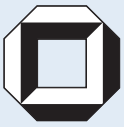
Adrian Trifu

Towards Automated Restructuring of Object Oriented Systems



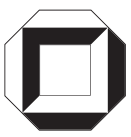
universitätsverlag karlsruhe

Adrian Trifu

Towards Automated Restructuring of Object Oriented Systems

Towards Automated Restructuring of Object Oriented Systems

von
Adrian Trifu



universitätsverlag karlsruhe

Dissertation, Universität Karlsruhe (TH)
Fakultät für Informatik, 2008

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2008
Print on Demand

ISBN: 978-3-86644-274-0

Towards Automated Restructuring of Object Oriented Systems

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
von der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)
genehmigte

Dissertation

von

Adrian Trifu

aus Satu Mare, Rumänien

Tag der mündlichen Prüfung:	13. Februar 2008
Erster Gutachter:	Prof. Dr. Dr. h.c. Gerhard Goos
Zweiter Gutachter:	Prof. Dr. Ralf Reussner

Acknowledgements

First and foremost I wish to express my deepest gratitude to my supervisor, Professor Gerhard Goos. It has been an honor and a privilege for me to be able to call myself one of his students. He has been a true "Doktorvater" whom I admire, and from whom I've learned both science and attitude towards life. He managed to make me feel confident and full of energy and ideas, after every single discussion that we had.

I would also like to express my gratitude to Prof. Ralf Reussner, for being my second advisor, as well as for the objective critique and constructive advice he gave me.

A very special thank you goes to my good friend Dr. Radu Marinescu, whose PhD thesis laid the foundations on which this work is based. Thank you for the many extremely productive and energizing discussions we had on Skype, as well as for always helping me focus on the really important things.

I'm also grateful to my brother and colleague Mircea Trifu, for his continuous support throughout the years. We had many inspiring discussions, especially after work, and he helped me a lot by reviewing my papers and the final versions of this thesis.

I owe my sincere gratitude to all of my students, especially Iulian Dragos and Urs Reupke, who deserve credit for implementing many of my ideas in practice. I am also grateful to my close colleagues at FZI: Dr. Christoph Andriessens, Dr. Holger Bär, Dr. Markus Bauer, Heike Döhmer, Dr. Thomas Genssler, Volker Kuttruff, Hilke Meffert, Dr. Benedikt Schulz, Dr. Olaf Seng, Peter Szulman and Jan Wiesenberger, for the fruitful discussions and their contribution to creating a stimulating and pleasant work climate. A special thank you to Benedikt, Markus, Christoph and Hilke, for helping me solve the many problems that naturally go together with settling down in a foreign country.

My heartfelt thanks also go to my parents, Lucia and Ioan, for the excellent education that I got from them, and through them, and especially for making me learn those foreign languages as a child. Without their total support, none of this would have been possible.

Finally, I am deeply grateful to Andreea, my dear wife. Thank you for putting up with long years of separation, quitting your job and then following me to Karlsruhe, so that I can carry on with my work. For standing beside me and sharing my dreams, I thank, and wholeheartedly dedicate this work to you...

July 2008

Adrian Trifu

Contents

1. Introduction **1**
- 1.1. Problem Definition . . . 1
  - 1.1.1. Context of the Work . . . 1
  - 1.1.2. Problem Statement . . . 3
  - 1.1.3. Goal and Success Criteria . . . 3
- 1.2. Approach . . . 5
- 1.3. Outline of the Dissertation . . . 6

2. Background **9**
- 2.1. Terminology . . . 9
- 2.2. What Constitutes a High Quality Design? . . . 11
- 2.3. Tool Supported Software Quality Assessment . . . 14
  - 2.3.1. Structural Models and Meta–Models . . . 15
  - 2.3.2. Software Quality Models . . . 16
  - 2.3.3. Analyses Based on Metrics . . . 17
  - 2.3.4. Analyses Based on Structural Pattern Matching . . . 19
- 2.4. Tool Supported Program Transformation . . . 20
  - 2.4.1. Normalization Approaches . . . 21
  - 2.4.2. Refactorings and Derived Approaches . . . 21
  - 2.4.3. Meta-Programming Approaches . . . 22

3. Related Work **23**
- 3.1. Manual Approaches . . . 24
- 3.2. Tool Supported Investigative Approaches . . . 26
- 3.3. Tool Supported Regressive Approaches . . . 29
- 3.4. Automated Specialized Approaches . . . 30
- 3.5. Automated Search Based Approaches . . . 32
- 3.6. Conclusions . . . 34

4. Design Flaws **35**
- 4.1. Design Context . . . 35
- 4.2. Design Flaw . . . 40
  - 4.2.1. Reference Structure . . . 40

4.2.2. Definition . . . 42
4.2.3. Scope . . . 43
4.2.4. Anatomy of a Design Flaw Specification . . . 46
4.3. Design Flaw Catalogue . . . 47

5. A Method for Diagnosing Design Flaws 49
5.1. Example: Schizophrenic Class . . . 49
5.1.1. Description . . . 49
5.1.2. Context . . . 50
5.1.3. Imperatives . . . 51
5.1.4. Pathological Structure . . . 52
5.1.5. Reference Structure . . . 53
5.2. Idea . . . 54
5.3. Indicators . . . 57
5.3.1. Defining Indicators for “Schizophrenic Class” . . . 57
5.3.2. Formal Definition of Indicators . . . 59
5.3.3. Quality of Indicators . . . 61
5.4. Tool Support . . . 64
5.4.1. Structural Models . . . 64
5.4.2. Implementing the Diagnosis Process . . . 66

6. A Design Flaw Based Restructuring Process 69
6.1. Closing the Circle . . . 69
6.1.1. Reorganization Strategies . . . 69
6.1.2. Restructuring Patterns . . . 74
6.2. Design Flaw Based Restructuring . . . 76
6.2.1. Considerations on Dealing with Design Flaw Interference . . . 77
6.2.2. Overview of Proposed Restructuring Process . . . 81

7. Evaluation 83
7.1. Prototype Tool . . . 83
7.1.1. Containment by Inheritance . . . 87
7.1.2. Explicit State Checks . . . 88
7.1.3. Collapsed Type Hierarchy . . . 89
7.1.4. Schizophrenic Class . . . 89
7.1.5. Misplaced Control . . . 91
7.2. Case Studies . . . 92
7.2.1. Experimental Goals and Approach . . . 92
7.2.2. Experimental Setup . . . 93
7.2.3. Discussion of Results . . . 95

8. Conclusions 107

8.1. Assessment of Proposed Method . . . 107
8.2. Summary of Contributions . . . 109
8.3. Future Work . . . 109

A. Design Flaw Catalogue **111**
A.1. Collapsed Type Hierarchy . . . 111
A.1.1. Description . . . 111
A.1.2. Context . . . 112
A.1.3. Imperatives . . . 112
A.1.4. Pathological Structure . . . 113
A.1.5. Reference Structure . . . 113
A.1.6. Diagnosis Strategy . . . 114
A.1.7. Reorganization Strategy . . . 115
A.2. Embedded Strategy . . . 116
A.2.1. Description . . . 116
A.2.2. Context . . . 117
A.2.3. Imperatives . . . 117
A.2.4. Pathological Structure . . . 117
A.2.5. Reference Structure . . . 118
A.2.6. Diagnosis Strategy . . . 119
A.2.7. Reorganization Strategy . . . 120
A.3. Explicit State Checks . . . 121
A.3.1. Description . . . 121
A.3.2. Context . . . 122
A.3.3. Imperatives . . . 122
A.3.4. Pathological Structure . . . 122
A.3.5. Reference Structure . . . 123
A.3.6. Diagnosis Strategy . . . 124
A.3.7. Reorganization Strategy . . . 124
A.4. Dispersed Control . . . 126
A.4.1. Description . . . 126
A.4.2. Context . . . 127
A.4.3. Imperatives . . . 127
A.4.4. Pathological Structure . . . 128
A.4.5. Reference Structure . . . 129
A.4.6. Diagnosis Strategy . . . 129
A.4.7. Reorganization Strategy . . . 130
A.5. Misplaced Control . . . 132
A.5.1. Description . . . 132
A.5.2. Context . . . 133
A.5.3. Imperatives . . . 134

A.5.4. Pathological Structure . . . 134
A.5.5. Reference Structure . . . 135
A.5.6. Diagnosis Strategy . . . 135
A.5.7. Reorganization Strategy . . . 137
A.6. Schizophrenic Class . . . 138
A.6.1. Description . . . 138
A.6.2. Context . . . 139
A.6.3. Imperatives . . . 140
A.6.4. Pathological Structure . . . 140
A.6.5. Reference Structure . . . 141
A.6.6. Diagnosis Strategy . . . 142
A.6.7. Reorganization Strategy . . . 143
A.7. Embedded Features . . . 145
A.7.1. Description . . . 145
A.7.2. Context . . . 146
A.7.3. Imperatives . . . 146
A.7.4. Pathological Structure . . . 146
A.7.5. Reference Structure . . . 147
A.7.6. Diagnosis Strategy . . . 148
A.7.7. Reorganization Strategy . . . 149
A.8. Containment by Inheritance . . . 151
A.8.1. Description . . . 151
A.8.2. Context . . . 152
A.8.3. Imperatives . . . 152
A.8.4. Pathological Structure . . . 153
A.8.5. Reference Structure . . . 153
A.8.6. Diagnosis Strategy . . . 154
A.8.7. Reorganization Strategy . . . 155
A.9. Premature Interface Abstraction . . . 156
A.9.1. Description . . . 156
A.9.2. Context . . . 157
A.9.3. Imperatives . . . 157
A.9.4. Pathological Structure . . . 157
A.9.5. Reference Structure . . . 159
A.9.6. Diagnosis Strategy . . . 159
A.9.7. Reorganization Strategy . . . 160
A.10.Collapsed Method Hierarchy . . . 161
A.10.1.Description . . . 161
A.10.2.Context . . . 162
A.10.3.Imperatives . . . 162

A.10.4. Pathological Structure 162
A.10.5. Reference Structure 163
A.10.6. Diagnosis Strategy 163
A.10.7. Reorganization Strategy 164

Bibliography **167**

Chapter 1.

Introduction

1.1. Problem Definition

1.1.1. Context of the Work

The ANSI/IEEE standard 729-1983 defines *software maintenance* as the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment". Experience has shown that the vast majority of all the costs associated with a complex software system arise during the maintenance phase [Huf90, Eas93, Som00, Erl00]. As a result, there is a compelling amount of pressure placed upon the research community, to come up with methods that ensure two things: on one hand, a high quality of the initial design, in order to minimize the need and extent of subsequent maintenance activities, and on the other hand, a low cost of the maintenance activities themselves.

Software design is the process of elaborating a software based solution to a problem, in a given domain. During this process, the designer performs modeling activities and structuring activities, in an interwoven fashion. Modeling activities occur primarily on an abstract, mental level, and involve identifying the key actors in the domain, problem and solution space, as well as defining their individual responsibilities towards the realization of the application's behavior. The resulting abstract model will be referred to as design intent. During the structuring activity, design intent is materialized in the source code, in accordance with the best practices, conventions and rules of the programming paradigm and language.

In the context of software maintenance, the quality of a design is determined by the ease with which the maintainer can understand the design, and operate changes therein. Design documentation other than the source code itself, may provide help, assuming that it exists and is up-to-date. In practice however, maintainers often have to rely on the source code alone, because high level documentation is either outdated or of little practical use. This is confirmed by studies showing that the majority of the time spent by maintainers is dedicated to reading and understanding existing source code [Sta84, Rug00].

Consequently, maintainability of a software system is decisively influenced by the static structure[1] of the source code, in one of the following two ways. On one hand, a structure that constitutes an "unnatural" solution for a given design intent (e.g. by employing cryptic or over-engineered constructs), affects the ability of the maintainer to understand the design. On the other hand, since any given structure favors specific types of change while hindering others[Mar03], the use of an unsuitable structure affects the maintainer's ability to operate changes in the design. In other words, in order for the design to be easily changeable, the designer must take into account the types of changes that are more probable to occur in the future. Often this is difficult, because "our ability to design for change depends on our ability to predict the future" [Par94].

In the course of a system's life cycle, repeated changes to the system's design have been shown to lead to a continuous degradation of its structure. This phenomenon manifests itself through both types of problems described above, and is known as "software aging" [Par94]. In order to counter the effects of software aging, the system needs periodic *restructuring*. Restructuring is a reengineering activity, in which the structure of the code is improved without affecting the observable behavior of the system. In practice, in an attempt to automate the restructuring process, structural anomalies, also called code smells [Fow99], are used as starting points for the analysis, since they can be detected automatically. A structural anomaly is defined as "a property of a structure fragment that has a negative impact on maintenance costs" [Ciu01]. Typical examples include an overly large number of methods or attributes in a class, supertypes that depend on their subtypes, and large switch constructs.

The classic restructuring process, based on structural anomalies, has three steps:

1. *Problem detection*: is concerned with finding instances of structural anomalies in the subject system. This step can be automated, using approaches such as [Ciu01, Mar02].

2. *Problem analysis*: covers the activities that are involved in deciding how to improve the structure of the design fragment under consideration. It involves understanding the design fragment in order to recover design intent, deciding on the types of change that are more probable to occur to the fragment, and finally, deciding on a new structure that appropriately reflects the design intent and accommodates the expected changes.

3. *Reorganization*: deals with the implementation of the new structure, decided upon in the previous step. This step can be automated, using common refactoring tools, such as the ones described in [RBJ97, Gen04].

[1] throughout this work, we will refer to the static structure of the system, simply as structure

1.1.2. Problem Statement

While both the problem detection step as well as the reorganization step are carried out in a systematic and largely automated manner, problem analysis is fundamentally an ad-hoc, largely manual process, which relies heavily on human technical expertise and intuition.

The cause behind the current state of affairs is the symptomatic nature of structural anomalies, which generally prevents putting such an anomaly in direct correspondence with a univocal reorganization strategy. A code smell for instance, represents a highly empirical, often purely numerical characteristic of the structure, that represents an excess or a violation of some limits that are commonly accepted as norms. The presence of a code smell is usually symptomatic for a multitude of various possible design deficiencies, and one structural deficiency may typically manifest itself through various kinds of code smells.

The lack of a clearly defined procedure for problem analysis, and the reliance on human intuition and expertise, confers an ad-hoc, laborious and ultimately costly nature to the entire restructuring process.

1.1.3. Goal and Success Criteria

The goal set out in the present work has been to transform the restructuring process, from a process that is heavily dependent on intuition and personal know-how, into a systematic process that supports automation. Thus, we wanted to bridge the gap that had previously existed in the tool chain, between quality assessment of software structure on one hand, and code transformation on the other hand. We achieved this goal by elaborating a systematic method, designed to:

Identify badly structured code: design fragments whose structure does not reflect the design intent in a way that is natural from the perspective of the object oriented programming paradigm, or aren't flexible with respect to certain expected types of change. Such fragments are characterized by an unnecessarily high effort needed for understanding and operating changes, and can therefore be called "badly structured".

Perform problem analysis: derive a sequence of code transformations, that makes the structure easier to understand and easier to change.

Below, we discuss a list of essential requirements to the elaborated method. At the same time, these requirements constitute criteria, that are relevant in assessing any method having a similar purpose.

Comprehensiveness: The method should be able to address all major problem types that arise at the abstraction level of component design. Lanza and Marinescu [LM06] identify three major groups of such problems: problems pertaining to class definition, problems that pertain to the correct use of inheritance hierarchies, and problems that pertain to cooperation between unrelated objects. Problems pertaining to the architectural level (e.g. subsystem decompositions) are not addressed by this work.

Causality: The method must guarantee the existence of a causal link between the deficiencies that exist in the initial structure, and the derived sequence of transformations, which should thus be in accordance with the choices that a human engineer would make when manually restructuring the system. In essence, this requirement ensures that decisions are not based on indirect or global measures, such as coupling, cohesion and complexity, but have a causal link to the underlying design intent, and take into consideration the types of changes that are expected to occur to the respective fragment, in the foreseeable future. In other words, corrective measures must be tailored to the local realities of the fragment that is transformed, not aimed at prettifying overall statistics. This guarantees an in-depth, rather than cosmetic improvement of design quality. Throughout the thesis, we will occasionally refer to approaches that do not satisfy this criterion as being *symptomatic*. Metaphorically speaking, symptomatic methods focus on the appearance, rather than the substance of a design's structure.

Systematic process: The process described by the method must be systematic, in the sense that stipulates what decisions need to be made, at what times, and depending on what parameters. Thus, the decision making process can rely less on the skill level of the maintainer performing the operation, because he must only reason in terms of each decision, in a predefined decision tree, not on the structure of the decision tree itself. However, determinism of the process does not mean that decisions are made without human intervention. In particular, assessing design intent or expected types of future change require human input.

Automation: Automation is the major contributor to reducing the cost of the restructuring process itself, and making the method practicable for large industrial systems. The method should allow the implementation of tools that automate most of the activities in the problem detection and analysis steps. Nevertheless, the need for human intervention cannot be eliminated completely. In particular, human intervention is required to confirm the presumed design intent, and to assess the probability of various types of future changes, to the design. The attained level of automation should allow the use of the method on medium to large sized systems (i.e. over 80,000 LOC), in a matter of days or weeks.

1.2. Approach

As pointed out earlier, code smells provide a good starting point for the identification of bad structure. Nevertheless, a meaningful reorganization is determined by the *design context* of the fragment, which consists of the two elements described above: design intent and the spectrum of expected changes.

Our approach is based on the simple idea of constructing higher-level entities, called design flaws, as triplets that consist of a *design context*, a *pathological structure* and a *reference structure*. As argued earlier, a given design intent may materialize in various different target structures, which may or may not be qualitatively equivalent, from the viewpoint of maintainability. However, by relying on the software engineering body of knowledge, we can derive a set of minimal requirements that the structure must possess, in order to optimally meet the rigors of the design context. A design flaw's reference structure is a template structure, showcasing one of potentially several structures that fulfill the requirements posed by the design context. Conversely, the pathological structure represents a template structure that disregards one or more such requirements, and which has relevance in common restructuring practice. In other words, the pathological structure illustrates an anti-pattern in the given design context, while the reference structure represents a possible solution, that does justice to the rigors of the design context, and is therefore optimally maintainable within the given design context.

By defining the concept of design flaw in this way, we ensure an unambiguous mapping between a "bad structure" and a corresponding "good structure", for an arbitrary design context. Thus, a design flaw forms a sort of pattern that can be used for recording restructuring know-how.

The procedure describing the transition from a pathological structure to the reference structure that matches the design context, is called *reorganization strategy*. The process of detecting and confirming a design flaw instance is called *diagnosis*. Diagnosing a design flaw means that the design context with its two components, and the pathological structure, match those described in the design flaw specification. Since the pathological structure in a design flaw is put in direct correspondence with the reorganization strategy corresponding to the reference structure, a separate problem analysis step is no longer required (i.e. diagnosis replaces both problem detection and problem analysis).

Based on the idea discussed above, we present a tool supported method for diagnosing design flaw instances in large software systems. The method is iterative, each iteration consisting of two fully automated and one interactive step.

The first and the second step are dedicated to identifying the pathological structure that is characteristic for a given design flaw, along with the corresponding design intent. Both of these steps are fully automated, and employ state of the art structural pattern matching techniques on an abstract model of the system's structure, extracted from the source code.

Design intent is detected using a procedure that resembles medical diagnosis, where a disease is diagnosed based on a characteristic combination of symptoms. In our case, the symptoms are structural features, which we call indicators, that characterize a particular design intent in combination with the given pathological structure. Each design flaw possesses a characteristic set of indicators. Although structural anomalies are obvious candidates for indicators, an indicator must not necessarily describe a structural anomaly. As in medical diagnosis, a design flaw instance may not necessarily induce all its characteristic indicators. Furthermore, different design flaws may share common indicators. The level of confidence in the candidate instances obtained through this automated pattern matching, increases with the number of different indicators that are detected simultaneously.

In the third and final step, the maintainer must confirm the design intent as well as the types of expected changes to the fragment, as described in the design flaw specification. To this end, the maintainer is guided by a set of questions that are predefined for each design flaw. Based on the outcome of this step, the candidate flaw is either confirmed or rejected.

The main benefits of our approach can be summarized as follows:

- We can put the pathological structure in direct correspondence with a recommended reorganization strategy, that guarantees a causal treatment of the underlying design flaw. This problem-solution mapping is expressed in an explicit, pattern-like form.
- The maintainer follows a systematic, predefined process. Diagnosis is mostly automatic, resulting in a *named design flaw candidate*, that is either confirmed or rejected. The confirmation process consists of answering a set of questions that is predefined for each type of design flaw.
- Since we look at the interplay of several indicators which do not necessarily represent code smells, our approach is able to identify inappropriately structured design fragments, even if there are no obvious anomalies in the structure (e.g. very large number of methods, large method bodies, excessive cyclomatic complexity, etc).
- Based on the number and type of the observed indicators, candidate flaws can be sorted according to the probability that they represent real flaws, thus maximizing time efficiency.

1.3. Outline of the Dissertation

The rest of the work is structured as follows: Chapter 2 covers the basic terminology in the field of restructuring and provides a concise overview of the notions and previous

works, upon which we build our approach. Chapter 3 reviews and compares competing approaches, highlighting their deficiencies with respect to the criteria that have been set forth in section 1.1.3.

Chapters 4, 5, 6, and appendix A constitute the core of the dissertation. Chapter 4 defines the concept of design flaw and discusses the way design flaws are specified. Chapter 5 presents a tool supported method for diagnosing design flaws, while chapter 6 discusses the way in which deign flaws can be eliminated with the help of tool supported refactoring, and introduces a design flaw based restructuring process. Finally, appendix A contains a representative catalogue of design flaws, ready to be used in day to day practice.

Chapter 7 is dedicated to the validation of our tool supported diagnosis method through a series of case studies. Chapter 8 provides an overall assessment of the merits and contributions of the proposed design flaw based restructuring process, and gives some pointers for future research.

Chapter 2.

Background

As pointed out in the previous chapter, the goal of this work has been to develop a tool supported method, to identify badly structured fragments in object oriented code, and to derive sequences of behavior-preserving transformations that improve the structure of these fragments. Such a goal statement implies bridging two important fields of object oriented software engineering: tool supported software quality assessment and program transformation. The purpose of the current chapter is to provide a concise overview of these two fields. Thus, we start by reviewing some basic terminology. This is followed by a short discussion about what characterizes "good object oriented design", and finally, we review approaches that pertain to the two fields mentioned above, and are relevant from the standpoint of our work.

2.1. Terminology

For the purposes of this thesis, we define the following terms, as follows:

Software development: is the process of elaborating a software based solution to a problem, in a given domain.

Design: denotes both a process and the result of that process. The design process consists in "defining the architecture, components, interfaces, and other characteristics of a system or component" [IEE90]. The result of the design process is an abstract model, referred to as "the design of the system or component", and described by a set of design documents (see below).

Design document, design description: any document, either textual and/or graphical, that describes one or more aspects of a design. At an extreme, we can see the source code itself as being a detailed and up-to-date design description of the system.

Static structure, structure graph: referred to simply as "the structure", it is the set of all entities, their properties, and relations, which can be extracted from the source code of the system, using ordinary parsing and type analysis techniques.

We stop at this point, in order to reflect a little bit on the relations between the terms "design", "design description" and "structure".

The term "design" refers to a abstract, multi-faceted model, that exists in the mind of the designer. For example, such a model describes the system both from a statical as well as a dynamical perspective. Furthermore, in such a model, entities and their relations possess both design and application semantics. To illustrate what we mean by these two types of semantics, let us consider the case of a specialization hierarchy. Design semantics in this case, refers to the specialization relation as a general relation between individual classes of objects, and expresses the fact that the superclass represents a more general concept than the subclass. On the other hand, application semantics refers to the meaning of the classes involved in the specialization hierarchy, in the context of the application.

A "design document", or "design description", is an incomplete representation of this immaterial model, in a textual and/or graphical form. It is incomplete in the sense that it always highlights one or more particular facets of the design, while playing down or even completely ignoring others. For example, the source code representation in typical imperative languages, though large in size and very detailed, cannot be considered complete. In particular, it completely ignores the application semantics of the entities that form the design, and highlights a static view on these while downplaying the dynamic aspects. Nevertheless, the source code is the most complete design description there is. In theory, through the process of understanding, a human engineer can use the source code in order to reconstruct the original design in his mind.

Another useful property of the source code is that it is guaranteed to be the most up-to-date description of the design. This is why reengineering activities primarily, or sometimes exclusively, rely on the source code of the system.

And finally, the "structure" of a system is a mathematical representation of the source code, and thus equivalent to it. Because the structure is in effect an incomplete description of the design, we can say that "a design *has* a certain structure". Thus, throughout the rest of this work, we will use "*the structure of a given fragment of design*", as a shorthand for saying "the structure extracted from the source code description of that fragment of design".

Maintenance: is the longest and most costly phase in the lifetime of software systems, and is defined as "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" [IEE90].

Maintainability: refers to "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment" [IEE90]. The literature generally acknowledges three factors that contribute to an improved maintainability: ease of understanding the design, ease of operating changes in the design, and ease of extending a design.

External software quality: refers to the quality of a software system, as perceived by its users [Mey88]. In other words, external quality can be judged based on the runtime behavior of the program. A number of commercial and free tools (e.g. all lint-like tools such as FindBugs[1]) exist, that attempt to detect potential bugs by analyzing the source code for things like null pointer dereferencing, infinite recursion and invalid casts.

Internal software quality, design quality: refers to the quality of a software system, as perceived by its maintainers [Mey88]. In other words, internal quality is equivalent to maintainability. Since most maintenance activities involve the source code, maintainability and therefore the internal quality is decisively influenced by the static structure.

Structural anomaly, code smell: as we have seen before, the main task of the static structure should be to keep maintenance effort as low as possible. As a result, any characteristic of a fragment of the structure, that has a negative impact on maintenance effort, constitutes a structural anomaly [Ciu01]. Code smells [Fow99] are typical examples of structural anomalies. For example, "large method" constitutes a structural anomaly, because the large size of the method's body has a negative impact on the ease of understanding and changing, therefore on the maintainability of that method.

Restructuring: is the process of improving the structure, without affecting the system's external behavior (functionality and semantics) [CC90].

2.2. What Constitutes a High Quality Design?

In the life cycle of a software system, the design phase amounts to no more than 10-15% of the total effort [BMP87], yet it is of a paramount importance to all subsequent phases, of which by far the longest and most costly is the maintenance phase. Studies show that maintenance accounts for over 50% of the total time [LS81], and between 75-90% of the total effort [Erl00, Eas93].

It follows from the above, that design quality cannot be defined in absolute terms, but rather it must take into account its main stakeholders, the maintainers. Coad and Yourdon affirm that "the most important characteristic of a good design is that it leads to an easily maintained implementation" [CY91]. Therefore, the design must suit maintainers, the important question being *how*.

Maintainers perform two kinds of activities: they need to understand and change existing designs (here we also include extending a design). Therefore, a design is good if it makes

[1] http://findbugs.sourceforge.net/

understanding and changing it easy. Through an iterative refinement process, these basic requirements were distilled into a set of high-level principles [Mey88, CY91, LW93, Lak96, Mar03]. They generally touch upon three important themes: proper conceptualization, reducing complexity and improving modularity (i.e. low coupling and high cohesion), and are summarized below:

Single Responsibility Principle: A class should have only one responsibility. Martin formulates this principle as "a class should have only one reason to change" [Mar03].

The Open-Closed Principle: Software entities (classes, modules, etc) should be open for extension, but closed for modification [Mey88]. In other words, changes should be achievable only by adding new code, without changing existing code.

The Liskov Substitution Principle: Derived classes must be usable through the base class interface without the need for the user to know the difference. This principle can also be stated in terms of the design contracts as: "a routine redeclaration in a derivative may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger" [Mar03].

The Dependency Inversion Principle: Details should depend upon abstractions. Abstractions should not depend upon details. This principle can be translated to both classes and entire subsystems, and in effect means that a design entity should only depend on other entities, that are on the same or higher level of abstraction.

The Interface Segregation Principle: Clients should not be forced to depend on methods that they do not use. In other words, if a class has groups of methods that serve different sets of clients, each set of clients should use its dedicated interface.

The Reuse-Release Equivalency Principle: The granule of reuse is the same as the granule of release. In other words, this principle states that only components that are released through a tracking system can be effectively reused.

The Common Closure Principle: Classes that change together, belong together in a subsystem, and groups of classes that change for different reasons should not belong together in the same subsystem. This principle is similar to the single responsibility principle of classes, but it applies to the responsibility of an entire subsystem.

The Common Reuse Principle: Classes that aren't reused together should not be grouped together. In other words, the subsystem structure should reflect reuse of their contents.

The Acyclic Dependencies Principle: The dependency structure for subsystems must be a directed acyclic graph. In other words, subsystems should not depend on one another cyclically.

The Stable Dependencies Principle: Dependencies between modules must run in the direction of stability. This means that modules which are expected to change often should not be depended upon by modules that are harder to change than they are.

The Stable Abstractions Principle: A package should be as abstract as it is stable [Mar03]. This principle states that the more stable a subsystem is, the more abstract are the classes that it should consist of.

Since these principles are still rather general to be easily applied, they have been further refined into a multitude of heuristic rules, guidelines, and their equivalent anti-guidelines [Mey88, Rie96, Lak96, Ciu01, Mar03, LM06]. Still, elaborating a high quality design is hard, because of two important reasons.

First of all, many of the guidelines and rules are by their nature antagonistic. It is therefore impossible to comply with all of them at the same time. Rather, as Coad and Yourdon note, "a good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime" [CY91].

A consequence of this fact is that even though structural anomalies, such as code smells, may prove to be effective alarm signals in problem detection, they are neither guaranteed to indicate an existing problem, nor can they suggest the right corrective measures. In order to be able to decide whether the employed structure is appropriate or not, you have to understand the context of the analyzed fragment, and reconsider the trade-offs involved in that particular situation. The same is true in the case of deciding on corrective measures.

