Some of the syntax may be proprietary to ECLiPSe, the Prolog system we used.

```
% transitive closure of a relation named "Relation"
% trans (Relation, A, B) iff A Relation^+ B
trans (Relation, A, B) :- transRel (Relation, A, B, []).

transRel (Relation, A, B, _) :-
  Goal =.. [Relation, A, B, _], call (Goal).

transRel (Relation, A, C, AlreadySeen) :-
  Goal =.. [Relation, A, B, _], call (Goal),
  not member (B, AlreadySeen),
  transRel (Relation, B, C, [A|AlreadySeen]).
```