The second difficulty in elaborating high quality designs is change. The open-close principle demands that future needs be met by only adding new code, without touching existing code. Of course, this requires preparing the structure to be able to accommodate potential future needs. Certain kinds of future needs will be easy to foresee, while in other cases, this may prove to be hard or even impossible.

On the other hand, it is not possible to accommodate all kinds of potential needs, equally well. Thus, most of the times, the maintainer must decide which type of change to favor (Martin calls it "strategic closure" [Mar03]), and adapt the structure accordingly. In some situations however, it is better to adopt a demand driven strategy, as advocated by agile software development processes.

In conclusion, the successful day to day use of design principles and rules requires careful consideration and a lot of experience. As in other fields, a good way of improving one's skills is to study other successful designs, from which reusable solution ideas to common problems can be extracted. This constitutes the motivation behind the pattern movement in software engineering, initiated by Beck and Cunningham in [BC87], and later made popular by Gamma, Helm, Johnson and Vlissides, in [GHJV96].

In their book, Gamma et al. define the notion of *design pattern* as "a solution to a [commonly reoccurring] problem in a given context". The last part of this simple definition is very important, because it acknowledges the impossibility of universally good design, as argued above.

Motivated by the success of using a pattern language as a form of knowledge dissemination, the pattern movement rapidly expanded to cover a wide range of granularity, and to various application domains. From a granularity standpoint, we distinguish three levels, whereas the separation between them is not sharp [BMR+96]: idioms, component design patterns, and architectural patterns.

A series of empirical studies done by Prechelt et al. [PUT+01] investigate the general beneficial impact of design patterns on maintenance effort. In a different study [PUPT98], the same authors found that maintenance activities can be performed even quicker and with fewer errors, if explicit pattern documentation is present in the code.

From the standpoint of the present work, the above discussion allows us to draw the following conclusions. There are many rules and guidelines which take various viewpoints in order to describe properties of a good design, but compliance with all of them is impossible. Instead, trade-offs must be made by the designer, which take the local particularities as well as the change potential of the fragment into account. This is the reason for which the causality criterion, introduced in section 1.1.3 is essential for any restructuring methodology. Those methodologies that do not comply with this requirement are inherently limited in their capabilities, and the restructuring process remains ad-hoc.

2.3. Tool Supported Software Quality Assessment

Complex software systems represent key assets in today's enterprises. The "law of continuing change", first formulated in a classic study by Lehman and Belady [LB85, Leh96], states that a program that is used in a real-world environment must change, or else become progressively less useful in that environment.

Another important result of that study, is that as a program evolves, it becomes more complex, and extra resources are needed in order to preserve and simplify its structure. This happens because repeated changes (which we have seen are unavoidable) tend to degrade the system's architecture, a phenomenon referred to as "software aging" [Par94], or "software decay" [Fow99]. In particular, a lack of understandability and flexibility in the design, results in increased effort and costs associated with regular maintenance activities. Therefore, in order to keep the amount of effort and the costs under control, methods and tools are required to periodically assess the state of the software assets, and if needed, intervene with corrective measures.

We can largely classify tool supported quality assessment methodologies into three groups: analysis methods, verification methods, and testing methods. Since verification and testing methods target exclusively functional aspects, and therefore external software quality, they are outside the scope of the present thesis.

This section is intended to provide an overview of the fundamental concepts and methods for assessing internal quality, based on analyses on the source code. We distinguish four categories of approaches.

The first category consists of approaches for checking compliance to coding styles. Coding styles deal with very simple, language specific problems, such as convention violations for naming, commenting, code layouting or the use of particular language constructs. Style conventions can be either universal, or system dependent. Examples include the Kernighan and Ritchie style for the C language[2], the GNU programmer's style guide [3], or the official Java coding conventions[4].

The second category are clone detection approaches, which aim at detecting repetitions of identical or close to identical blocks of source code. Clones are considered harmful, because each occurrence of a fragment must often be separately understood, changed and tested. For example, forgetting to fix one or more occurrences may lead to inconsistent behavior or runtime errors. Some examples of methods and tools for detecting clones are [LHMI07, JMSG07, LJ05, DRD99].

These two categories of approaches are limited in scope, and therefore have a relatively small contribution to forming an overall picture about the state of the design. The other two categories of approaches, presented in sections 2.3.3 and 2.3.4, are those based on metrics and structural pattern matching. But before taking a closer look at these, we have to discuss two important prerequisites to tool supported quality assessment: how the structure of systems can be represented, to allow its inspection and manipulation by tools, and how we can model the abstract and subjective notion of quality.

2.3.1. Structural Models and Meta–Models

We mentioned before that design documentation can take many textual and/or graphical forms. We also mentioned the fact that the source code can be regarded as an accurate and up-to-date design document. Day to day experience shows that because of time pressure, other forms of documentation are almost never in sync with the current state of the code. Even documentation that is automatically generated from the source code, with tools such as JAVADOC [Fri95], is not guaranteed to be 100% accurate. Therefore, almost all approaches for tool supported quality assessment rely exclusively on the source code of the program.

But the source code is also a very detailed description of the design, which means many thousands of files to be analyzed. Even with tools, performing complex analyses across so many files is very inefficient, much of the information being irrelevant from the viewpoint

[2] http://www.cs.usyd.edu.au/~scilect/tpop/handouts/Style.htm
[3] http://www.nongnu.org/style-guide/
[4] http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

of the analysis. In addition, the program may be written in a mix of various programming languages, which brings even more technical complications to the analysis. Because of these reasons, analyses are usually not implemented to operate directly on the source code, but instead on top of a more or less abstract model constructed from the static structure. Such a model is called a *structural model*, and is obtained as a result of a process called "fact extraction", typically followed by a combination of *aggregation* and *selection* operations (jointly called "abstraction") [Ciu01]. The level of abstraction is tailored to suit the specific needs of the employed analyses.

In general, any model describes entities and their relations, as they exist in a given domain. A meta–model is a model that describes the structure and semantics of a certain class of models. A given meta–model describes the elements and relations that appear in a particular class of models. It provides a precise definition of the constructs and rules needed for creating concrete models. In turn, models are said to be instances of their corresponding meta–model (see [vG91]).

2.3.2. Software Quality Models

The notion of software quality is very complex, because it is by nature subjective and extremely broad in scope. First of all, it is subjective because customer experience, perhaps more than mere compliance with the specifications, plays a decisive role in judging a product as having a high quality. Secondly, the notion is extremely broad because it can refer to anything from internal design, to the conformance with requirements, the development process and the usability of the system. Consequently it is impossible to quantify software quality directly.

Therefore, a *software quality model* is a collection of rules and procedures that allow making qualitative statements about a system, by creating a mapping between the abstract notion of quality on one hand, and objectively ascertainable properties of the system on the other hand.

In general, a quality model is either strictly hierarchical, or takes the form of a directed acyclic graph. The root nodes of the graph represent various aspects of quality, that are of interest to the assessors. Successive layers of the model represent iterative refinements of the root aspects, until we reach a level that is usually populated by directly ascertainable or measurable properties.

The first quality models structured in this way were the ones presented by McCall et al. [MRW77] and Boehm et al. [BBL76], followed by many others, such as [Dro96, EL96, HR96, BD02, Mar02].

As a result, the international organization for standardization (ISO) elaborated the ISO/IEC 9126 standard, which covers a strictly hierarchical decomposition of six views

on quality, referred to as *quality factors*: functionality, reliability, efficiency, maintainability, portability and usability. Each quality factor is further decomposed into subcharacteristics, called *quality criteria*. In the case of maintainability, the ISO standard discerns between analyzability, changeability, stability and testability.

In spite of its popularity and success, the ISO standard does not say anything below the level of quality criteria. In other words, the mapping process, called *operationalization*, between the intangible characteristics of quality and the tangible properties of the code [LCP01] (e.g. code metrics), is not covered. However, there are several methods that can be used for the operationalization of intangible quality criteria, such as: the Quality Function Deployment approach [KA83], the Goal-Question-Metric approach [BCR94], and the Software Quality Metrics approach [BBL76].

Introduced by Basili in [Bas92], the better known "Goal-Question-Metric" (GQM) approach was originally defined for evaluating defects for a set of projects in the NASA Goddard Space Flight Center. It is designed to facilitate the operationalization with the help of code metrics, but can be easily adapted to accommodate structural patterns. The method consists of six steps, as follows:

1. Determine the set of measurement goals for quality. These can be for example the quality criteria described by the ISO standard.
2. Generate questions that define those goals as completely as possible in a quantifiable way
3. Specify the code metrics that need to be collected in order to answer those questions
4. Develop mechanisms for data collection
5. Collect, validate and analyze the data in real time to provide feedback for corrective action
6. Analyze the data in a postmortem fashion to assess conformance to the goals and to make recommendations for future improvements

2.3.3. Analyses Based on Metrics

In any engineering discipline, measurement is essential, because "you cannot control what you cannot measure" [DeM82]. In general, the term "measurement" is defined as "the process by which numbers or symbols are assigned to attributes of entities in the real world in such way as to describe them according to clearly defined rules" [FP96].

A *measurement theory* establishes the rules to be followed in order to be consistent in the measurement activity, and provides a foundation for interpreting measurement results [FP96]. It involves a rigorous, mathematical description of scales, measures and methods

of measuring [HS96]. There are several measurement theories, but the so called representational theory is the most common. The most important requirement imposed by the representational theory is that the mapping of entities and empirical relations into numbers and numerical relations must be of such a nature, that the empirical relations preserve and are preserved by the numerical relations.

In the context of software, IEEE standard 1061 (1998) describing a software quality metrics methodology, defines the term *software quality metric* as "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality".

Generally, software metrics are classified into *product metrics*, which describe the software system itself, and *process metrics*, which describe progress, resource efficiency and other managerial aspects of the development process.

A metrics suite is a set of individual metrics, intended to allow the formation of an overall picture on the state of the object of measurement, by shedding light onto various complementary aspects of its quality. Well known suites of object oriented product metrics are those of Chidamber and Kemerer [CK94] and Lorenz and Kidd [LK94]. For a detailed overview of these and other metrics suites, we refer the reader to [HS96].

Despite their relatively fast and effortless extraction, the effective use of software metrics poses some problems:

- Metrics operate on a relatively low level of abstraction, by measuring things such as method complexity, class cohesion and coupling, and depth of inheritance trees. Thus, the use of design patterns for example, often has a detrimental effect on certain metrics values. This is because a high quality design is always a compromise between different aspects of the structure, as measured by individual metrics.
- Metrics are generally symptomatic. They measure properties which may indicate the existence of problems, but which are difficult to put in correspondence with meaningful corrective measures.
- It is difficult to identify universally meaningful thresholds for each metric, as they may depend on the nature, size or other particularities of the system. Thus, measurement results must often be interpreted in the context of the current project.
- The validation of metrics is difficult, therefore it is hard to guarantee an acceptable level of confidence in their predictive abilities. The risk is especially high, when employing a quality model in order to aggregate several metric values, into a global quality index value, or fitness function.

In spite of the above difficulties, metrics form the basis for a large percentage of tool supported analysis methods. Two important groups of approaches are worth mentioning here.

The first group of approaches aim to express the qualitative principles and rules which govern object oriented design, in terms of sets of software metrics. The declared purpose of these methods is to make design rules quantifiable and thus suitable for automated problem detection. One of the first attempts was made by Erni and Lewerentz in [EL96], through the definition of "multi-metrics" as n-tuples of metrics that capture a given quality criterion, as described by a quality model. A more advanced method is based on the notion of "detection strategy", introduced by Marinescu in [Mar02, Mar04]. Detection strategies are metrics based logical expressions that embody negations of well known design rules. Compared to multi-metrics, detection strategies provide better abstraction and encapsulation capabilities, and allow an engineer to directly locate design entities that are affected by structural anomalies, such as "god class", "data class", and "feature envy".

The second group of approaches is represented by metrics based software visualization techniques, whose primary purpose is to support reverse engineering activities, but which can nevertheless be effectively used in problem detection as well. Particularly interesting in this context, are the so called "polymetric views" [LD03], which are ordinary visualizations, enriched with metrics information. In particular, design entities are represented using rectangles and relations are represented using edges. The essential visual properties of these graphical elements, such as position, size and color, are varied depending on the values of a carefully chosen set of metrics. Particularly useful examples of polymetric views are the "system complexity" view [Lan03], and the "class blueprint" [LD01]. Similar metrics enriched visualizations have been attempted in three dimensions. Examples include [LD05] and [LN03].

2.3.4. Analyses Based on Structural Pattern Matching

In section 2.1, we defined the notion of structure as the set of all statically observable entities, their properties, and relations. As a consequence, the structure can be mathematically represented as a graph, in which the observed entities are nodes, and the observed relations are edges.

In general terms, a *structural pattern* is a graph that contains all entities and relations that are common to a class of structural fragments of interest. Given the graph that represents the structure of a system, *structural pattern matching* is the process of finding subgraphs (i.e. pattern instances) that possess all of the constituents of a given structural pattern. In mathematical terms, a match is a graph homomorphism between the pattern graph and the host graph.

Depending on the implementation details of the pattern matching process, we have graph based and rule based structural pattern matching approaches.

Graph based matching, as implemented in *graph rewriting systems* such as [BS99, ERT99, NSW+02, GBG+06], employs a graph formalism to specify the pattern, which then constitutes the subject of a systematic search in a host graph. Since graph matching is an NP-complete problem, performance, but also memory efficiency constitute critical concerns. In the general case, graph matching algorithms and the analyses based on such algorithms scale exponentially with the size of the system model, and therefore have a limited applicability for large software systems.

On the other hand, rule based structural pattern matching works by defining a set of rules as relations over elements of the structural model. This approach proves to be more flexible in constructing ad-hoc analyses on large systems, because of the relative ease with which rules that are defined on extremely different abstraction levels, can be defined and combined into more complex analyses. In addition, a rule based formalism naturally fits the form in which principles and rules of design are formulated. Rule based pattern matching has been used successfully in design pattern detection for understanding and redocumentation [KP96, AFC98, HHHL03], architecture checks using reflection models [SSC96, FKvO98, MW99], and detection of structural anomalies [Ciu99, Ciu01, BNL03].

For example, in the context of restructuring, Ciupke describes [Ciu99] an automated method for detecting structural anomalies in object oriented code. He defines the term "structural problem" (i.e. structural anomaly, according to our terminology) as any property of a structure fragment, which negatively affects development or maintenance costs. The structural model is constructed directly from the source code, and takes the form of a PROLOG fact database. In order to detect structural anomalies, Ciupke turns to heuristic rules formulated in [JF88, Rie96, Lak96], such as "all data should be hidden within its class", or "a class should not depend on any of its subclasses". He then expresses the violation of each rule in the form of a PROLOG query, whose execution yields a set of candidates. Each candidate anomaly is validated by the software engineer through manual inspection.

2.4. Tool Supported Program Transformation

The field of program transformation comprises a number of basic techniques, whose purpose is to change existing programs in a useful way, according to a set of well defined rules. Transformations normally preserve the semantics of the program, but there may be applications where this is not required, or even wanted [PP96, Vis05]. Program transformation techniques and tools have varying degrees of specialization, ranging from single-purpose normalization based on simple pattern matching, to general purpose full fledged meta-programming systems. In the following, we give a brief overview of the most relevant approaches to program transformation, from the viewpoint of our work: normalization, refactorings, and general purpose meta-programming.

2.4.1. Normalization Approaches

Normalization approaches adapt an existing program, according to a specific purpose, by limiting the employed language constructs to a subset of the ones allowed by the programming language. Normalization may be implemented either directly on the source code, via textual pattern matching, or on a structural model of the system, using model transformations or graph rewriting systems.

Simple normalization techniques are for example the elimination of goto statements, desugaring[5], and replacing "if-else" constructs with "switch-case" statements. More complex normalization operations are for example the implementation of the Demeter normal form [Lie96] for improving modularity, normalization of inheritance hierarchies according to [Cas92, Cas94], and the elimination of specialization-related problems in inheritance hierarchies [Neu00].

2.4.2. Refactorings and Derived Approaches

The term *refactoring* was introduced by Opdyke in [Opd92], to denote a relatively low level program transformation, meant to change the structure of the code, without affecting behavior. Behavior preservation is understood in functional terms, in the sense that given the same set of inputs, the transformed program generates the same outputs as the original. Each refactoring has a set of pre-conditions. Behavior preservation of a refactoring is only guaranteed if all pre-conditions of that refactoring are met, prior to its application. Example refactorings introduced by Opdyke include basic ones, such as "move variable" or "create empty method", and composite refactorings, such as "replace inheritance with delegation", obtained by chaining together several basic refactorings.

Opdyke's original concept of composite refactoring required that each basic refactoring in its composition must be behavior preserving. In [Rob99], Roberts eliminated this limitation, by extending each refactoring with a set of post-conditions that describe the effects of the transformation on the program. He also described a composition method which makes it possible to derive the global pre- and post-conditions of a composite refactoring, based on the individual pre- and post-conditions of the composed refactorings. Thus, it is possible to guarantee behavior preservation of a composite refactoring, even if its component refactorings are not individually behavior preserving.

Refactorings such as those described in [Fow99] have become an essential ingredient in agile development processes, and enjoys an ever increasing support in modern development environments, such as Together[6], IDEA[7] and Eclipse[8]. They are typ-

[5] syntactic simplification, through the elimination of redundant language constructs
[6] http://www.borland.com/us/products/together/index.html
[7] http://www.jetbrains.com/idea/
[8] http://www.eclipse.org/

ically implemented by parsing the code into a model on the level of the program's abstract syntax tree representation, transforming this model, and finally regenerating the code from the transformed model (i.e. un-parsing). Furthermore, a series of works [Zim97, SGMZ98, Cin00, Gen04] developed the concepts and tools for the automated introduction of design patterns [GHJV96], through a sequence of refactorings. More recently, the idea was picked up again by Kerievsky, who elaborated a catalogue of "refactorings to patterns" in [Ker05].

2.4.3. Meta-Programming Approaches

Meta-programming, more exactly static meta-programming, represents the most general form of program transformation. As shown in [Lud02], it constitutes a theoretical foundation and technical basis for tool support, for all other program transformation techniques. In particular, meta-programming can be used for carrying out transformations that preserve neither behavior nor semantics of program. This is for example the case in invasive software adaptation scenarios, such as those described by [Gen04].

The idea behind static meta-programming is to parse programs into a tree or graph based model, manipulate this model using term or graph rewriting techniques, and finally unparse the program back into its original format (typically source code). A distinction can thereby be made, between the program being transformed, and the meta-program which drives the transformation. In order to allow general source to source transformations, the employed structural model usually corresponds to the meta-model of the programming language itself. The RECODER[9] framework provides such an infrastructure for the Java programming language, and supports the development of further meta-programming tools.

An example of a meta-programming tool based on the RECODER framework is Inject/J [Gen04], which supports all refactorings described in [Fow99]. The tool also provides a scripting language that provides advanced navigation capabilities, and allows the definition of composite refactorings. Furthermore, the tool is able to automatically test pre- and post-conditions of all applied transformations.

[9] http://recoder.sourceforge.net/

Chapter 3.

Related Work

The previous chapter is dedicated to the introduction of basic terminology, as well as the description of the foundations upon which our method is built. In particular, practical approaches discussed in the previous chapter focus on sets of problems that are specific either exclusively to code analysis and quality assessment, or exclusively to source code transformation.

However, in order to perform restructuring, we require a holistic approach that successfully combines methods and techniques that are specific for both quality assessment and source code transformation. The present chapter provides an overview of a series of attempts to bridge the gap that exists between the two categories of approaches. Each individual attempt is examined critically, with respect to the set of criteria defined in section 1.1.3.

For reasons of clarity, we decided to classify related approaches, based on common characteristics, into five disjoint groups:

Manual approaches: are represented by works that describe elements of, or entire restructuring scenarios, in a textual, usually pattern-like form. Scenarios follow the classic decomposition of the restructuring process into problem detection, problem analysis and reorganization. A pattern typically describes a starting structure considered as deficient, possible causes for its appearance, one or more desirable structures that can replace the existing one, and an example. Optionally, anecdotes, known exceptions or variations of the described situation may be given. No mechanism is provided that would allow an automated detection of the initial situation, or that would support the decision for one or the other of the alternative desirable structures.

Tool supported investigative approaches: are attempts to directly automate approaches from the previous category, by providing a mechanism for the automated detection of the deficient structure. Problem analysis is only indirectly supported with tools, if at all. The decision process employed by the maintainer is not *systematic*, because it has an ad-hoc nature, relies heavily on his intuition and

experience. The term "investigative" expresses the fact that each detected problem instance requires extensive manual investigation, while relying on the maintainer's personal skills and the literature. Automated code transformation capabilities are sometimes provided.

Tool supported regressive approaches: are approaches that avoid the problem analysis step altogether, by starting off from a given target structure, and looking for places in the code where this structure is desirable, but is either absent, or present in a distorted form. The target structures correspond to the well known design patterns. Thus, structural anomalies such as the common code smells are generally not addressed by these approaches. Some degree of tool support for automated detection as well as code transformations is provided.

Automated specialized approaches: are approaches offering a high level of automation, but limited in scope to optimizing a narrow aspect of a system's design, such as the minimization of duplication, or the optimization of the modularity of subsystem decompositions.

Automated search based approaches: are approaches that employ a search based mechanism in order to optimize the structural quality of a system. The scope is much larger than in the case of the previous category. Structural quality is expressed as an abstract cost or fitness function. The cost function employs a set of code metrics such as coupling, cohesion and complexity metrics. Tool support is generally limited to providing the automated computation of the structure that is optimal with respect to the defined cost function.

3.1. Manual Approaches

Based on the idea of patterns, Brown, Malveau, McCormick and Mowbray condense in [BMMM98] a number of common mistakes, in a wide spectrum of fields that range from software project management to architectural issues and coding practice. These common mistakes, called anti-patterns, are accompanied by a discussion of possible causes and recommended remedies, as well as examples and anecdotes. The book captures a vast body of experience in spotting and defusing potentially dangerous software development pitfalls, and presents it in an informal and entertaining way.

From the viewpoint of tool supported object oriented restructuring, anti-patterns have a number of weaknesses. Having an extremely broad scope, anti-patterns are rather abstract, keeping a certain distance from code structure, and instead focusing more on the process and organizational aspects of how mistakes get to be made. As a result, the recommended remedies are most often very general and refer to organizational and process improvement measures, rather than concrete code transformations. The *lack of systematic procedures* means a high degree of reliance upon the skills and experience of the

maintainer, which in turn prevents attaining a level of *automation* that would make the method practicable for large industrial systems.

Based on the previous work of Opdyke [Opd92] and Roberts [Rob99], Fowler discusses in [Fow99] the principles and use of refactorings as a means to improve the design of existing code, and presents a catalogue of more than 70 refactorings. A refactoring describes the mechanics of a behavior preserving code transformation. Triggered by the question of when to use refactorings, he goes on to describe a number of structural anomalies, referred to as "bad smells in code". These act as alarm signals that should attract the attention of the maintainer to the respective design fragments.

The book contains short discussions of possible refactorings that might be meaningful in the presence of each smell, but doesn't offer any *systematic* procedure that would help the maintainer determine the most appropriate sequence of refactorings in any given situation. Indeed, this is not possible, because of the symptomatic nature of code smells, which merely signal the presence of some problems, but a *causal* treatment of these problems requires further investigation (i.e. problem analysis) by the maintainer. As a result, the level of attainable *automation* is in principle limited to detection of code smell instances and the implementation of code refactorings.

In addition, a method that is based exclusively on code smells is oblivious to incipient problems, because it only addresses situations where there is already a level of damage that has become so significant as to induce the anomalous structural characteristics, captured by typical smells. In particular, as shown in chapter 1, the appropriateness of a structural construct depends on the design context of the fragment, which includes the change potential of that fragment. In other words, the structure may be inappropriate, if it does not accommodate future changes in a natural way, even if there are currently no anomalous characteristics, as captured by code smells. Code smells tend to describe the result of ignoring the incipient problem, over a longer period.

In [DDN03], Demeyer, Ducasse and Nierstrasz apply the pattern format in order to document best practices in the general context of object oriented reengineering. Their work is based on the earlier results of the FAMOOS research project, which are described in [Bae99]. There are nine clusters of related patterns, each focusing on a specific type of reengineering activity, such as initial understanding of the system, migration strategies or redistributing the responsibilities of classes. Reengineering pattern descriptions span across a wide range of abstraction levels, from aspects of the reengineering process, to code structure. Most of the reengineering patterns focus on the reengineering process itself rather than the design of the system (e.g. "speculate about design", "involve the users", "use a testing framework").

Those clusters that are relevant from an object oriented restructuring standpoint, deal exclusively with redistributing responsibility and transforming conditionals to polymorphism. Thus, reengineering patterns do not satisfy our *comprehensiveness* criterion. The

recommended structure does not take into account the entire design context because it ignores the changes that are expected to occur to the respective fragment. Furthermore, some of the reengineering patterns prove to be symptomatic, in the sense that there are several patterns that deal with facets of the same problem (e.g. "move behavior close to data" and "eliminate navigation code") and others that mix more than one kind of problem (e.g. "split up god class", which is a mixture between class schizophrenia[1] and a deficient distribution of responsibilities[2]). For these reasons, the *causality* and *systematic process* criteria are not satisfied. In addition, the authors do not provide any mechanism that would allow a tool supported detecting of the places in the code where a restructuring pattern would make sense. Thus, the reengineering patterns cannot be employed in an *automated* fashion.

The last approach in this category corresponds to the so called "refactorings to patterns", described by Kerievsky in [Ker05]. They are based on the idea enounced by Schulz, Genssler, Mohr and Zimmer in [SGMZ98], of applying refactorings to automatically introduce design patterns in existing code. In his book, Kerievsky presents a catalogue of such meta-refactorings. A meta-refactoring is organized in a pattern format, the focus being placed on the mechanics of implementing the corresponding design pattern.

Although each meta-refactoring contains a brief description of the initial situation, no method is provided that would allow an *automated* detection of these situations. Furthermore, the mapping between the described starting situations and design patterns is not always univocal, thus requiring the maintainer to decide between alternative solutions. For example, the meta-refactoring "replace conditional logic with strategy" discusses the strategy design pattern and the creation of subtypes as alternative solutions to the same problem. This means that the approach does not satisfy our *systematic process* criterion, and *causality* is not intrinsically guaranteed by the method, but left as a responsibility of the maintainer. Finally, the approach focuses exclusively on design patterns, as solutions to structural anomalies. As a result, lower level structural anomalies, such as those indicated by the presence of certain code smells (e.g. data class, feature envy, refused bequest), are not covered at all. Thus, the approach fails short of the requirements of our *comprehensiveness* criterion.

3.2. Tool Supported Investigative Approaches

Shortly after the introduction of the concept of code smells in [Fow99], first attempts to automate their detection in code appeared: [Ciu99, Ciu01, Mar01, Mar02]. On the other hand, based on the works of Roberts [Rob99], Cinneide [Cin00] and others, automated

[1] see design flaw "schizophrenic class" in A.6
[2] see design flaw "misplaced control" in A.5

refactoring support in commonly used development environments was becoming more and more common. In this context, there have been numerous attempts to link automated code smell detection with automated refactoring in integrated approaches, which are discussed in the following.

In [DW03], the authors present a tool for restructuring Java systems. The tool uses heuristic rules expressed in OCL to automatically detect code smells on an abstract syntax tree representation of the source code. Refactorings are implemented as transformations of the abstract syntax tree model, which allows code generation, using a dedicated pretty-printer. For every code smell that is detected, the tool suggests a number of applicable refactorings, but it is the responsibility of the maintainer to choose those that are best suited in each specific context (i.e. lack of *systematic procedures* in the restructuring process). Thus, the approach cannot guarantee a *causal* treatment of the underlying problem.

A very similar approach, but relying on logic meta-programming for code smell detection, is used by Tourwé and Mens in [TM03]. The authors acknowledge that often, several refactorings can be chosen to remedy a particular situation (i.e. a code smell instance), and "it is impossible to infer automatically which of these refactorings is most appropriate". Thus, a list of possible refactorings is provided, and the maintainer must choose those that are appropriate in each particular case, by performing problem analysis manually. The authors also talk about cascaded refactorings. It refers to situations in which carrying out certain refactorings may open the possibility for applying other refactorings. However, the method remains symptomatic, because it only provides for investigating the *possibility*, but not the *opportunity*, of the cascading refactorings.

An interesting, more recent approach, is the one using the metaphor of "harmonious design", described by Lanza and Marinescu in [LM06]. The work is based on, and combines previous results ([Lan03] and [Mar02]) of the authors. According to them, a harmonious design must give consideration to three aspects: class identity (what defines a class), object collaboration (how do instances of unrelated classes cooperate) and classification (how do classes in a common hierarchy relate to one another). For each of the three, the authors use the literature in order to distill a number of heuristic "rules of harmony" that describe desirable characteristics of design. They then define "disharmonies" as violations of these rules, and provide two mechanisms that are shown to be extremely effective in detecting disharmonies: detection strategies [Mar04] and polymetric views [Lan03].

From a restructuring standpoint, disharmonies are very similar to code smells, and assume all of their shortcomings. As in other approaches, problem analysis is not *systematic*. Although the possible decision paths are described, the maintainer does not get any support in making these decisions. Furthermore, the notion of software quality is absolute, in the sense that future changes are not taken into consideration. Because of these

reasons, the *causal* treatment of structural anomalies is not built into the method, and thus cannot be guaranteed.

Other approaches that follow similar strategies, and show similar weaknesses to the ones described above, are [GC03, Fre03, Tri06, Mey06].

The approach presented in [TK04] uses software metrics to assess the quality of a system based on a so called soft-goal graph decomposition. Soft-goals characterize various aspects of quality on a very abstract level, in terms of aspects such as modularity, coupling, and complexity. Soft-goals are then operationalized, by putting them in correspondence with a selected set of code metrics. The authors also define a basic set of abstract, high level transformations (called meta-transformations), and formalize their corresponding impact on the selected metrics, and consequently on the set of soft-goals. Based on this impact estimate, a number of such transformations can be recommended to the engineer, to improve system quality in certain places that scored poorly against one or more of the soft-goals.

The approach is focused exclusively on the class level, and therefore does not satisfy our *comprehensiveness* criterion. As suggested by their name, soft-goals are deeply symptomatic, because they capture general properties of certain design elements or groups of such elements, such as modularity, coupling and cohesion. Furthermore, the intent of a given design fragment, as well as the expected changes are not taken into account when proposing transformations. Thus, the method cannot guarantee a *causal* treatment of problems arising in the structure.

In [MBG06], Moha, Bouden and Guéhéneuc propose a systematic process for specifying and detecting structural anomalies, referred to as "design defects". Defects are classified according to their granularity into higher level (e.g. anti-patterns) and lower level defects (e.g. code smells). The authors recognize that lower level defects can act as symptoms of a higher level defect (e.g. the blob induces a central large class, surrounded by several data classes). They introduce the notion of rule card, which is a formal description of a high level defect, and contains structural and semantic information, and is backed by a special meta-model. The structural information consists of the formal metric-based specification of the lower level defects, while the semantic information consists in the explicit definition of a number of abstract roles, which various entities fulfill within an instance of the respective high level defect. Thus, a rule card is in effect equivalent to a semantic network. Some of the main ideas are picked up and presented in more detail in [MGL06]. The paper also provides some preliminary experimental results, but a full validation is missing. In a subsequent paper [MRG+06], the authors realize the juncture with tool supported correction with the help of formal concept analysis. A full validation is not provided.

From the standpoint of our success criteria, this approach too has some of the typical drawbacks in this category. Most importantly, the approach fails to guarantee a causal

treatment of design defects, because of the following three reasons. First, defects that are detected are inherently very abstract (e.g. spaghetti code, functional decomposition), shedding little or no light on how to defuse them. Second, in order to come to a recommended structure, the method relies on formal concept analysis and is in effect a clustering technique. In the case of the blob for example, the redistribution of methods between classes is based on coupling and cohesion measures, ignoring design intent as well as the change potential of the design fragment. Thirdly, *semantics*, though often advertised, is confined to the roles that entities have (e.g. in the case of the blob, the central class and the surrounding data classes). In particular, the design intent of the analyzed fragment is not taken into account.

Finally, although the approach is claimed to be systematic, it is systematic on a meta-level, in the sense that a sequence of steps is described in great detail. However, it does not describe the problem solution mapping process *systematically*, and thus cannot guarantee repeatable results assuming a non-expert maintainer.

3.3. Tool Supported Regressive Approaches

In [GAA01], Guéhéneuc and Albin-Amiot use formal descriptions of design pattern structures in combination with a constraint solver, with automatic constraint relaxation, in order to detect fragments that approximately match the specified structure. The degree of variance is kept under control using parameters to the constraint solver. Their method assumes that the found fragments represent distorted instances of the well known design patterns. Thus, the source code is automatically transformed, based on a set of predefined transformation rules, and in accordance with the structural differences to the canonical structure of these patterns.

From the standpoint of our criteria, this approach has a number of drawbacks. First of all, it is not clear why the use of a slightly different structure than the standard canonical design pattern structure is a defect. Second, the method has a symptomatic nature, because it ignores the design context of the identified “defects”. Thus, the corresponding design pattern or its specified canonical structure may not be appropriate in each concrete situation. Indeed, the authors themselves admit that "we need a repository of good designs, independent of the context, for reference". But we argue that there is no universally good structure, not even the canonical structure of design patterns. Finally the method is limited in scope to defects that map easily onto the well known design patterns.

In [JLB02], Jeon, Lee and Bae propose a tool supported approach to find candidate spots for design patterns, which also uses the modification history of the program, as given by the deltas between two or more complete source trees (i.e. releases). The assumption, which is a reasonable one, is that parts that suffer certain modifications frequently, benefit

the most from employing design patterns. The detection mechanism is query based, and uses an abstract structural model, expressed in the form of a PROLOG fact database. The model integrates both static structural information with historical information.

The approach is exemplified on the case of creational patterns, but no experimental results are provided. In particular, a validation of the method for more complex design patterns is desirable. Furthermore, the approach does not satisfy the *causality* criterion, because it dos not take potential future changes into account. Finally, like the rest of the approaches in this category, the scope is limited to structural problems, whose solution consists in applying design patterns.

Rajesh and Janakiram report on first experimental results obtained with their tool JIAD [RJ04]. The authors claim to be able to automatically infer which design pattern is applicable where, in the code of a Java system. The tool uses Prolog-like queries on a fact data base, extracted from the source code. The specification of detection rules are based on the intent and applicability sections of the classic design pattern descriptions.

Detection rules are symptomatic, because they are rigid and use structural information exclusively. In particular, design intent and potential changes are not taken into account. The authors themselves admit that "sometimes design patterns introduce complexity. Therefore, before applying the transformation process, the impact [...] on quality attributes needs to be estimated". Like the other approaches discussed above, the scope is limited to finding opportunities to implement a design pattern. Although design patterns arguably represent high-quality designs *in certain contexts*, they represent rather high-level solutions to high-level problems. In particular, code smells may indicate structural problems that are outside the scope of design patterns.

3.4. Automated Specialized Approaches

These approaches are designed to address one particular type of structural problems, without consideration for others, hence the adjective "specialized". Thus, no single specialized method can meet our *comprehensiveness* criterion. The reverse of the coin is that these approaches are able to attain relatively high levels of *automation*. Furthermore, though less symptomatic than search based approaches, specialized approaches do not generally guarantee that recommended transformations are *causal*, because design intent and potential future changes to the subject fragments are ignored. The following paragraphs provide a few emblematic examples out of an otherwise very large array of works.

In [Cas92, Cas94], Casais describes an iterative method for improving class hierarchies, which employs two kinds of reorganizations. The so called decomposition consists in the separation of the various abstraction steps that are usually merged in a single inheritance

relation, such as specialization, implementation reuse or extension. Through this operation, alternative modeling possibilities are detected, which contribute to the reduction of the semantic overloading of inheritance links. The second operation, called factorization, extracts class members shared by several classes and isolates them in a common ancestor. This is meant to eliminate unwanted redefinitions of class members.

Moore describes a tool called Guru [Moo96], to automatically create inheritance hierarchies that avoid member and code duplications, out of existing hierarchies and/or unrelated classes. The tool operates on programs written in the programming language Self.

The method presented by Neumann in [Neu00] is aimed at systematically treating problems that result from the violation of the Liskov substitution principle [LW93] in inheritance relations. Such situations are unavoidable in certain types of applications, especially in the development of framework based applications, business process modeling and certain algorithms and data structures. To this end, the author defines a set of correctness criteria, which also take into account how the respective subclass members are used. For the situations in which a transformation of the hierarchy is not possible, a systematic method for recognizing and interactively eliminating invalid usage patterns is provided.

The Demeter method for adaptive software development [Lie96] is a method that minimizes the coupling between groups of classes that cooperate in the realization of the system's functions. The so called law of Demeter is a principle that requires that objects should only have knowledge about other, directly related ones. The author describes a method for transforming the structure of a system, so that it satisfies the law of Demeter.

Finally, a notable group of works deal with the automated computation of subsystem decompositions, for systems whose subsystem structure is either lost or in need of optimization. Also, computing subsystem decompositions based on coupling and cohesion metrics may provide useful insights during reverse engineering of a large system. A subsystem is generally defined as a group of connected components that exhibits a relatively high cohesion, and low coupling with respect to other subsystems. In the case of object oriented systems, the most used techniques are based on clustering methods. Clustering, or cluster analysis is a mathematical technique for grouping entities based on their common characteristics [Har75]. Successful use of clustering techniques for computing subsystem decompositions in object oriented systems has been reported in [RRHK00, EPS00, Tri01, MM01, Mit02].

In [BT04], Bauer and Trifu present an improved clustering method that relies on structural pattern matching, to detect common architectural patterns that are taken into account during clustering. For example, their method is able to identify and treat library code as a cohesive cluster, and avoid its breakup and dispersal in other subsystems that use it. Thus, their method is able to produce decompositions of a substantially better quality, from the standpoint of the maintainer.

3.5. Automated Search Based Approaches

Harman in [HJ01], and later Clarke et al. in [JCS], give an overview of the application of search techniques to software engineering activities in general, and to restructuring in particular. Harman coins the term "search based software engineering" and recognizes it as a new emerging field in software engineering. Their motivation is based on the realization that software engineering activities can be viewed as a search for solutions that balance many competing constraints to achieve an optimal or near optimal result. Thus, the aim of search based software engineering research "is to move software engineering problems from human-based search to machine based search".

In order to make machine based search possible, the characteristics to be optimized are expressed as a cost or fitness function, whose basic atoms are software metrics. Because of their exclusive reliance on metrics, all automated search based approaches to restructuring have the following fundamental weaknesses:

- Because of the metrics used, they tend to optimize coarser grained, higher level structural constructs such as design patterns away, because they operate on a much lower level of abstraction. For example, the use of certain design patterns (e.g. visitor, facade) may increase coupling metrics between subsystems. In fact, most design patterns represent solutions that trade some aspect of quality for others, such that there's an overall benefit concerning understandability and flexibility.
- They are generally symptomatic, because there is no case-to-case justification to the resulting transformations. In other words, restructuring decisions are not based on the local context, but rather on the value of the global cost function. In addition, the cost function usually aggregates several conflicting metrics that measure completely unrelated characteristics of the structure. Because of these reasons, the resulting structure may end up being much harder to understand and change, in spite of the improved metrics values.
- They operate with an incomplete, or static notion of software quality. According to the open-closed principle discussed in the previous chapter, the quality of a particular structure also depends on the type of change that the structure is likely to suffer in the future. Search based restructuring does not take this component into account.

Let us now look at a few examples in more detail.

The first attempts to apply search based techniques in the context of general restructuring (i.e. not limited in scope to a single aspect, such as subsystem decompositions) were focused on ways to automatically propose code refactorings. In [MSG99, SGM00], the authors propose such a method, that employs a simple quality model to estimate the impact of simple refactorings on maintainability. The quality model uses common coupling and

inheritance metrics. Detection of problematic fragments occurs by looking for anomalous values of the same set of metrics. Because it recommends refactorings based on a universal set of a few metrics, the method is inherently *symptomatic*, and cannot guarantee meaningful reorganization measures with respect to each fragment's particular intent and change potential.

Ó Keeffe and Ó Cinnéide present an automated method that relies on three types of search techniques to automatically transform Java programs [KC06]. The tool operates on an AST representation of the program and uses a hierarchical quality model based on 11 individual metrics. The employed search techniques are first-ascent hill climbing, steepest-ascent hill climbing and low temperature simulated annealing. The six implemented transformations are limited in scope to inheritance hierarchies, and work on the level of granularity of class members and above.

The authors argue that previous approaches "have focussed on improving one particular aspect of design, such as method reuse or code factorization. However, since object-oriented design involves numerous trade-offs, this narrow focus could result in overall quality loss". We argue that too wide a focus isn't helpful either. A cost function that incorporates many different aspects becomes too abstract. A method that tries to find an equilibrium between a multitude of antagonist forces loses touch with the local design context of each analyzed fragment. Thus, transformations are *symptomatic* because they lead to an "optimal" but not necessarily meaningful structure. In addition, the approach does not address fine-grained structural anomalies, such as those indicated by most of the well known code smells.

Another optimization approach for structural improvement, based on a genetic algorithm, can be found in [SSB06, Sen07]. The merit of the approach consists in trying to counter one of the weaknesses described above, by augmenting traditional search based techniques with a structural pattern matching technique, called "architectural clues" [BT04]. Architectural clues allow guessing the roles that methods play inside a class. For example, a class that only consists of delegating methods is probable to constitute a facade [GHJV96], and is therefore not broken up by the optimization algorithm. In other words, the method avoids destroying some design patterns, in the hope of preserving understandability.

On the other hand, this feature is also a weakness of the approach. As discussed in the previous chapter, the presence of a design pattern does not guarantee a high quality of the structure. A given design pattern has a specific intent, which only makes it meaningful in a well determined context. In cases where the design context of the fragment doesn't match the intent of the pattern, its presence constitutes a flaw, that the approach fails to address.

Furthermore, unlike the previous approach, the approach proposed by Seng does not attempt to detect design deficiencies in the structure. Instead, the structure is simply op-

Evaluation criteria	*Related approaches*				
	Manual	*Investigative*	*Regressive*	*Specialized*	*Search based*
Comprehensive	+	+	−	−	o
Causal	o	o	−	o	−
Systematic	−	−	+	+	+
Automated	−	o	+	+	+

Figure 3.1.: Condensed assessment of related work

timized with respect to the cost function that is defined. Thus, *causality* of the resulting transformations cannot be guaranteed.

A further weakness of the approach is its limitation in scope. For example, transformations do not reach into method bodies, and method moves between inheritance hierarchies are not supported. Because of these limitations, the approach fails to comply with our *comprehensiveness* criterion.

3.6. Conclusions

This chapter gave an overview of research that is aimed at bridging the conceptual divide as well as the gap in tool coverage, between current quality assessment based on static analysis, and code transformation techniques. Based on their commonalities, we identified five groups of approaches, and discussed representative examples in each group. Furthermore, we highlighted the most important weaknesses of each approach, by referring back to the four criteria, defined in section 1.1.3 of the introduction.

We found that there is a correlation between our classification of the approaches into groups, and their compliance with our criteria. This allows us to compile a condensed view of the entire assessment, as shown in figure 3.1.

As can be seen from the table, the five groups of approaches are rather complementary, and there is no category of approaches that fully complies with all our criteria. Another important conclusion is that none of the approaches fully complies with the causality criterion. The manual and investigative approaches tend to do better because they rely on human intervention, which makes it possible to come to a causal solution, in principle. The specialized approaches, particularly those that focus on inheritance relations, also tend to do a better job, because they are able to pay more attention to those aspects of design quality that form their focus. However, none of the approaches guarantees causality by construction.

Chapter 4.

Design Flaws

4.1. Design Context

As pointed out in chapter 2, software development is a process of elaborating a software based solution to a problem, in a given domain. A significant percentage of the activities involved in software development revolve around solving two distinct categories of problems: *modeling problems* and *structuring problems*.

Modeling is part of the design process, and it accounts for the following activities:

- Working out the system's functionality requirements and breaking them down into manageable chunks
- Establishing abstractions in the problem space, the solution space and the application domain, which form actors that participate in realizing the functional requirements of the application
- Assigning responsibilities to the various abstractions by distributing the chunks of functionality between actors

The resolution of modeling problems is a creative process that occurs on an abstract, mental level, resulting in an abstract mental model, which translates into a *design intent*. Thus, design intent is an abstract description of what needs to be achieved, in a given fragment of design. Design intent is technical in the sense that it does not describe functional requirements of the application, but rather states a design strategy to realize one or more such requirements. It is in fact a goal statement for subsequent structuring activities, and is comparable to the intent description of a design pattern [GHJV96].

In general, for any problem that needs to be solved in software, modeling may result in several possible design intents. Decisive factors are for example the level of granularity at which the domain, the problem and the solution are modeled, and the topology of the design, which can be either action oriented or object oriented, irrespective of the programming language used.

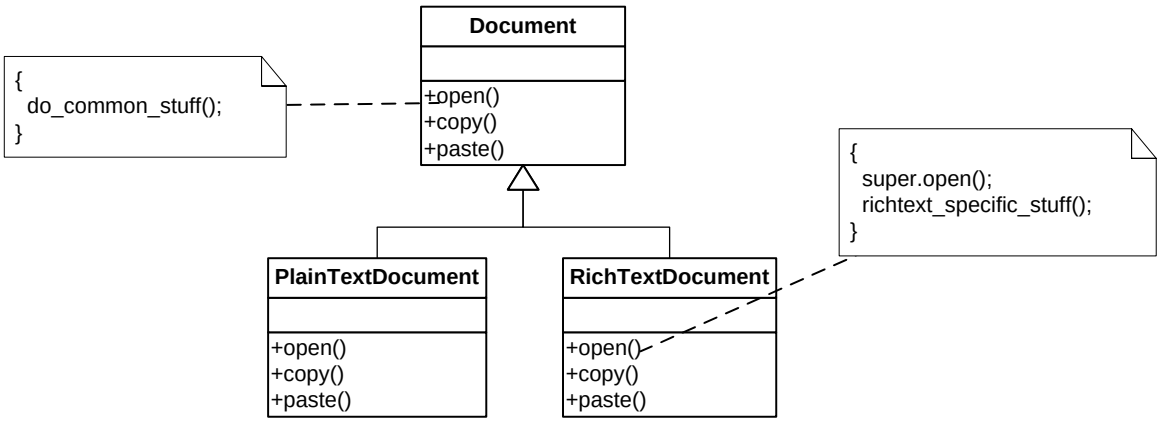


Figure 4.1.: Diagram showing a fragment of a text editor's design

After a design intent crystallizes, structuring is responsible for making it explicit, in consecutive iterations, using natural text, diagrams, and ultimately source code. Thus, the abstractions that result from modeling become for example classes, organized in hierarchies, whose assigned responsibilities are reflected in their public interfaces. The cooperation between objects may be governed by various design patterns, acting as a means of communication between developers, and from developers to maintainers.

During the initial design of a system, design intent is the result of modeling activities. In the course of a system's life, the original intent might change, as a result of changes to the design. Therefore, when we restructure a system, we must *infer* the currently existent design intent for a design fragment, by *understanding* the source code.

As an example, let's consider the case of a hypothetical text editor program that must handle plain text as well as rich text files. For such a program, we might have the design fragment depicted in figure 4.1. The design intent for the depicted fragment can be expressed briefly, as providing a specialization hierarchy for the `Document` abstraction. In other words, clients receive a uniform interface (i.e. `Document`) through which they are granted access to an abstract set of services (i.e. `open(), copy(), paste()`). The implementations of these services can vary transparently to the clients, depending on the actual type of document that is handled by the application at any given time. In the case depicted here, we are dealing with an object oriented design. Therefore, design intent is expressed in the structure with the help of type inheritance, a mechanism provided in all object oriented languages.

To use an analogy, the process of elaborating a software design resembles the process of elaborating and giving a talk. On one hand, modeling activities correspond to identifying and constructing the complex edifice of ideas, that the speaker wants to transmit to the audience. On the other hand, structuring corresponds to the process of expressing these

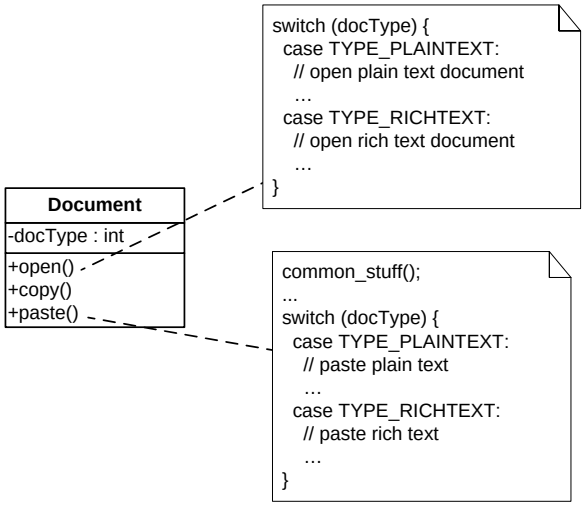


Figure 4.2.: An alternative structure for the `Document` class

ideas in sentences, in accordance with certain grammar and stylistic rules, particular to the language.

Just as an idea can be expressed in different ways, a design intent can be expressed using several possible structures. The decision depends on several factors:

- The principles, rules and dedicated mechanisms offered by the particular programming paradigm and language
- The types of changes that are expected to occur to the design
- Programming style and conventions, that may exist in the development team, which may sometimes contradict the principles and current best practices
- Various constraints imposed by the physical environment in which the system runs, such as performance, efficiency or costs associated to certain communication channels.

The first two factors are universal, while the last two strongly vary from team to team and from project to project. In the present work, we decided to limit the scope of the discussion to the first two of the four factors mentioned above.

Since nobody wants to touch a running system, all maintenance activities are the result of changes in the system's requirements or environment. These changes require changes to the design of the system. Making these changes requires understanding the existing design. Thus, in order to have a maintainable system, the source code must be structured in a way that favors understanding and making changes. But how do the first two factors mentioned above influence the ability of understanding and making changes to a system?

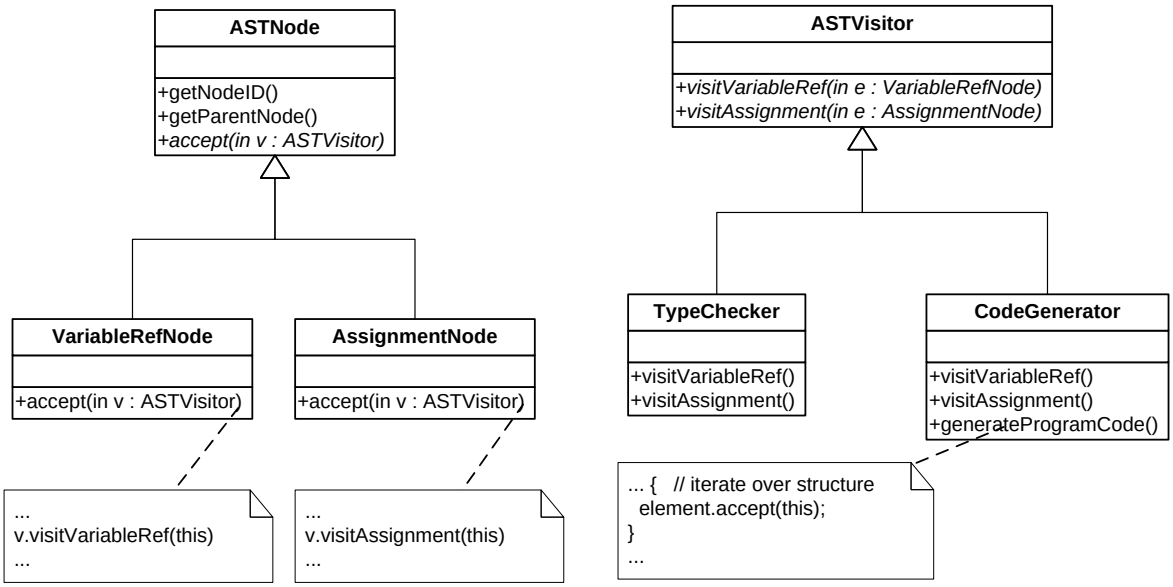


Figure 4.3.: A compiler's design, structured using the visitor design pattern

First of all, in every programming paradigm we have a set of best practices and rules, that guide developers in their choice of structure. In turn, programming languages provide paradigm specific mechanisms and constructs, that the implementer can use to express a design intent in a clear and natural way. Therefore, a structure that disregards the rules and best practices of the programming paradigm, by being an unnatural, surprising expression of the design intent, increases the effort of understanding the design.

For example, object oriented languages provide the mechanism of inheritance, as a means to express a specialization hierarchy. Moreover, the principles and rules of object oriented design [Mey88, CY91, Rie96, Mar00, Mar03] *recommend* the use of the inheritance mechanism to express *any* specialization hierarchy. For example, let us compare the structure of the design fragments shown in figures 4.1 and 4.2. Both structures express the same design intent (i.e. the specialization hierarchy of an abstraction called "document"). However, from the viewpoint of object oriented design, the structure shown in figure 4.1 conveys this intent more naturally and clearly, and is therefore recommended for the given design intent.

Let's now turn to the second factor that influences choice of structure: the expected changes. The open closed principle [Mar03] tells us that a program should be open for extension but closed for modification. Ideally, a fragment of design that conforms to the OCP can be extended without modifications to the existing code. Unfortunately, ideal conformance to OCP is impossible. Robert Martin notes in [Mar03]: "In general, no matter how «closed» a module is, there will always be some kind of change against which it is

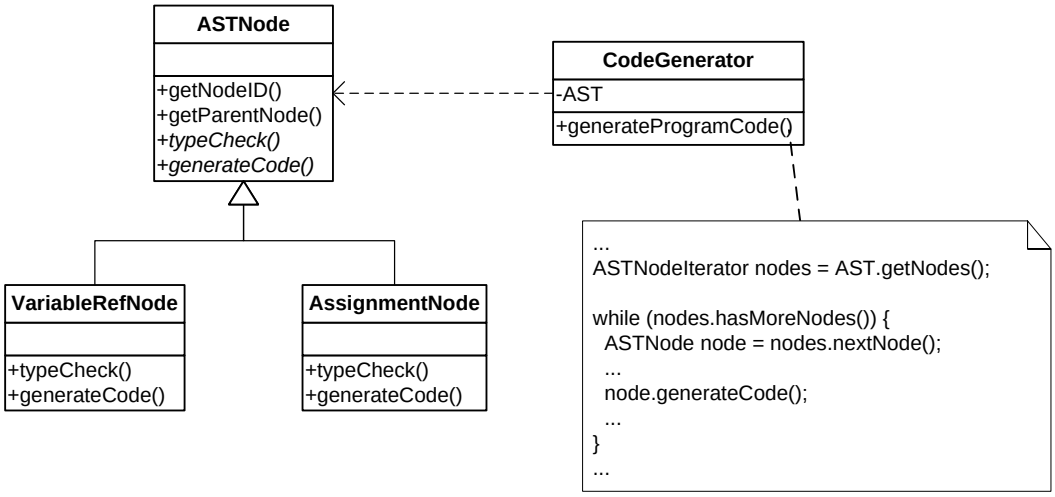


Figure 4.4.: A compiler's design, structured using simple subtyping

not closed. There is no model that is natural to all contexts!". In other words, every structure favors specific types of change and hinders others. [Mar03] refers to this as "strategic closure". As David Parnas puts it in [Par94], "since it is impossible to make everything equally easy to change, it is important to estimate the probabilities of each type of change. Then, one organizes the software so that the items that are most likely to change are confined to a small amount of code, so that if those things do change, only a small amount of code would be affected".

We will refer to those types of changes that have a higher probability as the *strategic closure*. Unfortunately, in the lifetime of a system, the strategic closure is itself subject to change. Since we cannot accurately predict all coming changes during initial design, we have to settle for those that are obvious at that time, and expect to be taken by surprise by others, later on. These unexpected changes, if done in a haste and without being accompanied by proper refactoring, are the cause of the "software aging" [Par94] phenomenon. The same applies when restructuring an existing system. Certain kinds of future needs will be easy to foresee, while others will be hard or impossible. The maintainer must decide on the strategic closure that seems most probable at the time of analysis.

As an example, let's consider the design of a hypothetical compiler, which represents programs as abstract syntax trees. Since a compiler needs to perform several complex analyses and operations, specific for each type of node in the tree, we might consider structuring the code in accordance with the "visitor" design pattern [GHJV96], as shown in figure 4.3. However, the visitor pattern is only recommended when it is desirable to make the set of operations flexible, with the cost of compromising the encapsulation of the node classes. In this case, the visitor pattern favors changes to the operations but

hinders changes to the abstract syntax tree model, especially if there are changes to its internal data.

Alternatively, placing the operations in the AST node hierarchy would lead to a structure that favors changes to the model, but makes changes to the set of operations harder, as shown in figure 4.4. If we were designing this compiler from scratch, we would expect the abstract syntax tree to suffer a lot of changes during development of the system, and stabilize at a later time. In order to minimize the costs associated with implementing these changes, it would probably be a good idea to initially embed the operations into the model, and refactor to the visitor pattern at a later time.

In conclusion, in order to favor understanding and making changes in a fragment of design, the structure needs to convey the design intent in a clear and natural way, as well as be "closed" against the changes that need to be performed. Given the importance of the two elements *design intent* and *strategic closure*, we define the notion of *design context* as follows:

Definition 1 (Design context)**.** *The design intent and the strategic closure corresponding to a design fragment will collectively be referred to as the design context of that fragment.*

4.2. Design Flaw

4.2.1. Reference Structure

As argued in section 4.1, in a given design fragment, the choice of structure depends on the fragments's design context. In each individual case, by relying on the software engineering body of knowledge and by deciding on the strategic closure, we can deduce one or more alternative structures, which have ideal characteristics from the standpoint of maintainability. The core features that characterize the quintessence of the relevant structuring decisions, are the same in all of these structures. We will refer to these common features as the *imperatives* associated to the given design context. Thus, the imperatives associated to a design context, is a set of mandatory design decisions, whose implementation in the structure guarantees optimal maintainability in that context.

Consequently, in order to judge the structure of an existing piece of code, we have to recover the underlying design context and derive the corresponding set of imperatives. Then, we analyze the existing structure with respect to the set of imperatives. If the structure fails to comply with one or more imperatives, it is deemed inappropriate in that context.

Based on an arbitrary design context, we can construct a showcase structure, that embodies all specific imperatives. We will refer to this structure as *the reference structure* for that

context. Thus, the reference structure represents a generic template, which constitutes an appropriate solution in the given design context.

Definition 2 (Reference structure)**.** *Given the set of maintainability guidelines described above, the reference structure for a concrete design context is defined as a generic template structure, which embodies all the imperatives that result from applying the guidelines to the given context.*

Remarks:

1. The reference structure is an idealized structure, without domain semantics, that is an archetype of an satisfactory solution to a given structuring problem. A reference structure cannot, and does not deal with issues that pertain to the modeling problem.
2. The generic structures described in the well known design patterns catalog by Gamma et al. [GHJV96], represent reference structures in the design contexts that warrant the use of the corresponding design pattern.

As said before, in order to come up with the imperatives in a given design context, we have to resort to the software engineering body of knowledge. The various quality models for software maintainability available in the literature offer valuable guidance in this endeavor. One of the most used quality models is the one described in the ISO/IEC 9126 standard (see section 2.3.2). It decomposes the abstract concept of maintainability into four sub-characteristics: analyzability, changeability, stability and testability.

While every model decomposes maintainability a little differently, virtually all of them touch on the two crucial aspects discussed before: ease of understanding, and ease of modifying and extending. Further desirable characteristics addressed in various models of maintainability are usually consequences that directly or indirectly result from the ones described above. Such examples include stability (few unexpected effects resulting from a change), ease of testing and ease of reuse.

Based on the existing literature, we distilled the following set of guidelines to help in the process of establishing the imperatives in a given design context:

Ease of understanding: the structure should favor the understanding of the design by humans. While software systems are complex systems by nature, reducing this complexity is possible through abstraction and decomposition techniques:

- Clean separation and encapsulation of domain abstractions into classes;
- Extraction and separation of commonality between abstractions/classes;
- Minimizing unwanted coupling by properly distributing knowledge and responsibilities among subsystems and classes

- Consistent use of a vocabulary of proven solutions to recurring problems (design patterns).

Ease of modifying or extending: the structure should favor easily modifying or extending the design. This can be achieved through:

- Isolating unrelated concerns from one another;
- Isolating things that change from things that stay the same;
- Isolating things that change more often from those that change more rarely;
- Achieving a balance between specificity and generality in order to minimize the need for redesign in case of unexpected changes in the requirements or runtime environment.

The process of applying the above guidelines to the design entities involved in the generic description of a design intent, will usually result in conflicting requirements towards the imperatives and the reference structure. By relying on the design context, an experienced maintainer is able to make the right trade-offs.

4.2.2. Definition

We are now ready to define the central notion of the present work.

Definition 3 (Design flaw)**.** *A design fragment is said to be affected by a design flaw, if its structure violates one or more imperatives in the design context of that fragment. In this case, the structure is said to be pathological.*

Remarks:

1. Although we have several pathological structures for a given design context, every such structure forms, together with design context and reference structure, a unique design flaw. Our objective is to identify and describe triplets that have practical relevance in restructuring practice.
2. The presence of a design flaw in a given design fragment justifies the need for restructuring that design fragment.
3. The resolution of a design flaw consists in replacing the existing pathological structure with one that adheres to all imperatives defined in the given design context. The reference structure is the archetype of such a structure.
4. The difference between a design flaw and a structural anomaly is crucial. A structural anomaly is an anomalous pattern in the structure, or a structural characteristic that is abnormal in some way, and represents an obstacle in the way of maintenance activities. For example, very large method bodies and very deep and narrow inheritance hierarchies represent structural anomalies. As will be shown later, a design

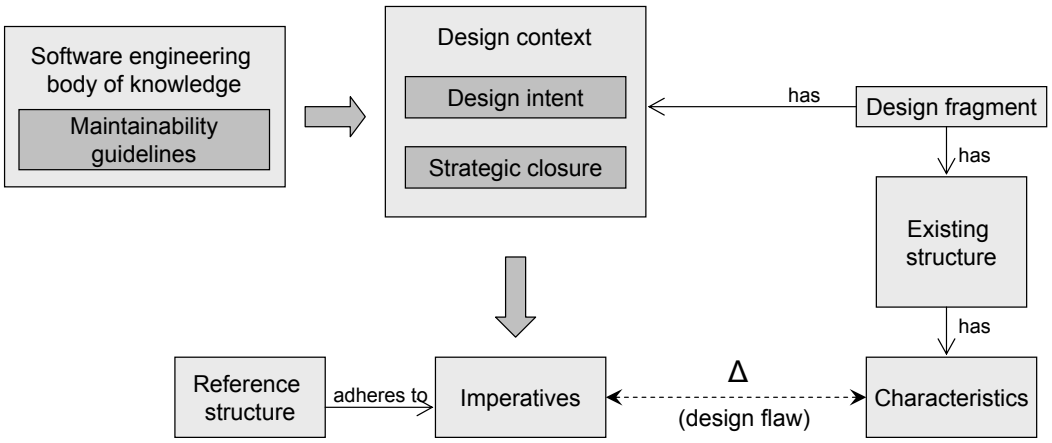


Figure 4.5.: Overview of newly defined terminology

flaw manifests itself through symptoms that may or may not constitute structural anomalies.

Let's now come back to the example of the compiler, from section 4.1. Since changes to the model are expected to be much more frequent in the foreseeable future, we may decide to "close" our design with respect to those kinds of changes. Therefore, in order to maximize maintainability of the fragment, it is required that we embed the operations into the nodes of the tree. Therefore, the reference structure would correspond to the situation in which the operations are embedded into the nodes of the tree. Any other structure that would not comply with this imperative would be deemed pathological. In particular, the structure that employs the visitor pattern would represent a pathological structure in the given design context. Under these circumstances, the presence of the visitor design pattern in the source code of our compiler represents a design flaw. The reorganization strategy that would improve the quality of the design consists in eliminating the visitor pattern by distributing the visitor methods among the corresponding AST classes.

Figure 4.5 provides a suggestive depiction of the various relationships between the main concepts defined above (block arrows represent flow of information).

4.2.3. Scope

Design flaws are a useful tool for assessing and improving *the structure of existing designs*, with respect to maintainability. They rely on the assumption that the functional requirements of the application are correctly addressed by an underlying abstract model, describing the actors and their cooperation towards the realization of these requirements.

Even when looking at design flaws that capture deficiencies in type definitions for instance, we will assume that the abstractions that those types represent are semantically valid, key abstractions in the domain, problem or solution space of the application. The only aspect of interest from a design flaw's perspective in this case, is the way in which type definitions reflect the identity, behavior and cooperation patterns of these abstractions. This section is dedicated to delimiting the scope of design flaws with respect to the confines of the present work.

In the following, we further refine the scope of design flaws with respect to four criteria: programming paradigms, granularity, organization of the restructuring process and system flavors.

Programming paradigms

In principle, the concept of design flaw is very general, and therefore applicable to any design paradigm. In particular, the set of problems posed by procedural programs are in essence very similar. However, considering the author's background, and because object orientation is today's leading programming paradigm, we decided to limit the scope of the discussion in this thesis to object oriented systems.

Granularity

Within the scope of object oriented design, one generally distinguishes three levels of increasing design granularity, whereas the separation between levels is not sharp. In the following, they will be referred to as *code level*, *component design level* and *architectural level*. Only if all three levels of granularity are properly addressed, can a system's design be generally called well structured.

Considering their motivation and distinctive constraints, design flaws are best situated on the first two of the three levels mentioned above: code level and component design level. Nevertheless, the author chose to limit the scope of the flaws discussed in this work to the intermediate abstraction level, for the following reasons:

- All concepts and methods presented in this work are applicable on the code level, without modifications.
- Our approach is not an optimization based approach, but rather a rule based one, in the sense that we rely on design principles, rules and heuristics. The vast majority of design rules, heuristics, patterns and code smells in use today, that might have a relevance in the context of tool supported restructuring are situated on the code or component design level.
- Because of the increasing level of abstraction, the intermediate level poses more challenges to automated processing than the code level.

Although in theory it should be possible to extend the scope of design flaws to the architectural level, we expect this to pose significant difficulties. The main reason is that certain aspects of the architecture are hard to capture in a pattern language (e.g. subsystem decomposition). In addition, unlike component level structures, architectural constructs such as layers or subsystem boundaries are much more difficult to automatically pinpoint in the source code. On the other hand, since the architecture of a system is normally not expected to change as often as the design of its individual components, we can argue that we need to deal with the most common case first. Therefore, the architectural level is left as a worthwhile topic for future investigation.

Organization of the restructuring process

In the present work, we describe a method that unites problem detection and analysis activities within object oriented restructuring, in a systematic, tool supported process. Therefore, the method is not more than an effective tool in the hands of the maintainers, who must use it in a way that conforms to the goals of each restructuring process.

In particular, we rely on the assumption that implementing the reference structure described by a design flaw, does not have negative side effects elsewhere in the design. This is because applying the reference structure is like applying a design pattern. Before deciding to do it, we make sure that the design context asks for it. But on the other hand, the merits of the reference structure prescribed for a given design context are not always absolute. For instance, error handling may be implemented using either function return values, or exception handling. As already mentioned in section 4.1, the design flaws presented throughout the work ignore issues related to programming style and conventions, as well as any other design constraints that might arise from limitations imposed by the physical environment of the system, such as performance, efficiency or costs associated to certain communication channels. However, we don't mean to say that such issues are completely out of the reach of design flaws. Instead, we say that special circumstances require specially adapted design flaws. The maintainer is responsible for putting together and using a set of design flaws that adequately addresses the particular characteristics of the development team and project.

System flavors

The concept of design flaw in general, and the ones specified in the current work in particular, are independent of the application domain or implementation flavor. Constraints that are specific for the application domain (e.g. real time systems, safety critical systems, etc) or the implementation flavor (e.g. parallel processing, distributed processing), may impose conditions that cannot be met by generic design flaws. In such cases, it is the

responsibility of the maintainers to put together, and use a set of design flaws that adequately meet these conditions. These issues are outside the scope of the present work.

4.2.4. Anatomy of a Design Flaw Specification

As argued in the previous sections, a design flaw is uniquely determined by a design context and a pathological structure. The reference structure is one that embodies the set of imperatives that result from the context.

Concerning the form in which design flaws are described, we opted for a more pragmatic, semi-formal specification of our design flaws. Although a formal representation of software structure is possible (see section 2.3.1), the design context is much more difficult to represent formally, because of its semantic nature. Concretely, a design flaw specification combines natural language descriptions documenting design context, trade-offs and decisions, with UML representations of program structure, in a consistent, pattern-like form. Each design flaw specification is divided into the following sections:

Name
: Just as in the case of design patterns, a design flaw needs a name that captures the essence of the flaw. A suggestive name helps in forming a dedicated restructuring vocabulary, which in turn makes the exchange of ideas and know-how more efficient.

Description
: The purpose of this section is to convey the meaning of the flaw in a concentrated and suggestive way. A concrete instance of the design flaw is depicted in a UML diagram and is shortly discussed.

Design Context
: This section describes the design context of the flaw. In conformance with definition 1, the description of the design context comprises the description of the design intent as well as strategic closure.

Imperatives
: Guided by the basic maintainability model in section 4.2.1, and relying on the software engineering body of knowledge, we derive a number of mandatory features that the reference structure must have, called imperatives. Adhering to the imperatives guarantees that maintainability of the design fragment is maximized, within the specified design context.

Pathological Structure
: This section contains a UML representation and description of the pathological structure, specific for the design flaw. The generic structure discussed here illustrates a number of structural features, which will later be used in tool supported

diagnosis (see also chapter 5). In addition, a justification highlighting the shortcomings of the pathological structure with respect to the imperatives is provided.

Reference Structure

This section contains a UML representation and description of the reference structure for the design context at hand. The reference structure is characterized by the fact that it closely adheres to the imperatives established for the given design context.

4.3. Design Flaw Catalogue

As already explained above, a design flaw is uniquely defined by a design context and a pathological structure, while a reference structure can be derived from the two. The reference structure is the result of materializing the design intent, while taking into account the set of design guidelines for maintainability as well as the types of changes that are expected to occur to the system.

In theory, when you want to specify a design flaw, you can start with any one of the three elements, and successively work out the other two. The goal is to find those triplets that are statistically relevant by representing situations that are commonly found in practice. A good strategy to find relevant cases is to start either from a structural anomaly such as a code smell (potential pathological structure), or from a design pattern (potential reference structure).

Appendix A contains a catalogue of 10 design flaw specifications, derived partly based on existing design patterns, partly on well known code smells. The catalogue is of course by no means complete, but can be considered relevant for the following reasons:

- All the flaws contained within the catalogue capture situations which can be frequently encountered in large legacy systems, irrespective of their domain or implementation flavor.
- The catalogue is representative, because it touches on all important concerns of object orientated design, such as object definition, inheritance hierarchies and the distribution of responsibilities between classes. Thus, design flaws A.6 and A.7 address issues concerning class definition. Design flaws A.4 and A.5 address issues concerning the distribution of behavior between abstractions represented by classes. Design flaws A.8, A.9 and A.10 address various problems that pertain to specialization hierarchies. Finally, design flaws A.1, A.2 and A.3 deal with situations that warrant the use of inheritance, and are typical for software that is designed in a procedural style.
- The catalogue contains design flaws that address both well known structural anomalies (e.g. god class, feature envy, data class) as well as design patterns (e.g.

visitor, state, template method), bringing them together under the roof of object oriented restructuring.

Code excerpts are in Java, one of the most popular object oriented languages in use. The pattern form used to describe design flaws in appendix A, extends the basic template described in 4.2.4, with a number of additional sections that will be discussed throughout the following chapters.

Chapter 5.

A Method for Diagnosing Design Flaws

The previous chapter introduced the notion of design flaw, which is defined in terms of a delta between the reference structure in a given design context, and the existing structure. This chapter presents a method for tool supported diagnosis of design flaw instances. To this end, we will first introduce a running example that will be used throughout the present and the next chapter. Subsequently, we discuss the idea that forms the basis of our diagnosis method and define some supporting concepts. Finally, we present the notion of diagnosis strategy and discuss issues concerning the implementation of a diagnosis tool.

5.1. Example: Schizophrenic Class

As running example for the discussion in the current and the next chapter, we chose the design flaw called "schizophrenic class", specified in section A.6 of our catalog. For the sake of easier reading, we reproduce the specification in the following.

5.1.1. Description

In object oriented design, a class should not capture more than one key abstraction. Key abstractions are defined as the main entities within a domain model, and often show up as nouns within requirements specifications ([Rie96]). A key entity is an abstraction that stands on its own in the abstract model that results from modeling activities.

A "schizophrenic class" is a class that captures two or more key abstractions. Class schizophrenia is common in situations where a system is developed incrementally, without restructuring in-between increments. Thus, a class that is defined very early in the process, may prove to be too abstract later on, as it receives more and more responsibilities. Consequently, the class needs to be broken into fragments that capture more fine-grained abstractions.

Another possible scenario for the formation of schizophrenic classes is in the process of migrating a procedural system to an object oriented language. Large chunks of formerly

global data are grouped together with functions that use this data into a single, large and noncohesive class.

In both scenarios mentioned above, the class encapsulates the data and behavior of two or more design abstractions. In addition, the encapsulated abstractions are described based on their identity. In other words, we have an object oriented topology of the underlying abstract model that results from modeling activities (i.e. the model describes the encapsulated abstractions as individual, cooperating actors).

Classes whose names contain words such as "system", "subsystem" or "manager" are likely candidates for class schizophrenia. However, there are also exceptional situations, in which a class is intended to provide a unified, simpler interface, to a complex set of interfaces that form a subsystem. This is the case of the "facade" design pattern. Nevertheless, a facade is primarily delegating to the responsible classes and does not aggregate all the data that define the abstractions in the subsystem.

Figure 5.1 shows such an example instance, where the class `SmartHomeManager` implements three interfaces that define clearly separated responsibilities: the air conditioning system, the alarm system and the lighting system. The implementation of these three abstractions relies on partially overlapping data, but is mostly not related to one another. For example, the air conditioning shares the attribute `windowStates` with the alarm system, and the attribute `windowShadeStates` with the lighting system. The schizophrenic class doesn't always implement explicit interfaces, but we decided to include them in the example, because it is suggestive of the impersonation of various roles that the instance of a schizophrenic class does for various clients in the system.

A schizophrenic class negatively affects the ability to understand and change the individual abstractions that it captures, in isolation.

5.1.2. Context

Design intent

You want to express a set of individual design abstractions in a model having an object oriented topology, as classes in the system. Alternatively, you are migrating a procedural program to an object oriented language, and you need to define cooperating classes, from chunks of global data and functions.

Strategic closure

The design abstractions under consideration are expected to change independently from one another.

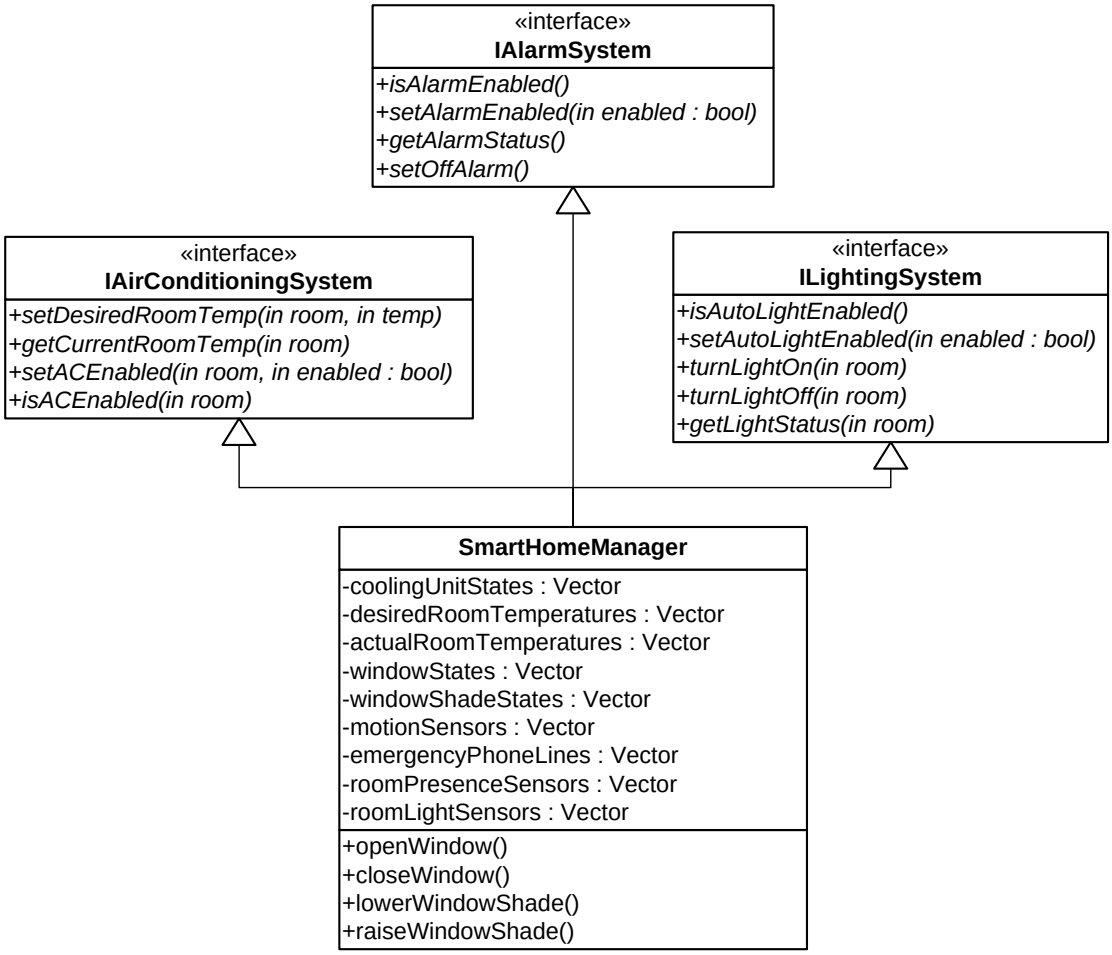


Figure 5.1.: An example of *schizophrenic class*

5.1.3. Imperatives

In general, in order to maximize maintainability, each class should capture no more than one key abstraction. Key abstractions are abstractions that stand on their own in the abstract model which determines design intent. In addition, all data that is related to one of the class' responsibilities, and all behavior that is related to the class' data should be kept together, in the same class [Rie96].

In order to maximize maintainability within the described context, we need to isolate those design abstractions from one another, that are expected to change independently. This implies that the functional decomposition of the original class' behavior needs to be replaced with an identity based decomposition that reflects the decomposition of data

among the involved abstractions, so that data and associated behavior are kept together.

5.1.4. Pathological Structure

As illustrated in figure 5.2, the pathological structure is characterized by the fact that the offending class encapsulates more than one key abstraction. As a consequence, we expect to find relatively isolated clusters of data and associated behavior, that represent the implementations of the corresponding logical interfaces. The interfaces can be either implicit, or declared explicitly, and they correspond to the various abstractions encapsulated by the class. In the latter case, they are not trivial, or so called marker interfaces. The first cluster in the generic structure presented in the figure, comprises the methods `method1()` and `method2()`, which use `attr1`, `attr2` and `attr3`, and the second cluster comprises methods `method3()` and `method4()`, which use `attr1` and `attr4`.

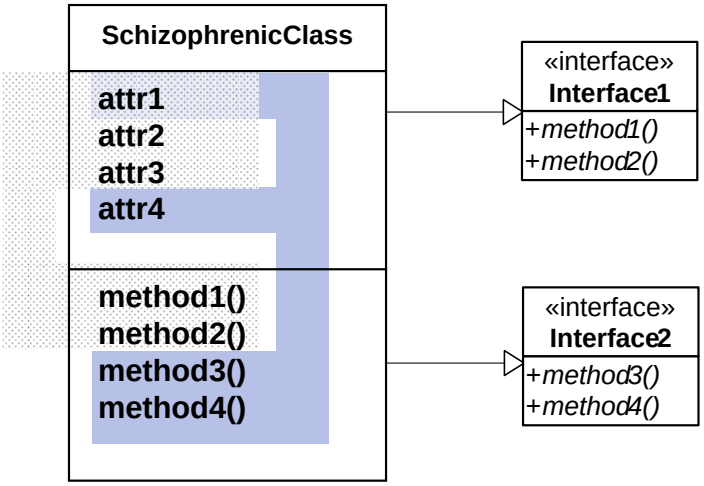


Figure 5.2.: Pathological structure for *schizophrenic class*

Figure 5.2 depicts a rather favorable situation, in which the methods of the class can be assigned more or less unambiguously to clusters of data. This denotes an strong object oriented topology of the underlying abstract model. In the worse scenario, the class may appear to be functionally cohesive, in the sense that an unambiguous association of methods to the attribute clusters which describe the encapsulated concepts is not possible. Such instances are harder to diagnose reliably because heuristics must rely on indirect manifestations in the structure, such as class size and the way in which clients use the class. Thus, a schizophrenic class is likely to be large in terms of data that is defined, and to have a relatively large number of clients in the system.

The pathological structure described above has a negative effect on the maintainability of the individual abstractions contained within, because it hinders understanding and changing them in isolation.

5.1.5. Reference Structure

As shown in figure 5.3, the original class has been broken up, based on the abstractions encapsulated by the class. Again, the figure illustrates the more favorable situation, in

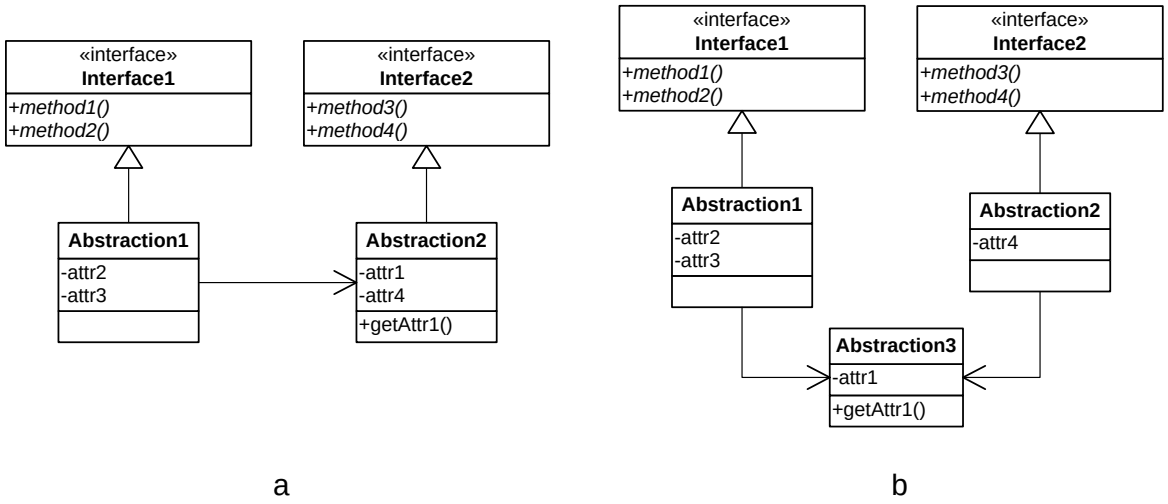


Figure 5.3.: Reference structure for *schizophrenic class*

which we have a strong identity based decomposition of class members, based on the identities of the encapsulated abstractions.

With respect to attributes that are used by two or more functional clusters, we have two possibilities. In the first scenario, the attribute can unambiguously be assigned to one of the newly created abstractions (figure 5.3 a). In this case, the attribute should be moved into its natural home, a possible guiding heuristic being that the abstraction that changes the attribute should also own it. All methods that use attributes belonging to a foreign class, may need to be split according to the class identities and associated responsibilities (see A.5). In case there are still foreign methods that need read access to the attribute, a getter accessor method can be created.

The second scenario is when one or more commonly used attributes really don't belong in any of the abstractions implementing the interfaces, but semantically forms a helper class for them (figure 5.3 b). In this case as well, we should avoid providing accessor methods. Rather, the methods that use foreign attributes should be investigated, in order to identify

possible higher level services that could be moved along with the attributes into the helper class. Thus, we ensure that functionality is properly decomposed, according to each class' identity and associated responsibilities.

In any of the two situations the original class can provide a "facade" to the newly created classes.

5.2. Idea

Given a design flaw specification such as the one presented in section 5.1, the questions immediately arise: how can we identify instances of the flaw, and how can we automate the process? Before we discuss these questions, we must state more precisely what we actually mean by "identify instances of a design flaw". To this end, we define the term *diagnosis* as follows:

Definition 4 (Diagnosis)**.** *We define diagnosis as the process of identifying design fragments, whose structure and design context match the pathological structure and design context described in a design flaw specification.*

According to this definition, for a successful diagnosis we need a positive match on three things: the design fragment's structure, design intent and strategic closure. Since strategic closure is generally independent of the existing code, matching it will require human intervention. Nevertheless, we will attempt to target the other two elements, structure and design intent, in an automated fashion. However, we want to limit ourselves in this endeavor to a static analysis of the system, because we want to avoid the complications arising from the need to guarantee the full code base coverage of the test runs, as well as the instrumentation of the code.

In other words, we want to use static analysis to identify fragments of structure, that match the description given in the design flaw specification and have a well defined intent. However, in order to do this, we need to set up a hypothesis which we will validate in chapter 7. The hypothesis that we rely on in this case, is that design intent manifests itself in the static structure. Although this is a reasonable assumption to make, such structural features are sometimes extremely subtle and require a very fine-grained analysis.

To illustrate this point, let's consider the two cases depicted in figure 5.4. While in the case of the first structure, the intent is a specialization hierarchy of the class `Document`, in the second case we are dealing with variation of object behavior, based on an abstract notion of state (i.e. the variable `conState`). These are two very different design intents, which correspond to different design flaws. Nevertheless, the two structures are nearly identical, except for the names of the variables, and the fact that in the second case the state variable is updated in the conditional branches, while in the first case it is not. For

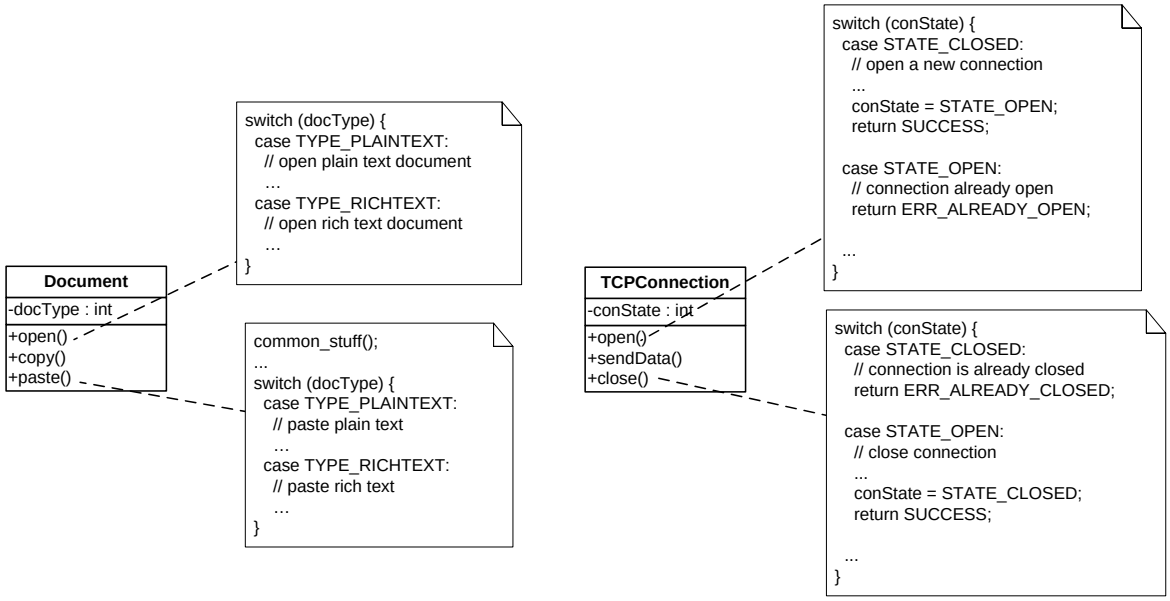


Figure 5.4.: Different intents may result in very similar structures

a detailed presentation of the two design flaws involved, please consult the specifications A.1 and A.3 in appendix A.

The above example suggests a different problem as well. Certain features that allowed us to establish the correct intent in each of the two cases might not be always present in the structure. For example, the names used for the checked variables were suggestive of the different intents behind the otherwise very similar structures. One could for example build a lexical analysis of variable identifiers into the diagnosis process of these two individual flaws. However, it is reasonable to assume that variables may also have different, less suggestive names, and therefore, such a constraint on identifiers must be treated as optional. This is somewhat similar to the way in which a disease is diagnosed in the medical world, where some of the symptoms that characterize a disease may or may not manifest themselves. Thus, the idea of our approach is to define a set of "structural symptoms" that *are likely to appear together* for a given design intent and pathologic structure. Using state of the art pattern matching techniques, we look for instances of these symptoms, in a manner that is similar to medical diagnosis. In other words, the design flaw is characterized by a linear combination of "structural symptoms", and the more of these symptoms we can detect, the more probable it is that we got the right design flaw. Of course, this is only a qualitative statement, and may not hold in all cases, since individual symptoms might need to be given different weights. See section 5.3.3 below for a discussion on weighing different indicators, based on their relevance. In addition, different

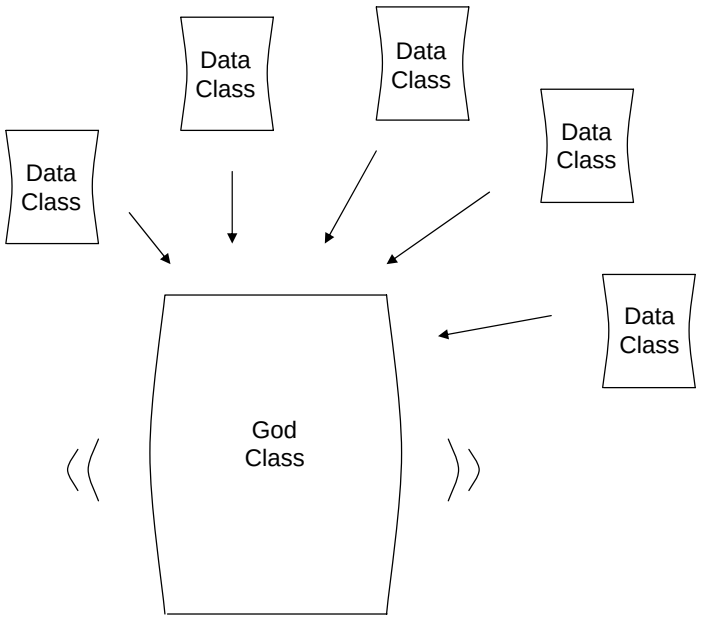


Figure 5.5.: Migration of functionality to a central class

design flaws may share common symptoms, such as the repeated conditional constructs in figure 5.4.

As a further, perhaps easier to grasp illustration of this idea, let's consider the situation depicted in figure 5.5, in which a class of the system attracts functionality from a number of other classes around it. This situation corresponds to the anti pattern known as "the blob" [BMMM98], or "god class" [Rie96]. The class that attracts functionality has a tendency to gain weight, which corresponds to the code smell "large class" [Fow99]. Conversely, the classes holding the attributes and loosing functionality tend to become slimmer, which corresponds to the code smell "data class" [Fow99]. Furthermore, those methods in the central class that use foreign attributes in the data classes, are likely suffering from the code smell "feature envy" [Fow99].

This is a good example of how three apparently unrelated code smells may collectively indicate a common underlying cause: the migration of functionality from the periphery towards the central blob. However, this does not mean that in every case of migrated functionality, all three smells will be present. For instance, the centralization of behavior might not be extreme enough to create data classes, in which case this code smell would not be detected. Furthermore, a large class can also have other causes, such as code duplication or too many responsibilities assigned to it by the designer.

The rest of this chapter addresses the topic of specifying the above "structural symptoms"

(which we call indicators) of a design flaw, as well as implementation issues of the diagnosis process.

5.3. Indicators

As discussed in the previous section, design flaw diagnosis is similar to medical diagnosis. Instead of combinations of symptoms that characterize a disease, we use combinations of structural features that describe a specific pathological structure and design intent. For the purposes of the present work, we define the notion of indicator, as follows:

Definition 5 (Indicator)**.** *Any kind of automatically detectable finding, originating from the static structure of the system, which expresses a distinctive feature of the pathological structure or the manifestation of design intent, is called an indicator of the corresponding design flaw.*

Remarks:

1. According to the above definition, code smells can be indicators for various design flaws. At the same time, an indicator must not necessarily represent a code smell. Unlike other approaches, our method focuses on the combined presence of several different symptoms, that must not necessarily have a negative connotation, or represent code smells. Thus, we are able to detect inappropriately structured code, even if there are no out of the ordinary manifestations, in the form of structural anomalies.
2. In medical practice, a disease must not necessarily manifest itself through all of its symptoms, and a symptom may be shared by several distinct diseases. Similarly, a design flaw may manifest itself through a subset of its characteristic indicators, and indicator sets for different flaws may partially overlap.

Let us now return to our example design flaw “schizophrenic class”, introduced in section 5.1, and to find a set of indicators that characterize the corresponding pathological structure as well as design intent.

5.3.1. Defining Indicators for “Schizophrenic Class”

First, let’s focus on the description of the pathological structure, given in section 5.1.4, to identify some of the more obvious indicators. The description mentions two types of structural features: the presence of “more than one key abstraction”, which translates into “isolated clusters of data and associated behavior”, and the presence of “explicit or implicit interface definitions”, which the class implements.

Based on the above, we can formulate the first two indicators of the design flaw, as follows:

Indicator 1: *The analyzed class has a low internal cohesion.*

Indicator 2: *The class exhibits distinct personalities with respect to disjoint groups of clients, either by explicitly implementing two or more non-trivial interfaces, or by having disjoint groups of clients that use disjoint fragments of the class' public interface.*

So far so good, but can we find further indicators? Let us now look at the description of the corresponding design intent. The description talks about several design abstractions, or chunks of global data originating from a procedural system, which constitute elements in a design fragment having an object oriented topology. What this is actually meant to express is that the abstractions in question represent key abstractions in the design, which means that they have semantically independent responsibilities. These responsibilities may potentially serve several clients, and thus justify the existence of these abstractions as separate entities (i.e. types) in the design of the system. How could we distill this information into further indicators for our design flaw?

First, if we are talking about several abstractions or "chunks of data" that make sense on their own, it is reasonable to assume that a schizophrenic class which is supposed to encompass several of these, could potentially be characterized by a "larger than usual" amount of internal data. This can be expressed as:

Indicator 3: *The class defines a large number of attributes.*

This indicator is a perfect illustration of the facultative nature of design flaw indicators discussed in section 5.2, because a schizophrenic class *doesn't have to* have a large number of attributes, but rather it *is more likely to* have a large number of attributes. Conversely, if a class that we stumble upon in our analysis of a system happens to define a large number of attributes, increases our suspicion that the class in question is "schizophrenic", in the sense of our design flaw.

Similarly, a class that encloses several independent abstractions is likely to also have more behavior than usual. In addition, if the object oriented topology of the enclosed abstractions is not that pronounced, as described in the second part of section 5.1.4, the class is also likely to be more complex. These two characteristics are jointly expressed by the following indicator:

Indicator 4: *The class is heavyweight, in the sense that it is very large and has a high complexity.*

Finally, the last proposed indicator for the design flaw "schizophrenic class" expresses the idea that a class which encloses several key design abstractions must serve all the clients of each individual abstraction, and therefore are likely to have a higher incoming coupling than usual:

Indicator 5: *The class constitutes a "bottleneck", in the sense that a significant proportion of all classes in the system depend on it. A class depends on another if it references it.*

At this point, it is important to note that the above indicators are defined informally, in a language independent manner. Also, the definitions may contain expressions such as "very large class", "low cohesion", "many clients" and so on. Of course, this type of definitions constitute the first step in automating the diagnosis process, but their main purpose is to document, to describe. In the subsequent step, indicator descriptions need to be translated in something more precise, which can be executed mechanically. In order to achieve this, we will build upon the results of previous doctoral theses and research projects at the technical university of Karlsruhe[1] and its associated research center, FZI[2]. The author of this work has been involved in some of these projects personally.

5.3.2. Formal Definition of Indicators

In general, identifying structural fragments that satisfy a set of requirements is called structural pattern matching. As discussed in section 5.4 below, for practical reasons, structural pattern matching approaches work on an abstract model of the system's structure, called the structural model. The rules that govern the construction of a structural model are described by its corresponding meta-model, which is language specific. In order to be able to use our indicators in an automated structural pattern matching tool, we need to express them as relations on the set of entities defined by the meta-model.

In today's state of the art, we have two categories of approaches for structural pattern matching, with applications in static analysis. They are on one hand the graph pattern matching approaches, with applications in automated design pattern detection, such as [KP96, AFC98], and on the other hand rule based querying with applications in automated detection of code smells, such as [Ciu01].

In general, graph pattern matching techniques are not suitable for our problem, because of the following reasons:

- They require an a-priori mapping of the artifacts of interest to the nodes and edges of a graph. This task has proven feasible especially at or above the level of granularity that corresponds to class members, such as in the case of most design patterns. As will be shown, we generally need to approach the granularity level of the abstract syntax tree. Thus, the size of the graph as well as the pattern that needs to be identified becomes unpractical. Defining a sufficiently large and fine-grained search pattern is very difficult, or impossible. Furthermore, as suggested by figure 5.4, we will often have to deal with a very wide dynamic range of granularity. In other words, we will need to specify both very fine-grained and coarse-grained features at the same time.

[1] Universität Karlsruhe (TH), http://www.uni-karlsruhe.de
[2] FZI Forschungszentrum Informatik, http://www.fzi.de

- Performance and memory efficiency of the search decrease, usually in an exponential manner, with the increasing size of the model and search pattern. Our goal is to analyze real world systems that have several hundred thousand lines of code.
- The fragments we will be looking for may differ slightly in their structure, between instances of the same design flaw (i.e. individual indicators are generally optional). Supporting optional or variable structures may be hard or impossible in graph based pattern specifications.

Rule based approaches have been successfully used in identifying both code smells and design patterns, because rule based formalisms naturally fit the form in which principles and rules of design in general, and indicators in particular, are formulated. Since rules usually express some limited characteristics of the structure, they scale much better with respect to both specification effort and execution time. Also, it is relatively easy to specify both coarse-grained and fine-grained rules.

Since rules are commonly defined as relations over elements of a structural model, existing approaches do not support variability. In other words, it is usually not possible to define optional portions of a rule. We overcome this limitation by capturing each optional aspect (i.e. indicator) in a separate rule, and combine these rules in a process that resembles medical diagnosis.

In order to express indicators as rules, we can resort to several formalisms, such as detection strategies [Mar02, LM06], queries expressed in a logic programming language such as Prolog [Ciu99, Ciu01], or relational algebraic operators or the SQL query language [Ciu01, TSK05]. Alternatively, we can simply write a program that computes relations using the development environment's specific APIs.

While in theory, all of these formalisms can be used interchangeably, each of them presents certain advantages, depending on the type of indicator as well as the form in which the structural model exists. For example, detection strategies are best at specifying informal, fuzzy, metrics based rules, such as indicator 4 defined above. Prolog queries on the other hand, are a natural match for sharp structural pattern matching rules, such as "a class must not know any of its descendants". Furthermore, if the structural model exists as a set of prolog facts, the specification is directly executable by an inference engine. Finally, SQL, as a loose implementation of a relational algebra, is a natural, directly executable formalism, for the case in which the structural model exists in the form of a relational database.

In the following, we exemplify the specification of an indicator using a detection strategy. Indicator 4 defined in the previous section, seems a perfect candidate for being expressed using a detection strategy, because it is informal, fuzzy and refers to measurable sizes. We repeat its definition below, for convenience:

Indicator 4: *The class is heavyweight, in the sense that it is very large and has a high complexity.*

As described in [LM06], the process of transforming an informal rule such as indicator 4 into a detection strategy, has four steps:

1. The first step consists in picking out the structural characteristics of interest from the informal rule. In our case, the rule directly mentions two types of characteristics: class size and complexity.
2. Next, we need to select appropriate metrics that quantify each of the previously identified characteristics. Depending on the situation, one may choose well-known metrics from literature, or define a custom metric. In our case, we could imagine at least two different ways of measuring the size of a class: counting class members and counting lines of code. Similarly, the literature provides several complexity measures, mostly based on McCabe's cyclomatic complexity [McC76]. Alternatively, we might choose a single metric that combines both of the previous aspects, such as the WMC (Weighted Method Count) metric [CK94].
3. In the third step, we need to establish appropriate comparators and threshold values for each metric. In all but the rarest of cases, we need to rely on statistical data for threshold values of acceptable quality. This statistical data can either be based on a large number of case studies, or it can be derived from the system that is analyzed. For a more detailed discussion on choosing thresholds in the context of detection strategies, please refer to [LM06]. Let us assume for a moment that the chosen threshold values for WMC is 40, and the comparator used is "greater than".
4. Finally, the detection strategy is "composed" using logic operators, as appropriate for the rule being specified. In our case we do not have to do anything, since we only use one metric.

Thus, the final textual form of the detection strategy would be:

```
I2 := WMC(c) > 40
```

where `c` represents an instance of "`Class`" (see figure 5.7 below), in the system's structural model.

At this point, the following questions arise: how "good" is this indicator? Does it fulfill its intended purpose well? Could we potentially improve it? In order to answer these questions, we need to discuss the issue of indicator quality.

5.3.3. Quality of Indicators

Conceptually, an indicator can be regarded as a filter, whose input and output are sets of structural entities on a given level of granularity. In the case of the indicator formulated above, we could for example define the input as the set of all classes in the system. Then

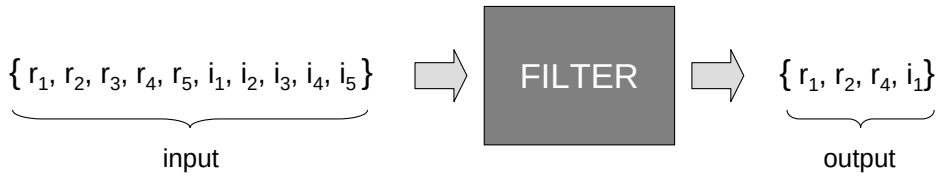


Figure 5.6.: A set filter, viewed as a retrieval mechanism

the output is a subset of the initial set, that has the property of having a weighted method count that is higher than 40.

Generally, the quality of a set filter can be assessed in terms of the relevance/usefulness of its output for a given input. To this end, we can call upon the classical data retrieval measures of precision and recall. If we look at the filter as a retrieval mechanism (see figure 5.6), that is expected to extract relevant elements (r_i) from its input and pass them to its output, precision and recall of the filter are defined as follows:

Precision: represents the ratio of the number of relevant elements in the output to the total number of elements in the output. For the example in figure 5.6, $P = \frac{3}{4}$;

Recall: represents the ratio of the number of relevant elements in the output to the number of relevant elements in the input. For the example in figure 5.6, $R = \frac{3}{5}$;

An ideal filter has both 100% precision and 100% recall. In most practical applications however, it is very hard or impossible to implement ideal filters. Consequently, precision and recall are usually not standalone, but inversely related: the higher one of the measures, the lower the other. Thus, our goal is to find a suitable balance, so that we have an acceptable precision, without loosing too many of the relevant inputs.

Therefore, in theory, in order to assess the quality of an indicator, we need to measure its precision and recall. Unfortunately, this is easier said than done, because in order to compute recall we need to know the number of relevant elements in the input. In the case of our example indicator, this would require us to either know or accurately estimate the real number of schizophrenic classes in the entire system.

Since there is no practical way of knowing this, other than a painfully laborious manual inspection of the system, we need a different way of ensuring a minimum level of quality for newly specified indicators. To this end, we will employ the following artifice in the process of specifying detection rules for indicators. Specifically, we will use a two step procedure. In the first step, we specify a raw version of the indicator, in such a way that we get close to ideal recall. Then, if possible, in the second step of the procedure we optimize the indicator's precision, by imposing further constraints, in a way that arguably minimizes loss of recall. As a rule of thumb, we should afford giving priority to recall

over precision, because design flaws are diagnosed in terms of a combination of several different indicator types.

If we apply the artifice described above, we get indicators that tend to lie on the side of higher recall and lower precision. Thus, precision alone tends to receive a greater relevance in judging the quality of an indicator:

$$P(I) = \frac{Conf(I)}{Found(I)} = \frac{Conf(I)}{Conf(I) + Dis(I)} \tag{5.1}$$

where $Found(I)$, $Conf(I)$ and $Dis(I)$ represent the total number of automatically found, manually confirmed and dismissed instance candidates of a given design flaw, for which the indicator I had been found during the analysis.

Although $P(I)$ can tell us something about how well an indicator works individually, it doesn't tell us anything about how that indicator performs in combination with others, in the diagnosis process of a given design flaw. In other words, we would like to have a way of characterizing the relative contributions of various individual indicators in the diagnosis process of a design flaw. To this end, as in the medical sciences, we can resort to regression analysis[DS98], a statistics technique which can be used to describe the relation that exists between a so called response variable (in our case the presence or absence of a design flaw instance), and a set of so called explanatory variables (the outputs of various indicators that characterize that particular design flaw).

The simplest form of regression is the so called linear regression method, in which the relationship between the response variable and the explanatory variables takes the following general form:

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p + \varepsilon \tag{5.2}$$

where Y is the response variable, X_i are the explanatory (also called independent) variables, ε is a random term, α is a constant term (also called the intercept), and β_{1-p} are a set of coefficients that determine the regression model.

Given a series of Y_j and a series of $X_{1,j}, X_{2,j}...X_{p,j}$, where $j = 1,...,n$ the so called linear regression method of ordinary least squares [Gau21] determines a set of $\beta_1,...,\beta_p$ such that the function of $Y(X_1,...,X_p)$ corresponding to equation 5.2 best approximates the given series of Y_j. The name of the method comes from the fact that the estimated coefficients minimize the sum of squared error estimates for the given data set. The method assumes that the variables $X_1,...,X_p$ are as suggested by their name, independent.

In the context of design flaw diagnosis, the resulting coefficients can be understood as weights, which tell us about the relative influence that each individual indicator has in the outcome of the diagnosis. In chapter 7, we will make use of the above techniques when assessing the quality of the indicators that are used in evaluating the approach.

5.4. Tool Support

In the previous sections, we described the process of defining indicators for a design flaw. In the following, we discuss the implementation of tool support for indicator detection and the diagnosis process in general.

5.4.1. Structural Models

As already discussed in chapter 2, the source code constitutes the most detailed and only guaranteed reliable source of structural information for static analyses. In practice however, tools usually do not analyze source code directly. Instead, they use a model that is extracted from the system's source code, known as the structural model. The so called meta-model describes the building blocks and rules that are used for building specific structural models (see 2.3.1).

Any meta-model can be used in the diagnosis process, provided that it is fine-grained enough to allow the implementation of all indicators that are required. The indicators defined for the design flaw that we chose as our running example throughout this chapter, requires a meta-model that can reach down to the individual statements. As a minimum, it must be fine-grained enough to allow distinguishing conditional statements and individual variable accesses within method bodies. Such fine-grained meta-models can be usually found in IDEs with support for code refactoring, such as eclipse[3], or in libraries and frameworks for code analysis and transformation, such as RECODER[4]. These meta-models usually go down to the abstract syntax tree level, and thus are complete with respect to a given programming language. Consequently, they are usually limited to that single programming language. However, a complete model has the advantage that it allows generating the complete source code of the system, which in turn allows the implementation of both analyses and refactorings on the same model.

An interesting compromise from the standpoint of model granularity is achieved by a meta-model, designed at the FZI Forschungszentrum Informatik, within the frame of the project QBench[5], and used in the open-source analysis tool SISSy[6]. It is presented schematically in figure 5.7.

The QBench system meta-model [TSK05, TS05] was especially designed for fine-grained static analysis, goes down to the level of and distinguishes between several types of statements, and can be annotated with information concerning code duplication. It is however not detailed enough to allow code generation.

[3] http://www.eclipse.org

[4] RECODER (http://recoder.sourceforge.net) is a project under joint development between the University of Karlsruhe and the FZI Forschungszentrum Informatik

[5] QBench is a research project funded by the German federal government (BMBF 01ISC10A-01ISC10G)

[6] http://sissy.fzi.de

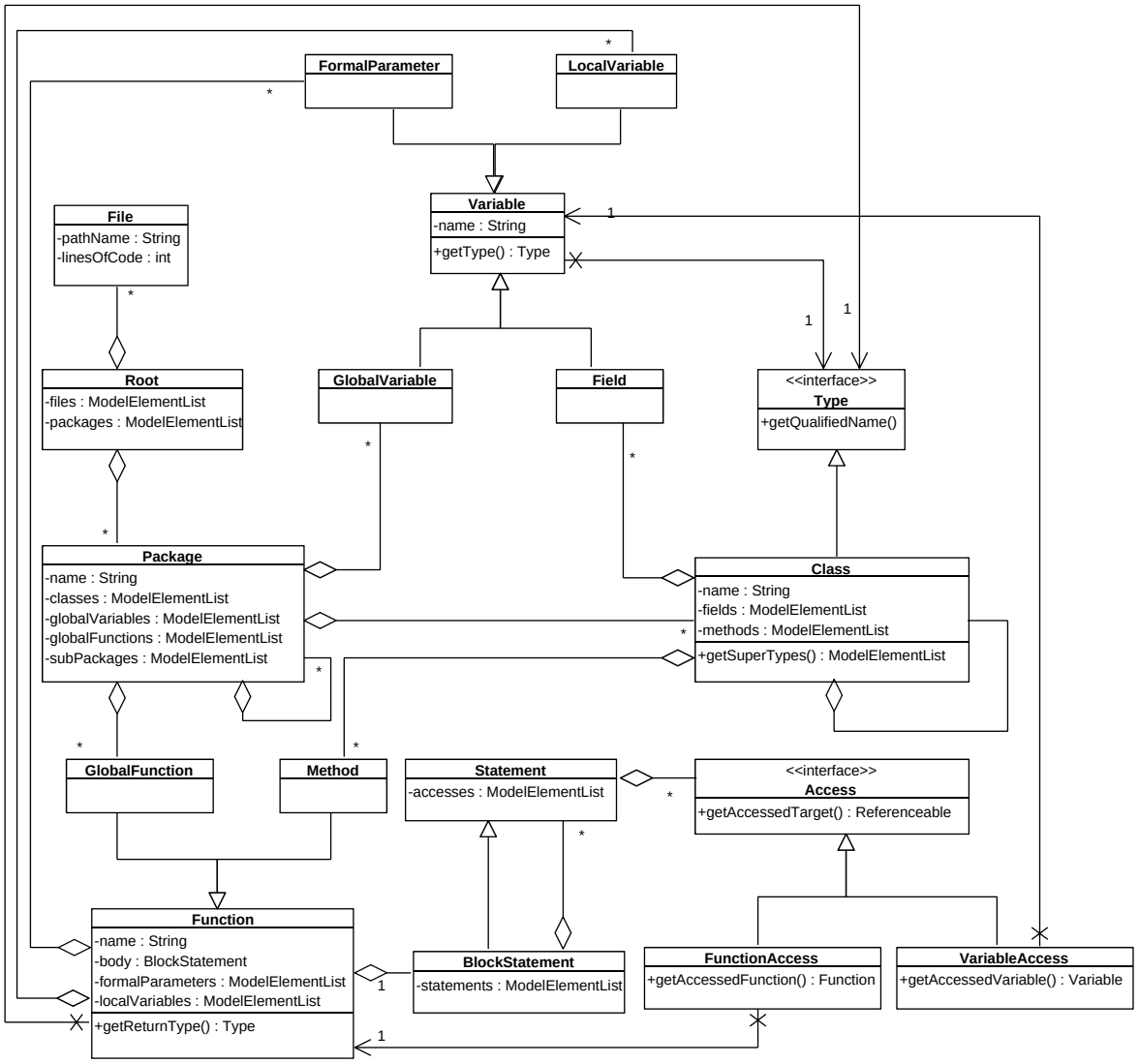


Figure 5.7.: Overview of the meta-model used in the tool SISSy

Nevertheless, instance models that adhere to it are able to simulate all common code refactorings. Thus, this meta-model can be successfully used for making various predictions, such as impact analysis of code refactorings.

5.4.2. Implementing the Diagnosis Process

Until now, we showed how individual indicator instances can be automatically detected, using existing rule based structural pattern matching. In the rest of this chapter, we discuss implementation issues that pertain to the diagnosis process as a whole.

Diagnosis is based on the assumption that the larger the set of observed indicators in a given design fragment, the higher the confidence that the fragment's structure and design intent match those specified for the respective design flaw. Conceptually, checking that a given design fragment simultaneously exhibits several indicators, is analogous to using the unification feature in order to specify complex rules in Prolog, where the terms being unified represent elements in the structural model. The only difference is that indicators are generally optional, so failure to detect an indicator should not break the entire query.

In the simplest of cases, all indicators of a design flaw are completely independent, so their detection can happen in any order. However, for practical reasons, we want to choose one of them as an initial filter, in order to reduce the search space for the analysis as much as possible. This filtering role of the chosen indicator means however, that any potential design flaw instance that does not have that particular structural characteristic is irrevocably excluded from the analysis. That is why, the initial filter should only capture features that are mandatory for *any* instance of that design flaw. Most of the times, the pathological structure possesses one or more such features. In addition, the initial filter may be used as a means to include further constraints, meant to pragmatically reduce the search space and the potential number of candidate flaws, depending on the particular needs of the person performing the analysis.

For example, in the case of the design flaw "schizophrenic class", we could choose the first indicator as the initial filter, since it can be regarded as a characteristic feature of the pathological structure. In addition, we can decide to impose a further constraint on the elements of the search space (classes in our case), namely to exclude all trivial classes from the analysis. Thus, the final form of the initial filter could look the following: "*Any non trivial class, that is not cohesive with respect to data use of its method*". Defining what trivial classes are, and expressing this heuristic formally as a detection rule is done as described in section 5.3.2. For example, the formalization of the initial filter with the help of a detection strategy could look like this:

$$InitialFilter = \big(TCC(c) < 0.15\big) \wedge \big(NOM(c) >= 4\big) \wedge \big(NOA(c) >= 3\big)$$

where TCC represents the tight class cohesion metric [BK95], NOM represents the number of methods defined by the class c, and NOA represents the number of attributes defined by the class c.

The automated method presented above is an attempt to automate the matching of the pathological structure as well as the underlying design intent, using heuristics. As explained in section 5.2, the diagnosis process also contains a third step, which consists of evaluating the strategic closure for the analyzed fragment and matching it against the one described in the design flaw specification. In addition, design intent can sometimes be elusive and therefore hard to capture reliably in indicator definitions. For these reasons, the output of the automated part of the diagnosis should not be regarded as fail proof. Consequently, the third and final phase of the diagnosis process is interactive. In order to support the maintainer in this phase, we formulate a number of yes/no questions for each design flaw. The analysis tool will guide the maintainer through the last steps of the diagnosis process, by presenting him with the right questions for the candidate flaw under consideration. Furthermore, this feature can be integrated with state of the art code browsing and code visualization, in a wizard-like fashion, such that the user is always presented with only the information that is relevant for the decision at hand.

In the case of "schizophrenic class", our running example in this chapter, we formulated the following two questions for matching the design context and confirming that a candidate represents a real design flaw:

Question 1: *The maintainer must confirm that the suspected class captures at least two key abstractions that are not related through specialization, with data and functional methods of their own.*

Question 2: *The maintainer must confirm that the two or more design abstractions captured by the class are expected to change independently from one another. If the class in question represents a facade or is intended as a library of utility functions for a subsystem, the user must ensure that separating the individual abstractions enclosed in the class is meaningful and desirable.*

In section 4.2.4 of the previous chapter, we described a pattern form for specifying design flaws. We are now ready to extend this pattern with information that describes the diagnosis process of the design flaw. To this end, we will add a new section, called *diagnosis strategy*. A design flaw's diagnosis strategy is structured into the following four parts:

Search space: specifies the search space used in the automated detection of indicator instances. The search space specifies the type of structural elements that constitute potential inputs for the indicators, seen as filters.

Initial filter: defines the informal rule that is used to reduce the search space by filtering out those elements of the search space that are not considered relevant for the analysis.

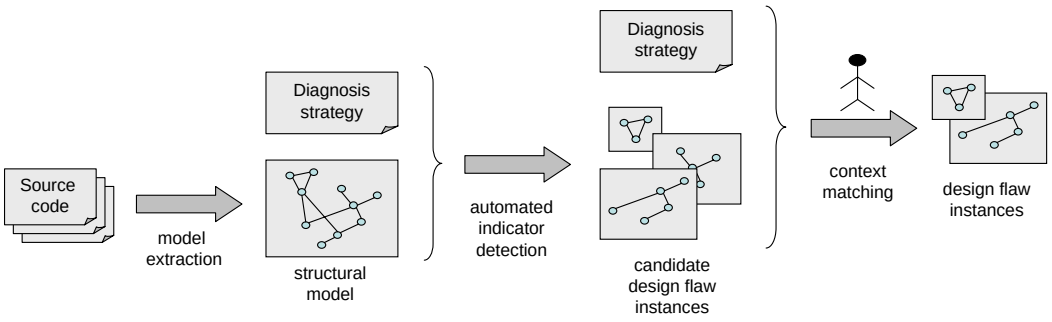


Figure 5.8.: Overview of the diagnosis process

Indicators: contains the list of informal indicator descriptions that are characteristic for the design flaw, as described in 5.3.1.

Context matching: contains the set of questions that are used interactively in order to match the underlying design context with the one described in the specification.

In conclusion, the current chapter presented a method for tool supported diagnosis of design flaw instances. As summarized in figure 5.8, the method has three main steps. In the first step, a structural model of the system is built, based on the source code. In the second step, using existing rule based pattern matching techniques, the structural model is investigated in order to find relevant combinations of indicator instances, that characterize various design flaws. Both the first and the second step run in a fully automated fashion, and result in a list of suspects. In the third and last step, the maintainer confirms or rejects each suspect in an interactive fashion, based on a set of questions that the user must answer. The questions posed by the system are meant to ensure an exact match of the fragment's design context, and are predefined for each type of design flaw. In the process of answering these questions, the maintainer uses integrated visualization, code browsing and cross referencing features.

Chapter 6.

A Design Flaw Based Restructuring Process

The previous chapters introduced a novel method that replaces classic approaches to problem detection and problem analysis with an integrated, systematic diagnosis process. Compared to the classic approaches, our method is designed to guarantee a causal treatment of existing structural deficiencies, while simultaneously increasing the level of automation. This corresponds to the goals that have been set out in section 1.1.3 of the thesis.

The purpose of the present chapter is twofold. First, we complete the cycle of the restructuring process, by addressing conceptional and implementation related issues of its concluding, reorganization phase. In the latter part of the chapter, we discuss the concept of a design flaw based restructuring process.

6.1. Closing the Circle

6.1.1. Reorganization Strategies

By definition, a design flaw is a mismatch between the features of a specific "is" structure, and the imperatives associated to the fragment's design context, as embodied by the reference structure. The imperatives embodied by the reference structure render it optimal with respect to the maintainability of the fragment.

The unambiguous mapping between the deficient "is" structure and the optimal "should be" structure means that further problem analysis, beyond the successful diagnosis of a design flaw, is no longer necessary. Thus, diagnosis spans both the classic problem detection and analysis activities. However, compared to previous methods, our method describes a systematic process, thus relying less on the intuition and personal experience of the maintainer.

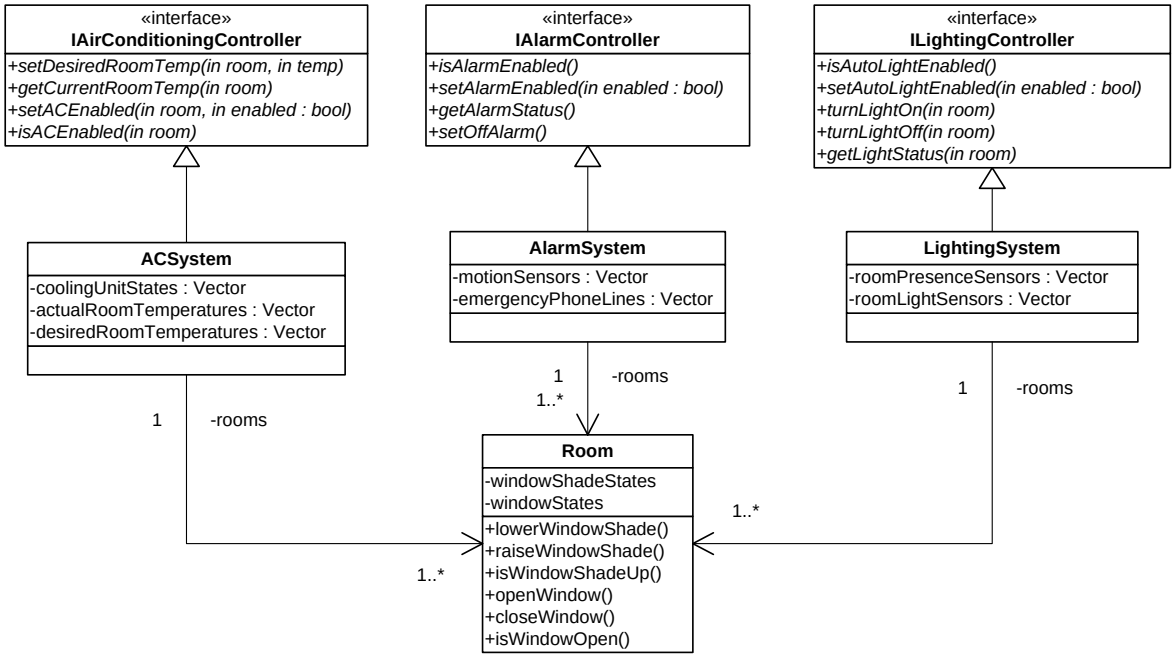


Figure 6.1.: Solution structure for the example in figure A.16

Given a successful diagnosis, the elimination of the design flaw is straightforward, and is achieved by transforming the source code so that it matches the given reference structure. The procedure which describes the steps of this transformation process forms the so called *reorganization strategy* of the design flaw.

In chapter 4, we distinguished between two kinds of design activities: modeling and structuring. Modeling involves identifying the key actors and their responsibilities in the domain, problem and solution space of the application, and results in an abstract model which translates into a design intent. In general, for any non-trivial application, modeling may result in several possible design intents, depending on factors such as the level of granularity at which the domain, the problem and the solution are modeled, and the topology of the design. Irrespective of the programming language used, the topology can be either action oriented or object oriented (see [Rie96]). Once a design intent takes shape, structuring activities are responsible for giving it expression, in accordance with the principles, rules and best practices of a given programming paradigm and language. As explained in section 4.2.3, design flaws represent deficiencies that occur during the structuring activity. They do not address potential deficiencies of modeling activities. Therefore, a reorganization strategy is elaborated under the assumption that the underlying model, and thus the design intent, is adequate with respect to the requirements of the application.

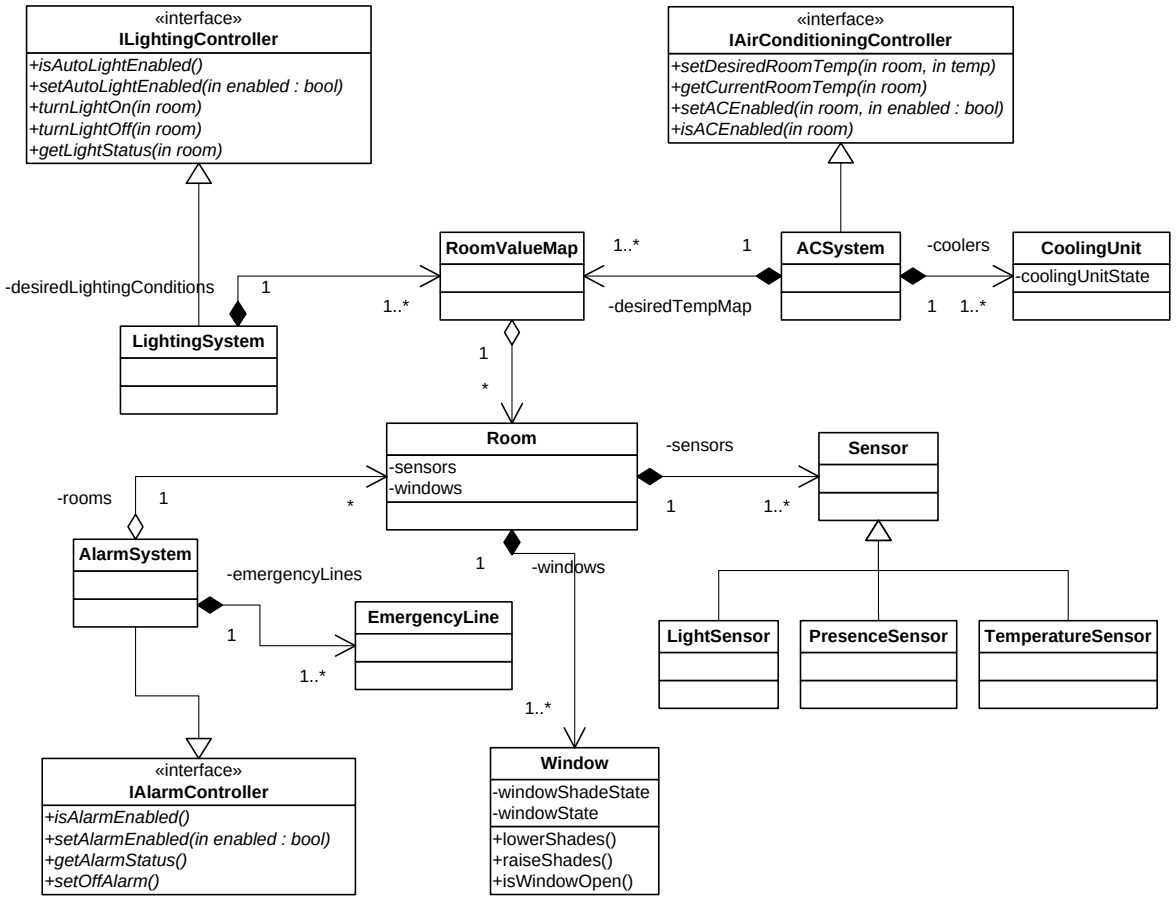


Figure 6.2.: Alternative design for the fragment in figure A.16

The best illustration of this idea is perhaps in the case of the design flaw “schizophrenic class” (A.6), introduced as a running example in section 5.1. A “schizophrenic class” is a class that captures two or more distinct (i.e. not related through specialization) design abstractions, as exemplified by figure 5.1.

Let’s look at this example a little bit closer. By looking at the existing design described by the UML diagram, we immediately identify three unrelated abstractions, encapsulated within the schizophrenic class `SmartHomeManager`, as suggested by the three interfaces. These abstractions correspond to the air conditioning system, the alarm system and the lighting system of a house. We also see that the topology of the underlying abstract model is object oriented. In other words, the three encapsulated abstractions are described by relatively isolated clusters of data and methods in the schizophrenic class. This is a requirement of the design flaw specification. Thus, if the topology of the underlying model were action oriented, we would not have a schizophrenic class.

The reorganized structure, corresponding to the reference structure of the design flaw, looks like the one in figure 6.1. As shown in the figure, the three abstractions have been separated into individual classes, based on their identities and implemented interfaces. Some of the original data and associated behavior could not be unambiguously assigned to one of the new classes. Thus, they form a fourth, utility class that has been named `Room`.

Let us now consider the alternative structure, shown in figure 6.2. Wouldn't this structure be a more adequate solution? In order to answer this question we must observe that figure 6.2 does not merely depict an alternative structure to the same structuring problem. Rather, it depicts a completely different design, because it has a different underlying model. Figure 6.2 corresponds to a much more fine grained original model than that which transpires from figure 5.1. Thus, figure 6.2 would actually constitute a complete redesign of the fragment, with both renewed modeling and structuring. The reference structure and the reorganization strategy, rely on the assumption that the underlying abstract model (including its topology) is appropriate and should not change. Thus, from the standpoint of the design flaw, the structure shown in figure 6.2 is not an alternative to the pathological structure, but the structure of a different, more fine grained underlying model.

The description of the reorganization strategy takes the form of an algorithm in pseudocode, and employs classic code refactorings such as those described in [Opd92, Rob99, Fow99]. The following represents the reorganization strategy for the design flaw "schizophrenic class".

1: Let O be the schizophrenic class
2: Check that we have an identity based decomposition of data in O, based on the identities of the encapsulated abstractions. // *An action oriented topology in the case of the encapsulated abstractions would require a complete redesign of the fragment, which is outside the scope of design flaws in general (see 6.1.1)*
3: Encapsulate all attributes in O with public accessors // *The public visibility is only temporary, in order to make moving functionality around easier*
4: Identify all the abstractions A_i, that need to be separated and establish their future interfaces
5: Create empty classes that correspond to each of A_i
6: **if** O has subtypes // *we assume that they contain valid specializations for one or more of the abstractions contained in O* **then**
7: Establish those abstractions from A_i that are affected by each specialization of O
8: Create appropriate subtypes for the classes that correspond to these abstractions
9: **end if**
10: **for all** attributes a_i in O **do**
11: Find the natural place for a_i in one of the newly created classes, including helpers,

based on the A_i determined before and apply "move field" [Fow99]
12: **if** a_i is an array or collection **then**
13: Decide between keeping the structured type and having an association multiplicity of 1 to the host class, or increasing the association's multiplicity and replacing the collection or array with only one of its elements. In the latter case, the class interface and the implementation of the facade need to be adapted accordingly
14: **end if**
15: **end for**
16: **for all** methods m_i in O **do**
17: **if** m_i can be unambiguously assigned to one of the new classes **then**
18: Apply "move method" [Fow99] to move m_i's body to the respective class
19: **if** m_i is specialized in one of O's subclasses **then**
20: Apply "move method" [Fow99] to move the overriding method into the appropriate specialization
21: **end if**
22: **else**
23: Apply "extract method" and "move method" [Fow99] to break up the original method, based on the attribute clusters that determine the encapsulated abstractions, and reunite functionality with its associated data
24: **if** m_i was previously specialized in one of O's subclasses **then**
25: Apply "extract method" and "move method" [Fow99] to break up the original overriding method, based on the attribute clusters that determine specializations of the encapsulated abstractions, and reunite functionality with its associated data
26: **end if**
27: **end if**
28: **if** m_i had public visibility **then**
29: Implement "facade" [GHJV96] method in O, delegating to the appropriate abstraction(s).
30: **end if**
31: **end for**
32: Create initialization methods in the facade O, or adapt its constructors to instantiate and wire together all newly defined classes and their specializations
33: Reduce data and accessor visibility as much as possible in all of the newly created classes

A first observation is that reorganization strategies involve high level transformations, such as the introduction of design patterns (e.g. the design pattern "facade" in step 16). High level, complex refactorings can be expressed as combinations of basic refactorings. For a detailed discussion on the theory and practical use of this method, we refer the reader to the results of [Zim97, SGMZ98, Rob99, Tok99, Cin00, Gen04, Ker05].

Secondly, because of the necessity of human intervention, a reorganization strategy is more than a high level refactoring. For example, establishing the interfaces of the contained abstractions in step 4 of the strategy, requires interaction and exploration capabilities, which are characteristic for a modern integrated development environment. Since the original Smalltalk refactoring browser [RBJ97], several tools have emerged that combine advanced refactoring capabilities with advanced code browsing and user interaction capabilities (e.g. Eclipse[1], IntelliJ IDEA[2], Inject/J [Gen04], SISSy Advanced Refactoring Wizard [Tri06]). These tools face and solve very similar challenges to a tool for implementing reorganization strategies, and prove that a highly effective platform for carrying out such complex transformations is part of today's state of the art.

Thirdly, all reorganization strategies given in our design flaw catalogue resort to basic object oriented building blocks, and are therefore language neutral. Thus, the implementation of tool support may require adapting a reorganization strategy, according to the features of the particular programming language and meta-model.

Finally, reorganization strategies, as described in the present work, rely on the assumption that the structural elements involved in the reorganization are not affected by other design flaws. While most of the times this is actually the case, we may have situations in which several design flaw instances may overlap, or their reorganization strategies may otherwise interfere. This particular situation is addressed in section 6.2.1, below.

6.1.2. Restructuring Patterns

By putting together a design flaw's specification with its corresponding diagnosis and reorganization strategies, as in the catalog described in appendix A, we obtain what we call a *restructuring pattern*.

Just like various other pattern languages, restructuring patterns describe reoccurring problems and the core of their solution, in a particular domain of human activity, in our case object oriented restructuring. They are a compact way of communicating best practice in this field.

Restructuring patterns have the following merits:

Tuned on restructuring: restructuring patterns have been designed with restructuring in mind. They are a means of recording and communicating restructuring know-how.

Causality: through diagnosis, restructuring patterns help us go beyond common structural anomalies and code smells, and address the cause that induced them. In addition, diagnosis takes into account the types of changes expected to occur to the fragment. Thus, we can deduce the correcting measures that are meaningful for the

[1] The Eclipse Website: `www.eclipse.org`
[2] The IntelliJ IDEA Website: `http://www.jetbrains.com/idea/`

given design fragment, and guarantee an in-depth rather than cosmetic improvement of the structure.

Systematic process: restructuring patterns take a significant part of the decisional burden off the shoulders of maintainers, by guiding them throughout the entire decision making process involved in restructuring. Thus, the maintainer's role and responsibilities are subject to a transformation that is similar to the transformation that occurred in aviation, from the days of early flight pioneers to modern commercial airlines. Where early flight pioneers used to depend on their personal skills and experience, modern pilots use standardized procedures and checklists.

Automation: the systematic nature of the process based on restructuring patterns, allows significantly higher levels of total or partial automation, which decreases costs and increases reliability. In terms of the above analogy, the standard procedures and checklists previously used by a pilot, can now be implemented as functions of the autopilot. Thus the executory role of the maintainer diminishes, while a supervisory role becomes increasingly dominant.

In the following, we look at how restructuring patterns relate to other pattern languages, relevant for our discussion:

Design patterns [GHJV96]: are formulated form the viewpoint of forward engineering, and do not describe an a-priori structure (i.e. the pathological structure). Thus, a design pattern can help a savvy maintainer in the problem analysis and reorganization phases, but a restructuring pattern offers more. It describes a comprehensive, step-by-step process, that spans an entire restructuring iteration, from problem detection until its elimination through code transformation. In effect, restructuring patterns can be seen on a meta-level, in the sense that they employ and describe the use of ordinary design patterns, within the context of restructuring.

Anti-patterns [BMMM98]: describe common mistakes, and are in this respect analogous to our design flaws. However, anti-patterns have a very broad scope, ranging from management to architectural issues and coding practice, which hinders direct tool support. On the other hand, restructuring patterns are tailored for the restructuring process, and provide all the ingredients that are necessary for a tool supported identification and removal of design flaw instances.

Reengineering patterns [DDN03]: apply the pattern format in order to document best practices in the general context of reengineering. Potential future change of the design is not considered, and reengineering patterns are symptomatic, in the sense that there are several patterns that deal with facets of the same problem (e.g. "move behavior close to data" and "eliminate navigation code") and others that mix more than one problem (e.g. "split up god class", which is a mixture between the problems of class schizophrenia and a deficient distribution of responsibilities). As in the case

of anti-patterns, reengineering patterns are intended to be read by humans, and there is no mechanism to allow their automated utilization.

Refactorings to patterns [Ker05]: are presented in a pattern-like form, and focus on the mechanics of implementing a design pattern in a place where it is missed. However, there is no mechanism to diagnose such situations. Furthermore, in the tradition of Fowler's refactorings, many refactorings to patterns are symptomatic in the sense that they address situations where the actual problem is not clear, and the maintainer is faced with decisions between alternative refactoring routes, along the way. For example, the refactoring "replace conditional logic with strategy" discusses the strategy design pattern and the creation of subtypes as alternative solutions to the same problem. This is due to the fact that the diagnosis of a real problem in the sense of causality is not addressed.

Disharmonies [LM06]: represent distortions that constitute violations of the structural properties of a "harmonious design", in effect, code smells. Disharmonies have a pattern-like presentation, and are organized in three clusters that correspond to object identity, object collaboration and object classification (i.e. inheritance). The process of employing disharmonies follows the classic decomposition of the restructuring process into three steps: detection, analysis and reorganization. Thus, each disharmony has a formally defined detection strategy, but problem analysis is not systematic, due to the symptomatic nature of the disharmonies, which prevents the specification of a clear reorganization strategy. In contrast, restructuring strategies rely on a causal, systematic diagnosis process, that encompasses both detection and analysis, and is less dependent on the maintainer's own judgement. Diagnosis also takes future change into account and results in a named design flaw, whose elimination can be described more easily and precisely. Thus, restructuring patterns allow for a higher level of automation throughout the restructuring process.

6.2. Design Flaw Based Restructuring

At this point, restructuring patterns provide us with the main elements that we need in order to put together a restructuring process based on design flaws. But before we can do that, we need to address the issue of design flaw interference.

In our previous chapters, we silently assumed the ideal case, in which design flaw instances appear isolated from one another, and can therefore be dealt with separately. While this may represent the more common case, it is conceivable that two or more design flaw instances or their corresponding reorganization strategies might affect the same design entities. An in-depth examination of the problems of design flaw interference con-

stitutes future work. However, the following section contains some brief considerations on the topic.

6.2.1. Considerations on Dealing with Design Flaw Interference

We say that two design flaw instances interfere, if they overlap, or if their reorganization strategies affect one or more common design entities. Figure 6.3 shows a design fragment, adapted from a real telecommunications system, and affected by interfering design flaws.

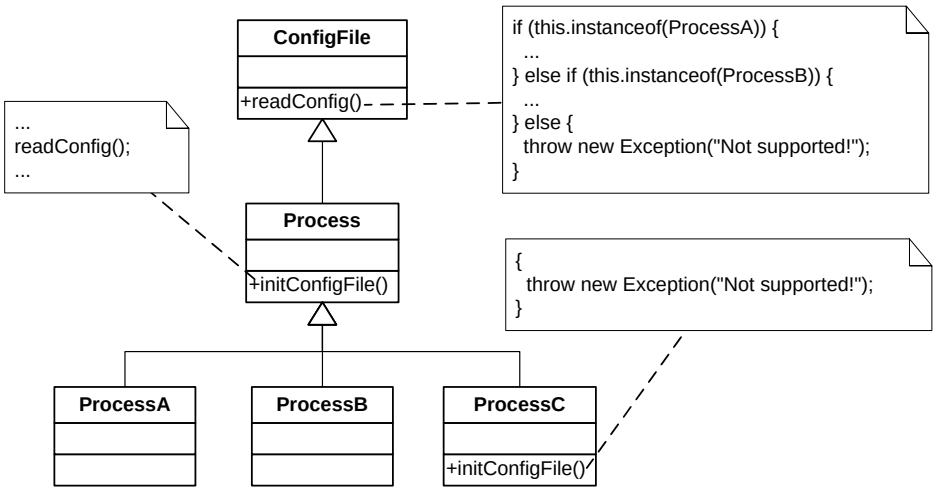


Figure 6.3.: Example of design flaw interference

The system models various types of processes as derived classes of the superclass `Process`. Most but not all process classes represent processes that are configurable through special configuration files. The concept of configuration file is represented by the class `ConfigFile`, which among other things, provides the functionality for parsing the various file formats. `Process` extends the class `ConfigFile` for the purpose of reusing its code, however it is obvious that the inheritance relation between the two types does not represent a semantically valid specialization. In order to provide each process instance with the appropriate parsing functionality, the method `readConfig()` uses runtime type identification to check the identity of the current object instance, in a complex conditional construct. To top it off, one of the three processes is not configurable, as indicated by an explicit refusal of the method `initConfigFile()`, inherited from its parent.

Obviously, the fragment shown above has several problems. First of all, we have an instance of “containment by inheritance” A.8, between `ConfigFile` and `Process`.

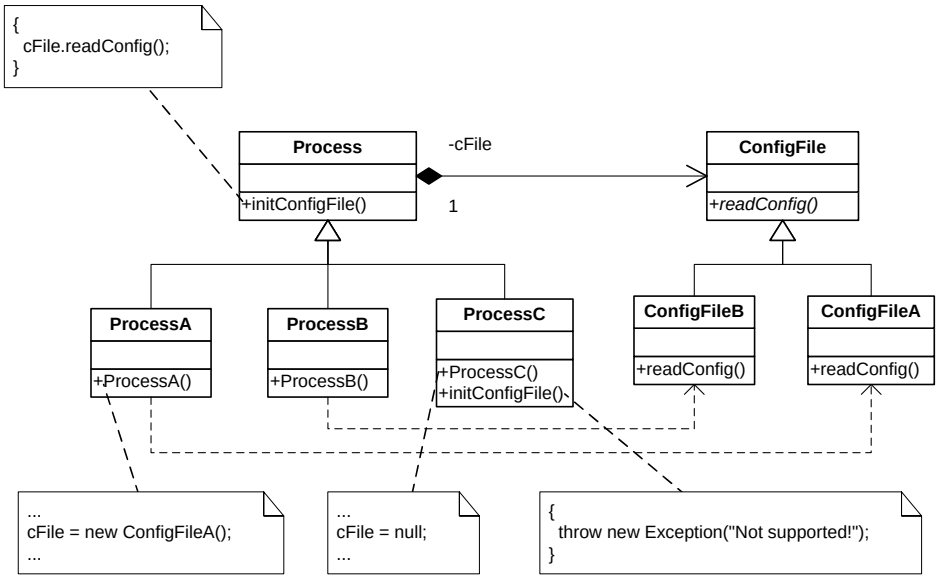


Figure 6.4.: Structure resulting from the elimination of "containment by inheritance"

Second, the process hierarchy suffers from "premature interface abstraction" A.9, because `Process` defines the method `initConfigFile()`, which is then refused by `ProcessC`.

Finally, we have the supertype `ConfigFile`, that depends on its subtypes `ProcessA` and `ProcessB`. At first sight, we would be tempted to see an instance of the design flaw "collapsed method hierarchy" A.10 in this case. However, according to the flaw's definition, the involved class hierarchy must be a valid inheritance hierarchy, which it's not, because of the invalid specialization between `ConfigFile` and `Process`.

This contradiction results from the way in which we defined the context of our design flaws. The context description must be as precise as possible, in order to guarantee the causality criterion. In other words, we want to be able to describe a reorganization strategy that is certain to be meaningful for the situation at hand. For this reason, although `readConfig()` collapses a hierarchy of code specialization in itself, it is not a valid instance of the design flaw, because part of the flaw's context does not match the concrete situation.

As it turns out, the reorganization strategy of the design flaw "containment by inheritance" A.8 contains provisions that handle the situation in which the supertype depends on one or more of its subtypes, so this case would be handled correctly when reorganizing the fragment (see figures 6.4 and 6.5).

However, there may be situations, in which we may have design flaws that are present

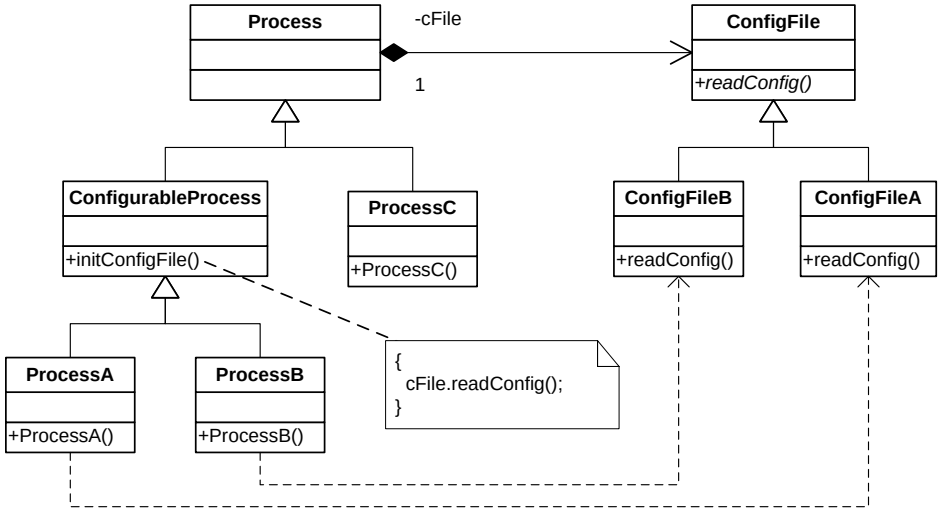


Figure 6.5.: Structure resulting from the elimination of "collapsed method hierarchy"

in an incipient form, and thus cannot be confirmed as such, unless some other interfering flaw is first removed. Such a situation is depicted in figure 6.6, where an instance of "schizophrenic class" A.6 in class `SmartHomeManager` prevents the incipient instance of "misplaced control" A.5 between `RemoteControl` and `SmartHomeManager` to be detected, unless the schizophrenic class is first dealt with. In addition, there may even be situations in which an incipient design flaw instance is not even detectable at all, until another instance's full removal. This would for example be the case of a class that reuses the code of another class by inheriting from it (i.e. "containment by inheritance" A.8). If in addition, the former subclass contained code that operated on attributes defined in the former superclass, after removing the original design flaw we would automatically induce another design flaw: "misplaced control" A.5.

Thus, a potential method for the integrated treatment of both confirmed and incipient design flaw instances is complicated even more, because it would have to take all of these cases into account. We leave the full investigation of these issues as future research, and limit the discussion in the present work, to a simpler but nevertheless valid approach, that is based on iterativeness.

The idea is to perform a diagnosis on the entire system, and then eliminate everything that can be eliminated, in accordance with the idealized reorganization strategies. After the reorganization is complete, diagnosis is repeated for all elements that have been newly created, or otherwise affected by the previous reorganizations. This iterative process goes on until no more design flaw instances are diagnosed.

For the example depicted in figure 6.6, this would happen as follows. In the first itera-

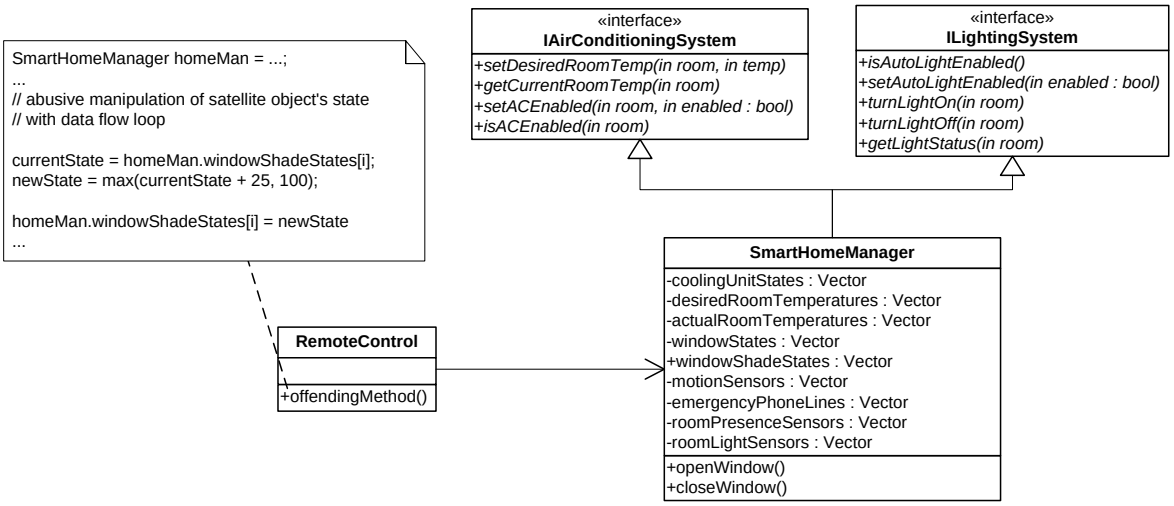


Figure 6.6.: Another case of interfering design flaw instances

tion, we would only diagnose an instance of “schizophrenic class” A.6, and the incipient “misplaced control” A.5 would be ignored. By breaking up the schizophrenic class, as described by the reorganization strategy, we obtain valid classes that contain subsets of the original set of attributes. By invalidating the diagnosis results for the newly created classes, we open up the possibility of diagnosing the previously ignored instances of “misplaced control” in a subsequent iteration. Since all classes involved in the candidate design flaw instance would now be valid classes, diagnosis would succeed. Following the removal of all instances of “misplaced control”, and the invalidation of affected entities, a subsequent diagnosis run would not find any more design flaws, and the process ends.

There are at least two possible ways of optimizing the iterative process suggested above. First, we can attempt to establish precedence rules between the various types of design flaws, based on the conditions posed by their context descriptions. Thus, we could say for instance that diagnosis and treatment of “schizophrenic class” A.6 should have precedence over “misplaced control” A.5, because a successful diagnosis of the latter requires the absence of the former. Such a requirement can also be justified intuitively, if we consider that in order to be able to discuss about the distribution of responsibilities between concepts, we first have to clearly describe these concepts as distinct classes.

The second type of optimization, concerns the diagnosis phase itself, and consists in reducing the number of candidate flaws that the maintainer has to look at, depending on the presence of certain others. Concretely, if we had the situation depicted in figure 6.6, confirming the schizophrenic class would automatically infirm the candidate instance of misplaced control. Thus, the maintainer would not even see the potential misplaced control as a candidate instance anymore.

An in-depth investigation of the topic of precedence hierarchies for the purpose of dealing with interfering design flaw instances is left as future research.

6.2.2. Overview of Proposed Restructuring Process

Since in real life, a working system is always changed out of necessity, any restructuring process is based upon a set of underlying reengineering goals that are a translation of this necessity. Possible reengineering goals may be:

- Fixing functional bugs
- Implementing new features
- Decreasing system fragility[3]
- Migrating the system onto a new platform or technology
- Splitting a monolithic system into separately maintainable or marketable parts
- Reusing parts of the system in a different project
- Integrating heterogeneous pieces to form a new system

Restructuring generally has the task of preparing the source code, or parts of it, in order to facilitate the implementation of the underlying reengineering goals. This translates into increasing the understandability, extensibility and flexibility of the code.

For the rest of the discussion we assume the existence of a tool, that supports diagnosis and reorganization activities, as described in the past and current chapters. The tool operates on an abstract structural model of the system, as described in section 5.4.

With the considerations above, the structure of the proposed tool supported restructuring process, based on design flaws, is as follows (see figure 6.7):

Step 1: Based on the primary reengineering goals, delimit system parts to be restructured. Add all corresponding structural model elements to the diagnosis queue.

Step 2: Build up an understanding of the types of future change that are reasonably probable to occur to the parts of the system that are to be restructured. This step is important in order to allow the maintainer to decide on the strategic closures during diagnosis.

Step 3: Choose a design flaw mix that is appropriate with respect to the underlying reengineering goals and future changes of the system. Also of value at this stage, is the impression that maintainers may have about certain aspects of the system's design. For instance, a maintainer that has suspicions about certain inheritance hierarchies, may decide to include more design flaws that deal with inheritance problems into the mix. The catalogue in appendix A can be used as a starting point, because it offers examples that touch on all major aspects of object oriented design.

[3] the probability of unexpected failures due to changes in the system

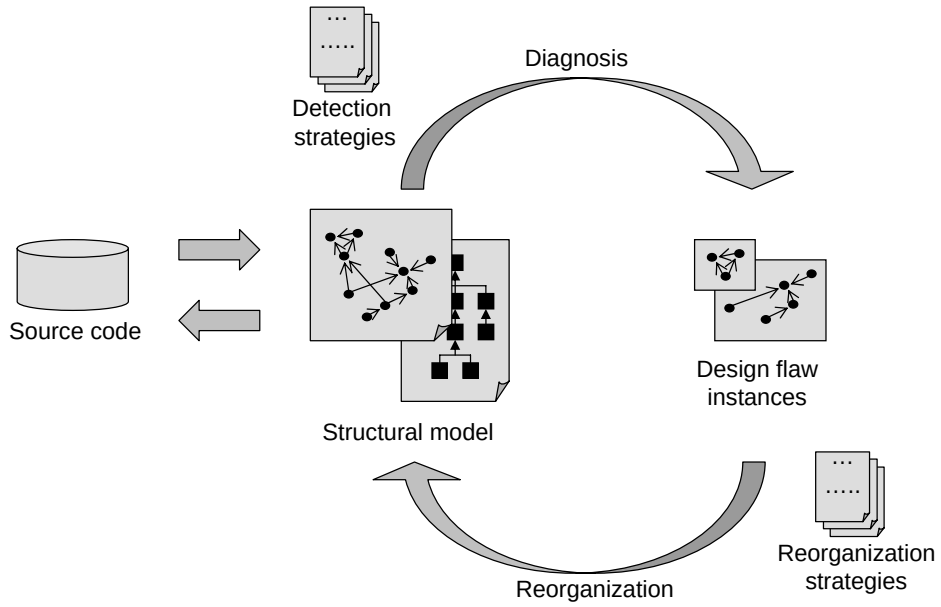


Figure 6.7.: Process overview

Step 4: Perform a diagnosis iteration on all model elements contained in the diagnosis queue, according to the method described in chapter 5. The first two fully automated phases of this step will result in a number of candidates that the maintainer can sort according to probability. Subsequently, the maintainer goes through the part or the entire list, and either confirms or rejects the match between design contexts in each case. As a result, we obtain a list of design flaw instances, which we add to the reorganization queue. The diagnosis queue is emptied.

Step 5: If the reorganization queue is empty, the process ends here.

Step 6: Carry out reorganization strategies for all design flaw instances in the reorganization queue. At the same time, add all changed or new model elements into the diagnosis queue.

Step 7: Empty reorganization queue and jump to step 4.

Chapter 7.

Evaluation

In this chapter, we evaluate the automated diagnosis method introduced in chapter 5. The chapter has two main parts. In the first part, we give a brief description of a prototype tool called CodeClinic, which implements our diagnosis method. The second part is dedicated to the actual evaluation, which consists of a series of case studies on intermediate to large sized software systems.

7.1. Prototype Tool

As illustrated by the screen shot in figure 7.1, CodeClinic is fully integrated with the Eclipse framework and IDE. The tool integrates seamlessly with the Java development tools, adding various elements to the graphical environment, and enables developers to have their projects scanned for potential design flaws in the background, while they code. Additionally, it is possible to restrict the analysis to individual packages and compilation units.

The candidates found are presented in two dedicated views. The design flaw list view, shown in the lower part of the screen, displays a categorized list of design flaw instances found in the code, and allows the engineer to jump to the affected code fragments, which are displayed and highlighted in the environment's main editor window. The second view, shown to the right of the main editor window, displays detailed information that is specific for each type of design flaw, about the currently selected flaw instance. In addition, this view presents the engineer with the sequence of questions that he needs to answer, in order to confirm or reject a given candidate.

The tool itself is designed as a framework that can be extended using the standard Eclipse mechanism of extension points. Figure 7.2 depicts a very abstract logical view of the system architecture. The Eclipse plugin that contains the actual application framework is shown on the left, and an extension plugin that defines a single design flaw is shown on the right. The shaded component depicted below the two plugins belongs to the Java Developer Tools (JDT) of the Eclipse platform, and represents the structural model of the java source code project that is analyzed.

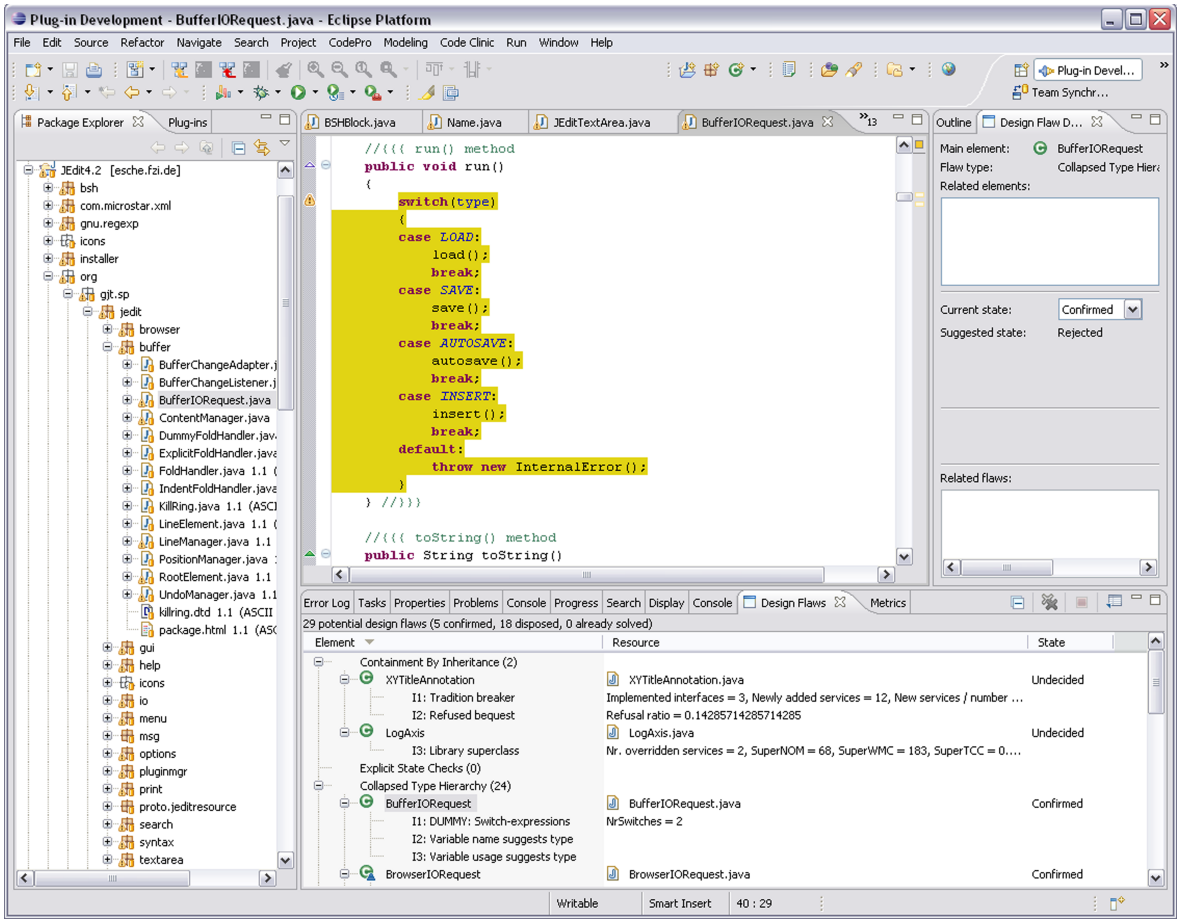


Figure 7.1.: Screen shot of CodeClinic

The main application is structured into three layers. The model layer holds the design flaw instances that are detected by the tool, together with information about the corresponding indicator presence and the current state of each instance. A design flaw instance can be in any of the three states: undecided (the initial state after detection), confirmed and rejected. The model persister is responsible for automatically persisting the entire model to disk upon closing the environment, and for restoring it back upon start-up. Thus, the tool supports the analysis of complex projects, across multiple sessions.

The presentation layer depicted in the middle, is responsible for acting as an intermediary between the data model and the various possible views, as well as for accepting and responding to user gestures. For example, it is the responsibility of this layer to initiate the diagnosis process in response to the user's corresponding action in the graphical environment. The automated part of the diagnosis process is coordinated by the detection

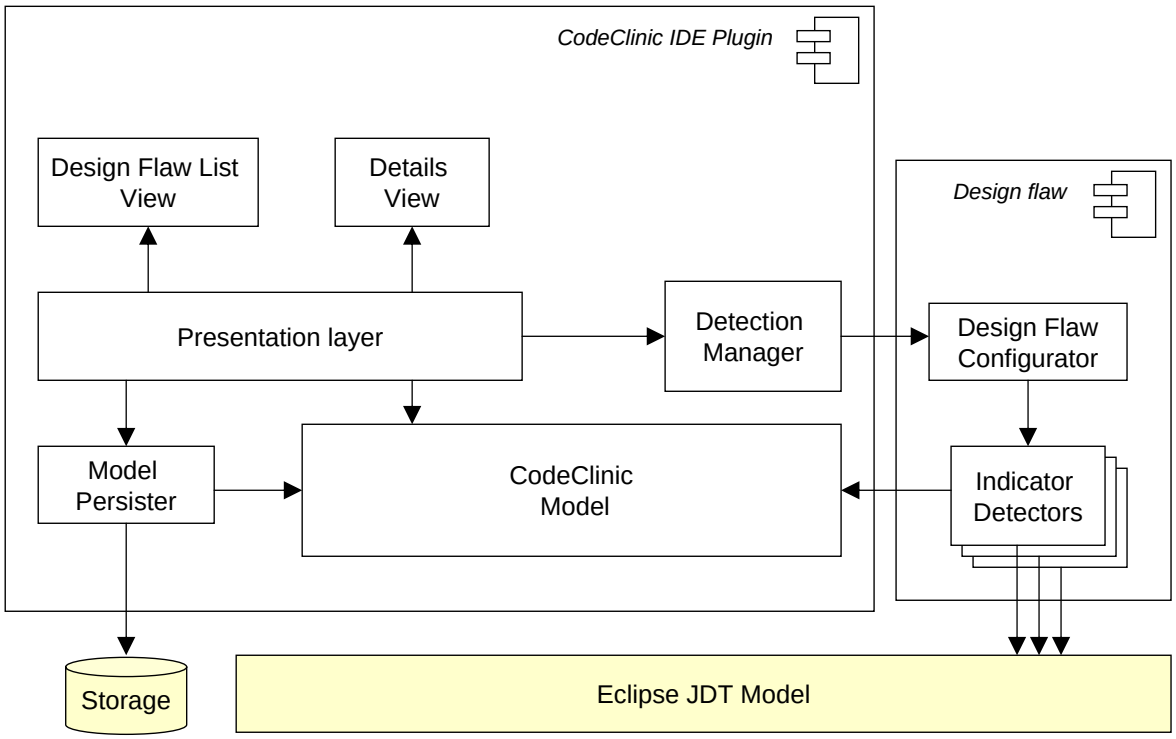


Figure 7.2.: Architectural overview of CodeClinic

manager, with whom all concrete design flaw implementations are registered. The system supports an arbitrary number of design flaws, which can be packaged as separate eclipse plugins, and dynamically deployed on top of an existing installation.

Figure 7.2 depicts the interactions that take place between the main application and one design flaw implementation. The design flaw configurator registers itself with the detection manager and configures the individual indicator detectors with appropriate parameters and metrics thresholds, which are configured by the user via the standard preferences dialog of the Eclipse platform. During analysis, individual indicator detectors receive access to the underlying structural model of the analyzed project, as well as to the CodeClinic model.

Finally, the top layer is represented by the views. The tool currently offers the two views mentioned above: the design flaw list view and the details view. In addition, for each candidate instances, a separate entry in the warnings list of the platform standard “Problems” view is added.

The architecture and the user interface of the tool are designed to allow an easy extension with further analysis features such as code visualizations, as well as the implementation of

an advanced refactoring assistant. Using the refactoring capabilities of Eclipse, the refactoring assistant would guide the user through the reorganization process of each design flaw. Such a reorganization assistant has been developed prototypically [Tri06], but has not yet been integrated with CodeClinic.

For the purposes of the evaluation, we chose a set of five design flaws from our catalog: "containment by inheritance" A.8, "explicit state checks" A.3, "collapsed type hierarchy" A.1, "schizophrenic class" A.6 and "misplaced control" A.5. The chosen set of design flaws is relevant for the following reasons:

- All five design flaws capture situations which can be frequently encountered in large legacy systems, irrespective of their domain or implementation flavor.
- The set of chosen flaws is representative, because they address a representative set of common problems in object oriented design: deficiencies pertaining to conceptualization and class definition ("schizophrenic class"), to distribution of responsibilities and cooperation between abstractions ("misplaced control"), to the use of the inheritance relation ("containment by inheritance"), as well as the unjustified absence of object oriented features, typical for procedural style programming ("collapsed type hierarchy" and "explicit state checks").
- The chosen set contains design flaws whose definitions are inspired both by well known structural anomalies (e.g. god class, feature envy, data class), and by design patterns (e.g. state, template method).

In addition, as will be explained in section 7.2.1, the choice of four of the design flaws mentioned above has also been motivated by the fact that they describe different causes for the appearance of two well known code smells: "god class" and "switch statements". Thus, we here especially interested to see if our method is able to distinguish correctly between the different cases.

In the following, we present relevant implementation details for the design flaws used in the evaluation. The various indicator definitions have been implemented to work in the context of the programming language Java.

Indicators that needed concretization in the form of metrics based rules, have been expressed using detection strategies, as described in 5.3.2. In general, metrics thresholds used in indicator detection rules are configurable, so they will be given as symbolic constants, along with default values. Most of these defaults are borrowed from [LM06], and are the result of statistical analyses over a large number of case studies. However, in order to improve the detection of some indicators, we decided to adjust some of the thresholds used in our evaluation to better match the characteristics of each of the analyzed systems. Further details on this matter along with the actual threshold values used in the evaluation are presented in section 7.2.2.

7.1.1. Containment by Inheritance

Search space

All pairs of classes.

Initial filter

Definition: *Classes must be related through a direct inheritance relation.*

Implementation: as defined.

Indicator 1

Definition: *The derived class is a "tradition breaker" in the sense that it adds a significant amount of new methods, or it extends several supertypes.*

Implementation:

```
I1 := NAS(s) >= MinNAS and PNAS(s) >= MinPNAS or
      NII(s) >= MinNII
```

where `s` represents the subclass, `NAS` and `PNAS` represent the number of added services and the percentage of newly added services in the subclass, as defined in [LM06], and `NII` represents the number of interfaces implemented by the subclass.

Default thresholds:

```
MinNII = 2
MinNAS = 6
MinPNAS = 0.66
```

Indicator 2

Definition: *The presence of "refused bequest" (interface form) [Fow99] in at least one method that is part of the inherited public interface of the base class. There are two possible scenarios. In the first one, the derived class employs private or protected inheritance. In the second scenario, the derived class employs public inheritance but overrides the inherited public interface with methods that are either empty, are limited to throwing exceptions, or returning an error code.*

Implementation:

```
I2 := NRO(s) >= MinNRO
```

where `s` represents the subclass and `NRO` represents the number of refusing overrides, as described in the definition of the indicator.

Default thresholds:

```
MinNRO = 1
```

Indicator 3

Definition: *The base class stores library code. We have the following two possibilities: either it is intended as a library for arbitrary classes, or the subclass uses it as such. In the first case, the base class is large and complex, yet has a very low internal cohesion. In the second case, the superclass is not simple (i.e. it has a certain minimal size and complexity), yet the subclass does not override any of the inherited services.*

Implementation:

```
I3 := NOM(S) >= MinNOM1 and WMC(S) >= MinWMC1 and
      TCC(S) <= MaxTCC or NOM(S) >= MinNOM2 and
      WMC(S) >= MinWMC2 and NOS(s,S) <= MaxNOS
```

where `s` represents the subclass, `S` represents the superclass, `NOM` represents the total number of methods of a class, `WMC` represents the weighted method count metric as defined in [LM06], `TCC` represents the tight class cohesion metric, and `NOS` represents the number of overridden services in `s`, inherited directly from `S`.

Default thresholds:

```
MinNOM1 = 10
MinWMC1 = 31
MaxTCC = 0.2
MinNOM2 = 7
MinWMC2 = 14
MaxNOS = 0
```

7.1.2. Explicit State Checks

Search space

All classes of the system.

Initial filter

Definition: *A non trivial class that contains at least two `switch` or equivalent `if-else` constructs, located in separate methods, not using runtime type identification, on the same specific class attribute, which is not declared as final.*

Implementation: as defined.

Indicator 1

Definition: *The name of the checked attribute contains the word "state", thus suggesting a state variable. Alternatively, the switch parameter is compared against a set of symbolic constants whose names contain such a word.*

Implementation: as defined.

Indicator 2

Definition: *Usage patterns of the switch parameter suggest a state variable, in the sense that the value of the checked parameter is changed, either within branches of the conditional constructs, or from the clients that called the respective operation.*

Implementation: as defined.

7.1.3. Collapsed Type Hierarchy

Search space

All classes of the system.

Initial filter

Definition: *A non trivial class that contains at least two* `switch` *or equivalent* `if-else` *constructs, located in separate methods, not using runtime type identification, either on the same specific class attribute or using formal parameters that have the same name.*

Implementation: as defined.

Indicator 1

Definition: *The name of the parameter used in the conditional expressions contains the word "type", thus suggesting a type variable. Alternatively, the parameter is compared against a set of symbolic constants whose names contain such a word.*

Implementation: as defined.

Indicator 2

Definition: *Usage patterns of the parameter used in the conditionals suggest a type variable. In other words, there is no write access on the variable, anywhere in the class, with the exception of object constructors.*

Implementation: as defined.

7.1.4. Schizophrenic Class

Search space

All classes of the system.

Initial filter

Definition: *Any non trivial class, that is not cohesive with respect to data use of its methods.*

Implementation:

```
IF := NOM(c) >= MinNOM and NOA(c) >= MinNOA and
      TCC(c) <= MaxTCC
```

where `c` represents the class, `NOM` represents the total number of methods, `NOA` represents the total number of attributes, and `TCC` represents the tight class cohesion metric.

Default thresholds:

```
MinNOM = 4
MinNOA = 3
MaxTCC = 0.2
```

Indicator 1

Definition: *The class defines a large number of attributes.*

Implementation:

```
I1 := NOA(c) >= MinNOA
```

where `c` represents the class and `NOA` represents the number of locally defined, non-static attributes.

Default thresholds:

```
MinNOA = 6
```

Indicator 2

Definition: *The class is heavyweight, in the sense that it is very large and has a high complexity.*

Implementation:

```
I2 := WMC(c) >= MinWMC
```

where `c` represents the class and `WMC` represents the weighted method count metric.

Default thresholds:

```
MinWMC = 40
```

Indicator 3

Definition: *The class constitutes a "bottleneck", in the sense that a significant proportion of all classes in the system depend on it. A class depends on another if it references it.*

Implementation:

```
I3 := NRC(c) >= MinNRC
```

where `c` represents the class and `NRC` represents the number of classes that reference `c`.

Default thresholds:

```
MinNRC = 6
```

Indicator 4

Definition: *The class exhibits distinct personalities with respect to disjoint groups of clients, either by explicitly implementing two or more non-trivial interfaces, or by having disjoint groups of clients that use disjoint fragments of the class' public interface.*

Implementation: as defined.

7.1.5. Misplaced Control

Search space

All methods defined in all classes of the system.

Initial filter

Definition: *Any non-trivial method, that is not a constructor or an accessor, and references a different type. Referenced types will be referred to as "satellites".*

Implementation: as defined.

Indicator 1

Definition: *The method accesses at least two foreign fields, either directly or through accessors. The accessed fields are declared in classes that are not superclasses of the one declaring the method.*

Implementation: as defined.

Indicator 2

Definition: *The method manipulates the internal state of another, unrelated class, by writing to one or more of its attributes, either directly or through a setter method.*

Implementation: as defined.

Indicator 3

Definition: *The method contains an information loop in the sense that information is pulled from instances of an unrelated class using accessors, transformed it in one way or the other, and the result is pushed back into the class either by writing to an attribute or as a parameter in a method call.*

Implementation: as defined.

Indicator 4

Definition: *Unrelated classes whose data are read or written to by the method, act as "dumb" data holders, by exposing a significant proportion of their data, either directly (i.e. public attributes) or through accessors.*

Implementation:

```
I4 := NEA(s) / NOA(s) >= MinExposure
```

where `s` represents the satellite class, `NEA` represents the number of exposed attributes, and `NOA` represents the total number of attributes.

Default thresholds:

```
MinExposure = 0.75
```

Indicator 5

Definition: *The method varies its behavior based on the identity of an object, using runtime type identification. In other words, it compares the type of a reference against an unrelated (i.e. through inheritance) type, adapting its behavior accordingly.*

Implementation: as defined.

7.2. Case Studies

7.2.1. Experimental Goals and Approach

For the validation of the proposed diagnosis method, we formulated the following experimental goals:

1. To check if the fully automated part of the diagnosis process, as described in chapter 5, works as intended, and is able to produce valid candidate flaws. This amounts to the following points:
   - Test the hypothesis formulated in section 5.2, that design intent manifests itself in the structure, and thus can be determined by an automated tool. Specifically, the design flaws "explicit state checks" and "collapsed type hierarchy" have been specifically chosen because of the similarity of their pathological structures, which corresponds to the code smell "switch statements" [Fow99]. Similarly, the two design flaws "schizophrenic class" and "misplaced control" capture complementary aspects of a single structural anomaly known in the literature as "god class" [Mar02], or "the blob" [BMMM98].
   - Test the hypothesis that design flaw diagnosis (at least the automated part) and medical diagnosis are similar in the sense that design intent is characterized by a unique combination of indicators, and that the precision of the diagnosis process tends to increase with the growing number of simultaneously detected indicators. This is of course just a tendency, since each of the indicators must be assigned a weight that reflects its relevance for the respective type of flaw.
2. To evaluate the quality of each indicator by computing a linear regression model in order to estimate its relevance for its respective design flaw, as described in section

5.3.3. At the same time, the weights obtained from the linear regression constitute a predictive model that is useful in future analyses. The assumption that is made here is that the specified indicators, which represent the explanatory variables in the regression, are statistically independent.

3. To compute the recall of the fully automated part of the diagnosis, by determining the real number of false negatives. For reasons of practicability, we decided to limit the scope of this task to only two design flaws: "explicit state checks" and "collapsed type hierarchy". The reason for choosing these two design flaws is that they have the smallest number of indicators from the mix that was selected for our experiments. Thus, we wanted to see if under these circumstances it is still possible to correctly detect most of, or all of the existing instances. Thus, we adapted our tool in such a way that it generates a list of all occurrences of the specific switch or if-else constructs that occur in two or more methods of any class. Subsequently, each of the reported cases was manually investigated.

As explained in chapter 5, the automated part of the diagnosis process only covers the detection of the pathological structure and design intent, and does not address the question of establishing the strategic closure for the given fragment. This is because strategic closure generally depends on momentary and planned requirements towards the application, and therefore can differ at various points in time. In other words, strategic closure depends on factors that are external to the system.

Consequently, we will ignore it in our experiments. Therefore, all precision values used in the validation have been computed by only taking into account the match between the actual and the specified pathological structure and design intents, and by ignoring strategic closure. This means that an instance will be considered as confirmed, if and only if the pathological structure and design intent match the specification of the flaw.

This way of computing diagnosis precision is in conformance with the set of experimental goals stated above.

7.2.2. Experimental Setup

We validated the proposed diagnosis method by employing the tool CodeClinic on a set of four intermediate to large-sized software systems, each representing a different application domain:

- *XUI (v1.04)*[1]*:* a Java and XML based framework for building desktop, handheld, mobile, web and enterprize applications with a rich user interface. XUI provides development and debugging tools, as well as a set of look and feel components, widgets, and database bindings. In addition, the framework offers support for declarative XML based user interface generation.

[1] http://www.xoetrope.com/xui

System	LOC	Nr. Types	Nr. Methods	Avg. Methods/Type	Avg. Fields/Type	Avg. WMC
XUI	47,750	547	3,593	6.56	3.12	20.04
JFreeChart	83,591	609	6,804	11.17	3.07	30.63
Inject/J	53,190	647	5,143	7.94	2.33	23.48
JEdit	87,479	859	4,877	5.67	3.26	24.85

Figure 7.3.: Size and complexity measurements for the analyzed systems

		Threshold	XUI	JFreeChart	Inject/J	JEdit
Containment by inheritance	Indicator1	MinNAS	6	10	6	5
	Indicator3	MinNOM1	7	11	8	5
		MinWMC1	19	27	21	21
		MinNOM2	10	16	12	8
		MinWMC2	40	60	46	49
Schizophrenic class	Indicator2	MinWMC	40	60	45	50

Figure 7.4.: Threshold values used during analysis

- *JFreeChart (v1.0.6)*[2]*:* a free chart library for the Java platform, that allows developers to display professional quality charts of various types in their applications. It supports a wide range of output types, including Swing components and several bitmap and vector based graphic file formats.
- *Inject/J (Aug 2005:)*[3]*:* is a meta-programming library for the JAVA programming language. The tool allows an interactive script based execution of code refactorings, and provides a rollback mechanism. An important factor in favor of choosing this case study was that the author had direct access to the development team.
- *JEdit (v4.2)*[4]*:* is a full featured highly customizable text editor, with syntax highlighting and other spcialized support for more than 130 languages. It has a built-in macro language and an extensible plugin architecture.

The table shown in figure 7.3 offers some general size and complexity characteristics of these systems. The measurements were made using the tool CodePro Audit[5], version 5.1.0.

As can be seen in the table, some metric values vary significantly across the individual systems. Therefore, in order to improve the detection of some indicators, we decided to

[2]http://www.jfree.org/jfreechart
[3]http://injectj.fzi.de
[4]http://www.jedit.org
[5]http://www.instantiations.com/codepro/analytix/about.html

replace default threshold values with ones that are better tailored for each individual system. The table shown in figure 7.4 provides an overview of the actual values used in the experiments. For example, we see that the average number of methods per type varies between 5.67 in the case of JEdit, and 11.7 in the case of JFreeChart. Consequently, the values of `MinNOM1` and `MinNOM2` used as thresholds in indicator 3 of the design flaw "schizophrenic class" had to be scaled accordingly. Such significant differences between values of the same metric may be the result of different sets of guidelines, design philosophies, or levels of competence in the respective development teams, and are a perfect illustration of the dangers of using thresholds that are statistically derived across a very large number of systems.

7.2.3. Discussion of Results

Experimental Goal 1

Figure 7.5 provides a condensed view of the statistics concerning our first experimental goal. Each of the individual rows in the given tables, provides the statistics for those candidate flaws for which a particular number of indicators had been detected by our tool. This number is given in the first column of each table. For example, for the system XUI, we have a total number of 5 candidate instances of the design flaw "containment by inheritance", for which CodeClinic detected exactly two out of the total of three indicators defined in the design flaw specification. Based on the manual inspection of each candidate, only 2 candidates out of the 5 were confirmed, which means a precision of 40%. The last column provides the precision computed over all four analyzed systems.

The figures in this table prove both hypotheses formulated in the first experimental goal in section 7.2.1. Concretely, our prototype implementation was able to correctly identify the design intent and the presence of the pathological structure in a total of 712, automatically detected and manually confirmed cases. This proves our first hypothesis, namely that design intent can at least in some cases be determined using an automated tool. The total precision over all five design flaws and four analyzed systems is around 33%.

In addition, as indicated by figure 7.6, we can clearly see a direct correlation between precision and the number of simultaneously detected indicators, although in most cases with a simultaneous decrease of recall. This is in accordance with the second hypothesis formulated for this experimental goal.

This drop in recall is illustrated by figure 7.7, which indicates the number of confirmed design flaw instances in relation to the number of simultaneously detected indicators. Interestingly, the design flaw "schizophrenic class" seems to represent an exception from this point of view. This tells us something interesting about this flaw's set of indicators. More exactly, it tells us that the indicators of "schizophrenic class" are statistically less independent than those of other design flaws, such as "misplaced control".

Containment by Inheritance

Nr. Ind.	XUI conf. / cand.	XUI precision	JFreeChart conf. / cand.	JFreeChart precision	InjectJ conf. / cand.	InjectJ precision	JEdit conf. / cand.	JEdit precision	Overall precision
1	34 / 119	28.6	49 / 123	39.8	5 / 52	9.6	2 / 22	9.1	28.5
2	2 / 5	40.0	45 / 66	68.2	4 / 4	100.0	1 / 4	25.0	65.8
3	1 / 1	100.0	10 / 10	100.0	0 / 0	—	0 / 0	—	100.0

Explicit State Checks

Nr. Ind.	XUI conf. / cand.	XUI precision	JFreeChart conf. / cand.	JFreeChart precision	InjectJ conf. / cand.	InjectJ precision	JEdit conf. / cand.	JEdit precision	Overall precision
1	1 / 1	100.0	1 / 2	50.0	2 / 5	40.0	6 / 16	37.5	41.7
2	0 / 0	—	0 / 0	—	0 / 0	—	0 / 0	—	—

Collapsed Type Hierarchy

Nr. Ind.	XUI conf. / cand.	XUI precision	JFreeChart conf. / cand.	JFreeChart precision	InjectJ conf. / cand.	InjectJ precision	JEdit conf. / cand.	JEdit precision	Overall precision
1	1 / 3	33.3	3 / 19	15.8	1 / 5	20.0	2 / 6	33.3	21.2
2	0 / 0	—	0 / 0	—	0 / 0	—	2 / 2	100.0	100.0

Schizophrenic Class

Nr. Ind.	XUI conf. / cand.	XUI precision	JFreeChart conf. / cand.	JFreeChart precision	InjectJ conf. / cand.	InjectJ precision	JEdit conf. / cand.	JEdit precision	Overall precision
1	1 / 16	6.3	2 / 9	22.2	3 / 19	15.8	4 / 27	14.8	14.1
2	9 / 16	56.3	3 / 12	25.0	5 / 11	45.5	2 / 15	13.3	35.2
3	10 / 16	62.5	11 / 19	57.9	7 / 9	77.8	4 / 13	30.8	56.1
4	3 / 4	75.0	14 / 17	82.4	3 / 3	100.0	4 / 5	80.0	82.8

Misplaced Control

Nr. Ind.	XUI conf. / cand.	XUI precision	JFreeChart conf. / cand.	JFreeChart precision	InjectJ conf. / cand.	InjectJ precision	JEdit conf. / cand.	JEdit precision	Overall precision
1	24 / 202	11.9	56 / 233	24.0	67 / 184	36.4	103 / 350	29.4	25.8
2	16 / 39	41.0	45 / 129	34.9	23 / 47	48.9	61 / 151	40.4	39.6
3	6 /9	66.7	14 / 17	82.4	5 / 11	45.5	28 / 73	38.4	48.2
4	0 / 0	—	0 / 0	—	0 / 0	—	7 / 8	87.5	87.5
5	0 / 0	—	0 / 0	—	0 / 0	—	0 / 0	—	—

Figure 7.5.: Overview of results for experimental goal 1

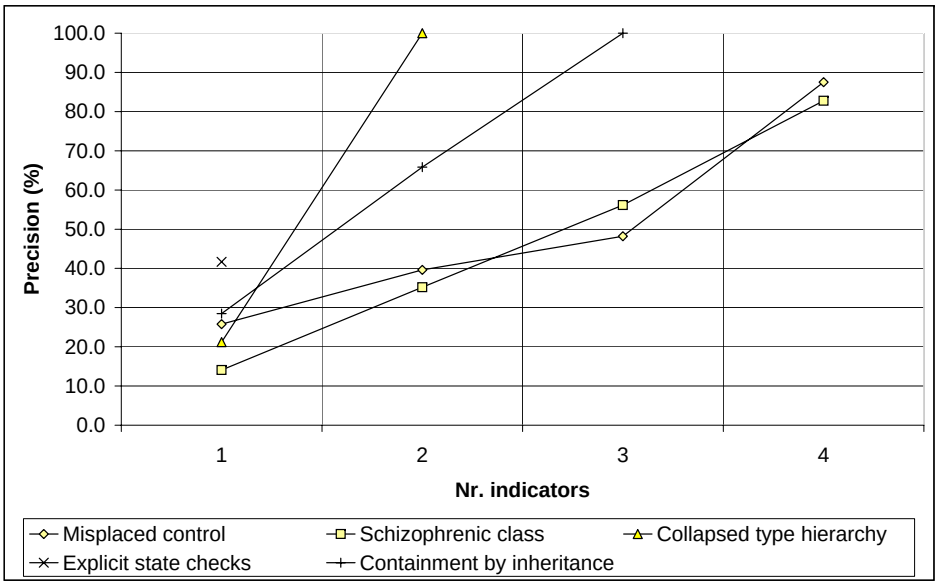


Figure 7.6.: Precision in relation to the number of indicators

Indeed, if we look at the respective specifications, we can see that the sets of indicators specified for the other design flaws (especially "misplaced control") often express alternative, even mutually exclusive manifestations in the structure. Unlike these cases, in the case of "schizophrenic class", the relation between the various types of indicators is much stronger, in the sense that the captured features are *more probable to occur together*, in any instance of the flaw. In other words, the individual indicators are not completely independent, and we expect this to affect the quality of the linear regression performed for this particular flaw, as part of our second experimental goal.

Therefore, the conclusion that one might draw from figure 7.7, namely that it is better to have indicator sets such as in the case of "schizophrenic class", is false. The goal should not be to reduce loss of recall by defining statistically dependent indicators, but instead to capture every conceivable manifestation of a given flaw, even if some of them are mutually exclusive.

Experimental Goal 2

Our second experimental goal was to evaluate the quality of each particular indicator with respect to the way in which it relates to the other indicators defined for a given flaw. In order to do this, we applied a simple linear regression on the set of observations made on the four analyzed systems. The obtained coefficients are given in figure 7.8, along with

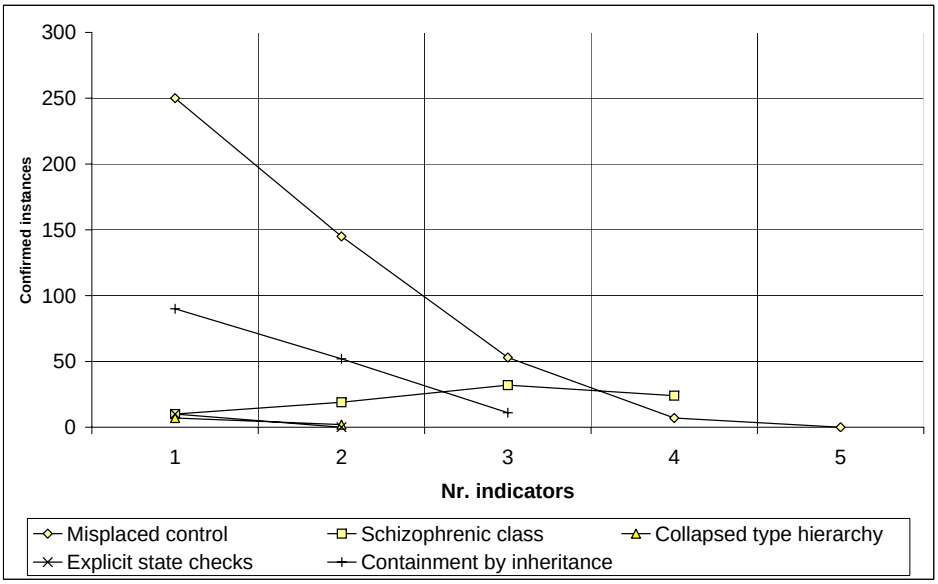


Figure 7.7.: Number of confirmed instances in relation to the number of indicators

the corresponding values of the standard error, as computed by the statistics package R[6].

From the results shown in the figure, we can conclude that "misplaced control" and "containment by inheritance" define the most relevant indicators, as indicated by the relatively low errors of the corresponding coefficients. For the reasons described above, the results for "schizophrenic class" are less conclusive and should therefore be taken with a grain of salt. Finally, in the case of both "collapsed type hierarchy" and "explicit state checks", the second indicator seems as expected, to be the more reliable one. This can be explained by the relatively rudimentary way in which the analysis of variable identifiers (which is at the heart of indicator 1 in both design flaws) was implemented in our tool. Thus, we could find only 3 confirmed instances of "collapsed type hierarchy" that presented indicator 1, and no instance of "explicit state checks" with that indicator. The latter situation prevented us even to compute a coefficient, as indicated by the empty cells in the table.

As discussed in sections 5.3.3 and 7.2.1, performing linear regression brings the added bonus that the obtained coefficients constitute the weights of the respective indicators. The resulting linear equation acts as a predictive model for the corresponding design flaw. Due to the observations made earlier, we can only regard the predictive models of "misplaced control" and "containment by inheritance" with a certain degree of confidence. This confidence is however limited by the relatively low size of the observation set used in the regression. To obtain more reliable models, further experiments need to be per-

[6]http://www.r-project.org/

Containment by inheritance			
	I1	*I2*	*I3*
Coeff.	0.34	0.42	0.26
Std. Err.	0.033	0.073	0.030

Explicit state checks		
	I1	*I2*
Coeff.	–	0.42
Std. Err.	–	0.103

Collapsed Type Hierarchy		
	I1	*I2*
Coeff.	0.38	0.25
Std. Err.	0.217	0.075

Schizophrenic Class				
	I1	*I2*	*I3*	*I4*
Coeff.	0.05	0.41	0.18	0.14
Std. Err.	0.049	0.061	0.062	0.052

Misplaced Control				
	I1	*I2*	*I3*	*I4*
Coeff.	0.23	0.15	0.21	0.18
Std. Err.	0.030	0.019	0.029	0.022

Figure 7.8.: Overview of results for experimental goal 2

formed.

For the two design flaws mentioned above, the corresponding predictive models can be expressed by the following equations:

$$MC = 0.22619\ I_1 + 0.15044\ I_2 + 0.207\ I_3 + 0.18421\ I_4 + 0.41154\ I_5 \tag{7.1}$$

$$CI = 0.33615\ I_1 + 0.42292\ I_2 + 0.26139\ I_3 \tag{7.2}$$

where MC and CI are approximations of the probability of presence for misplaced control and containment by inheritance respectively, and I_i are boolean variables that indicate the presence, or absence of the corresponding design flaw indicator.

Experimental Goal 3

Our third and final experimental goal was to compute the real recall for the two design flaws “collapsed type hierarchy” and “explicit state checks”. The results of our analysis are shown in figure 7.9. As indicated by the figure, we have identified only one false negative of type “collapsed type hierarchy”, in the case of class `Name` of the system JEdit. A relevant portion of the offending code is shown in listing 7.1.

Explicit State Checks		
System	*Potential false negatives*	*Confirmed false negatives*
XUI	0	0
JFreeChart	4	0
InjectJ	1	0
JEdit	6	0

Collapsed Type Hierarchy		
System	*Potential false negatives*	*Confirmed false negatives*
XUI	1	0
JFreeChart	1	0
InjectJ	3	0
JEdit	15	1

Figure 7.9.: Overview of results for experimental goal 3

```
Object resolveThisFieldReference( ... ) throws UtilEvalError {
    ...

    if ( isCompound( evalName ) )
       result = classNameSpace.getThis( interpreter );
    else
       result = classNameSpace.getClassInstance();

    ...
}
```

Listing 7.1: Collapsed type hierarchy in class `Name`

According to the comment associated to the class declaration, `Name` "implements a name resolver". An instance of `Name` stores an identifier, which may either be simple or compound (e.g. the simple name "`attribute`", as opposed to a fully qualified name such as "`obj.attribute`"). In effect, the class defines a hierarchy of two specialized versions of the common abstraction `Name`. Conditionals such as the one shown in listing 7.1 occur repeatedly in order to select the appropriate behavior, based on the actual type of name, which is stored in a private field called `evalName`.

The reason that prevented this valid instance of "collapsed type hierarchy" to be detected in previous phases of our experiment, is the fact that the field is not checked directly, but rather through the intermediary method `isCompound()`, which returns either true or false, based on the presence of the character "." in the contents of its parameter. In order to cover situations that are similar to this one, the indicator implementation in the tool can be extended to employ dataflow analysis techniques.

Examples

We conclude our discussion on the case studies, by presenting a few of the diagnosed design flaw instances in detail.

```
/**
  * This operation is not supported by this axis.
  *
  * @param g2  the graphics device.
  * @param dataArea  the area in which the plot and axes should be
      drawn.
  * @param edge  the edge along which the axis is drawn.
  */
protected void selectAutoTickUnit(Graphics2D g2,
                                  Rectangle2D dataArea,
                                  RectangleEdge edge) {
    throw new UnsupportedOperationException();
}
```

Listing 7.2: Refused bequest in `SymbolAxis`

Let us start with an example of the design flaw "containment by inheritance". In JFreeChart, the abstract class `ValueAxis` defines the generic concept of chart axes that display value data. The comment associated to the class declaration says: "The two key subclasses are `DateAxis` and `NumberAxis`". As expected, these concrete subclasses are valid specializations of the base abstraction, describing the behavior of time axes and numerical axes respectively. `NumberAxis` is further specialized by classes such as `CyclicNumberAxis`, `LogarithmicAxis`, and quite interestingly, by a class named `SymbolAxis`. With respect to the latter class, our tool found indicators 2 (refused bequest) and 3 (library superclass) of the design flaw "containment by inheritance". This raised the suspicion that the inheritance relation between `SymbolAxis` and `NumberAxis` is semantically not a valid specialization.

After looking at the code fragment, we could confirm that this suspicion was justified, as one could immediately guess from the names of the two classes. First of all, `SymbolAxis` does not support part of the interface of its superclass, as indicated by the presence of indicator 2 and the code fragment shown in listing 7.2.

In addition, the detector for indicator 3 found that `NumberAxis`, the superclass of `SymbolAxis` is a heavyweight class (WMC=125), defines no less than 38 methods, and has an extremely low cohesion at the same time (TCC=0.09). This made us suspect that the class is meant as a library of functionality. As it later turned out, we could also diagnose this class as being a "schizophrenic class".

In the course of investigating JFreeChart, we noticed that the misuse of the inheritance relation in this way is quite common throughout the system (33 similar instances were found).

The second example concerns a suspected instance of the design flaw "collapsed type hierarchy", in the class `BrowserIORequest` in JEdit. Our tool found both indicators

```
public void run() {
    switch(type) {
        case LIST_DIRECTORY:
            listDirectory();
            break;
        case DELETE:
            delete();
            break;
        case RENAME:
            rename();
            break;
        case MKDIR:
            mkdir();
            break;
    }
}
```

Listing 7.3: A method of the class `BrowserIORequest` affected by "collapsed type hierarchy"

of this design flaw, in connection with the private integer attribute called `type`. After inspecting the source code of this class, we confirmed that the repeated occurrence of switch statements in different methods, which is characteristic of the pathological structures for both instances of "explicit state checks" and "collapsed type hierarchy", were indeed referencing a variable that could be associated with the type of each class instance. The attribute is only written to once, at object creation, and thus has a constant value for the entire lifespan of the object. Several of the class mehods, such as the method `run()` shown in listing 7.3, check the value of this field in order to choose the appropriate specialized behavior.

Our third example is a suspected (based on the presence of indicator 2) instance of the design flaw "explicit state checks", detected in the class `ParserTokenManager` of JEdit. After a closer inspection of the code, we found that this generated class implements a parser, which in effect is a state machine with the current state represented by the current character in the input buffer. The explicit checks on the state are performed in no less than 38 different methods of the class. Therefore, as discussed in section 7.2.1, the design intent and thus the instance itself were confirmed as valid. However, if we were to take strategic closure into account, this flaw instance would probably be dismissed because the code of the class in question is generated, and therefore unlikely to ever be in need of human attention.

A good example instance of the design flaw "schizophrenic class", is the class `AbstractRenderer` in the system JFreeChart. This candidate manifests instances of all four indicators in the specification. Thus, the class is uncohesive, defines no less than

```
1  public void hidePage( XPage page ) {
2  ...
3
4      if ( ( lastPage != null ) && ( page != lastPage ) ) {
5          container.remove( lastPage );
6
7          page.setStatus( XPage.DEACTIVATED );
8          page.pageDeactivated();
9
10         container.doLayout();
11         validate();
12       }
13 }
```

Listing 7.4: Fragment of method `XApplet.hidePage()`

51 different attributes and over 150 methods, and has a weighted method count of 386, whereas the system wide average is around 30. In addition, the class is a bottleneck and defines several implicit interfaces that are used by disjoint groups of clients. According to the comment that is associated to the class declaration, the class is a "base class providing common services for renderers". Sure enough, `AstractRenderer` proves to be too abstract of a concept, and can be divided up into an entire rendering subsystem, defining many finer grained concepts, while the original class could play the role of a facade to the new subsystem.

Finally, we present example instances of the design flaw "misplaced control". Because this is the only method level design flaw in the set of flaws used in our experiments, it is also the most pervasive type of flaw, with 455 confirmed instances in the four analyzed systems. Furthermore, "misplaced control" has several possible manifestations in the structure, which also explains the large number of indicators defined for this type of design flaw. For these reasons, we will present three different instances, featuring different types of indicators that this flaw induces in the structure.

The first instance that we discuss here, was detected in method `hidePage()` of the class `XApplet`, in the system XUI. The offending method manipulates the internal state of the satellite class `XPage` (indicator 2). Furthermore, `XPage` favors external manipulations by exposing a significant number of its fields (approximately 90%), both directly and through accessor methods (indicator 4).

After manual inspection of the method body, we confirmed the presence of "misplaced control", in lines 7 and 8 of the source code fragment shown in listing 7.4. Concretely, lines 7 and 8 could be extracted to a new method called `deactivate()`, which should belong to class `XPage`. The problem with the existing design is that `XApplet` knows the two-step protocol for deactivating an instance of `XPage`: setting the internal state to

```
1  public void addObservation(double value, boolean notify) {
2      ...
3
4      SimpleHistogramBin bin = (SimpleHistogramBin) iterator.next();
5      if (bin.accepts(value)) {
6          bin.setItemCount(bin.getItemCount() + 1);
7          placed = true;
8      }
9
10     ...
11 }
```

Listing 7.5: Fragment of method `addObservation()`

```
1  public String getAssignmentOperator() {
2      String op = "";
3      if (recoderAssignment instanceof BinaryAndAssignment)
4          op = "&=";
5      else if (recoderAssignment instanceof BinaryOrAssignment)
6          op = "|=";
7      else if (recoderAssignment instanceof BinaryXOrAssignment)
8          op = "^=";
9      else if (recoderAssignment instanceof CopyAssignment)
10         op = "=";
11     else if (recoderAssignment instanceof DivideAssignment)
12         op = "/=";
13     ...
14
15     return op;
16 }
```

Listing 7.6: Fragment of method `getAssignmentOperator()`

`XPage.DEACTIVATED`, and firing an internal handler in `XPage`. Relocating this knowledge to `XPage` would shield `XApplet` and other potential clients of this class from possible changes to this protocol.

The second example of misplaced control affects the method `SimpleHistogramDataset.addObservation()`, and involves state manipulation of the class `SimpleHistogramBin` (indicator 2), combined with an information loop (indicator 3). Listing 7.5 shows an excerpt of method `addObservation()`. Line 6 of the listing contains the problematic information loop: in order to add a new observation to a `SimpleHistogramBin`, the method queries the current number of observations by accessing an internal field of the class, computes a new value by adding 1 to the queried value, and finally writes back the newly computed value into

the same field. The problem with this type of structure is similar to the previous case, namely that the offending method defines the protocol for adding a new observation to an instance of `SimpleHistogramBin`. This means that if this protocol would need to be changed (for example by adding some notifications to various listeners), the class `SimpleHistogramDataset` and maybe others would need to be updated as well. This kind of knowledge really belongs to the owner of the attribute that stores the item count, that is `SimpleHistogramBin`.

Finally, the last example that we are going to discuss shortly, is an instance of "misplaced control" involving type checks using reflection. The offending method is in this case `RecoderAssignment.getAssignmentOperator()` in system Inject/J. A portion of this method is shown in listing 7.6. The variable `recoderAssignment` that is checked in each `if` statement is a private field of type `Assignment`, declared in class `RecoderAssignment`. This type of explicit checking is wrong because it induces a dependency to every concrete type of assignment, and contains knowledge about the internal nature of other, completely unrelated classes. In addition, such constructs are also completely redundant. A much simpler and better solution would be to simply define an abstract method named `getAssignmentOperator()` in `RecoderAssigment`, and then appropriately override this method in the various specializations of this class. Thus, the previously explicit information about each assignment type would be encapsulated and distributed among the rightful owners of such information.

Conclusions

The results of the above experiments show that it is possible to implement a tool that automates most of the diagnosis method proposed in chapter 5, and that such a tool can be used effectively on intermediate to large-sized software systems. Working completely alone, the author was able to perform the four case studies in less than three weeks time. However, most of the performed activities can easily be parallelized and distributed among several people, thus reducing the absolute amount of time even further.

Chapter 8.

Conclusions

The purpose of this research was to explore the possibility of transforming object oriented restructuring, from an ad hoc process that is heavily dependent on intuition and personal know-how, into a systematic process that supports a high level of automation. To this end, we introduced a new definition for the notion of design flaw. The essential difference between our definition and previous ones is the encapsulation of semantics, in order to describe a context in which the structure of a given design fragment can *justifiably* be considered inadequate.

Apart from being able to make this type of justification, the knowledge of the context allows us to describe those characteristics of the structure that are missing, but desirable in that particular context. Thus, pinpointing an instance of a given design flaw amounts to establishing a new, from the standpoint of maintainability better structure, for the affected design fragment. This particular step (previously referred to as problem analysis) used to pose a major obstacle in the way of automation.

In light of the above, the problem of defining a systematic process for restructuring can be reduced to the problem of defining a systematic process for pinpointing design flaws. We successfully addressed this problem and thus achieved the goals set out in section 1.1.3, by developing a systematic process to diagnose design flaws. The process has certain similarities to medical diagnosis, and employs ordinary static analysis techniques in order to locate design flaw instances in the code. Furthermore, our diagnosis method is to a very large extent automatable, as demonstrated by a series of case studies performed on a representative collection of intermediate to large sized software systems.

We are now in a position to assess our method with respect to the set of criteria put forward in section 1.1.3 of the introductory chapter.

8.1. Assessment of Proposed Method

The proposed method complies with all criteria that had been put forward in section 1.1.3, as argued in the following:

Comprehensiveness: According to this criterion, any restructuring methodology should be able to address all types of problems that pertain to every major concern of component level object oriented design. In particular, the following three types of concerns were considered (see [LM06]): class definition as a means to express key abstractions of the problem and solution domains, inheritance relations as a means to express specialization hierarchies between families of abstractions, and patterns of cooperation between objects that represent instances of unrelated abstractions. As argued in chapter 4, the design flaw catalog of appendix A addresses a representative selection of problems that pertain to each of these types of concerns. Moreover, the feasibility and effectiveness of the proposed tool supported diagnosis method has been demonstrated in chapter 7, for design flaws that address each of the three concerns. Our method can not only address the same kinds of problems as traditional methods revolving around code smells and design patterns, but it does so in a more profound and accurate way. Specifically, our method is able to distinguish between subtle variants of problems that were previously treated as one, such as the code smell "switch statements" [Fow99], and the anti-pattern known as "the blob" [BMMM98].

Causality: This criterion requires the existence of a causal link between the design context of the affected fragment, and the recommended target structure that results from applying the restructuring method. In other words, the restructuring decisions that result from applying the method must match those that a human engineer would make while manually restructuring the system. We identified design intent and strategic closure as being the necessary ingredients in a fragment's context. Our restructuring method is causal by construction, because it fully takes the design context of the analyzed fragment into account. Furthermore, we showed how a tool that implements our proposed diagnosis process can recover design intent in an automated way.

Systematic process: The requirement imposed upon the method by this criterion is a systematic nature. Specifically, the human engineer has a predefined decision tree, and need only reason about a problem in terms of each decision, not about the structure of the decision tree itself. Our restructuring method fulfills this criterion by construction, because on one hand design flaws are defined in such a way that diagnosing a design flaw instance amounts to establishing the necessary target structure, and on the other hand the diagnosis process itself is systematic. Therefore, we have proposed a systematic process that replaces what was previously referred to as problem analysis.

Automation: The last criterion requires that most of the restructuring process support automation, especially the decision making process that was formerly part of problem analysis. Furthermore, the attained level of automation should allow the use of the method on medium to large sized systems, in a reasonably short interval of time.

Our prototype tool CodeClinic and the experiments described in chapter 7 demonstrate the viability of an implementation, as well as its practicability on a code base totalling around 300,000 lines of code, and with an associated effort on the order of magnitude of a few person weeks. Furthermore, this type of restructuring activity can easily be parallelized and distributed among several people, either by breaking up the subject system into its constituent subsystems, or by dividing up the set of design flaws that form the object of investigation.

8.2. Summary of Contributions

The research presented by this work brings a number of essential contributions to the field of object-oriented restructuring, as summarized in the following:

- **A new way of defining the notion of design flaw**, that is tailored to the specific needs of restructuring. Design flaws allow the specification of a causal treatment of commonly encountered deficiencies in object oriented designs.
- **A method for diagnosing design flaw instances**, which replaces the activities traditionally known as problem detection and analysis, with a systematic, largely automated process. The method has a major practical implication, namely that for the first time, a complete tool coverage of the restructuring process is made possible. A number of case studies demonstrate the practicability of the method for intermediate to large sized software systems.
- **A systematic, tool supported restructuring process**, that ensures a causal treatment of diagnosed design flaw instances.
- **Restructuring patterns**, as a means of recording and disseminating restructuring expertise. A restructuring pattern encapsulates a design flaw specification, along with all the information required for its diagnosis and correction, either manually or using specialized tools.
- **A catalog of ten representative design flaws and associated restructuring patterns**, ready to be used in day to day practice.
- **An extensible diagnosis tool**, integrated in the Eclipse development environment, which operates for systems written in the programming language Java.

8.3. Future Work

One possible direction to extend this research is into other programming paradigms and associated languages, most importantly procedural programming. As discussed in chapter 4, the concept of design flaw is very general, and therefore applicable in principle to

any design paradigm. In particular, the set of problems posed by procedural programs are essentially very similar, and the existing set of design rules and guidelines for these languages is ample.

The second worthwhile area of research would be a detailed study of design flaw interference. We see a significant potential for optimizing both the diagnosis and the reorganization processes by properly taking design flaw interference into account. In particular, as discussed in section 6.2.1, one could classify design flaws according to several criteria, such as the type of concern that they address, or their level of granularity. Based on these classifications, one could attempt to define precedence rules between various types of flaws. For example, such a rule could state that design flaws pertaining to inheritance relations should be resolved before design flaws such as "misplaced control", because it makes little sense to make decisions concerning the distribution of responsibilities between classes, unless the conceptual relations between these classes accurately reflects the conceptual domain of the application. During diagnosis, this would result in the elimination of a significant number of irrelevant candidates, and therefore to less effort on the part of the user. Performing costly and risky refactorings in vain could be avoided in this way.

Another possible area of investigation would be to see whether expanding the scope of the automated analyses to other than the static structure, would bring more accuracy to the diagnosis process. We think of two immediate possibilities. First, a combination of static and dynamic analyses could bring more accuracy in situations where the behavioral aspects of involved entities play an important role. For example, runtime information could help tip the balance in favor of design flaws such as "collapsed type hierarchy" A.1 and "explicit state checks" A.3. And second, historical information obtained from a version management system such as CVS, could help the user decide on the strategic closure for a suspected design fragment.

Finally, further improvements could be brought to the implementation, by integrating it with the refactoring engine provided by the development environment. Reorganization strategies could be implemented in the form of wizards, that could guide the user through every step of the specified reorganization strategy. In addition, both the methodology and the implementation could benefit from the integration of state of the art visualization techniques, such as those presented in [Lan03] and [LM06].

Appendix A.

Design Flaw Catalogue

A.1. Collapsed Type Hierarchy

A.1.1. Description

According to [Rie96], the most natural way of expressing the specialization relationship ("is kind of") between abstractions, is by implementing a type hierarchy using inheritance. In [Mey88], the author distinguishes between horizontal and vertical type generalization. Horizontal generalization is expressed through type parametrization, also known as generics. Specialization on the other hand, corresponds to vertical generalization, and is expressed through inheritance.

Thus, in an inheritance hierarchy, parent nodes represent vertically generalized abstractions of their children, which in turn are specializations of their parents. All members of the hierarchy support the interface of the root class, which can be used polymorphically.

A collapsed type hierarchy is the situation where an abstraction "absorbs" its own specializations, and emulates the specialization hierarchy, by explicitly checking the value assigned to a variable that represents the object's special type. Figure A.1 depicts an example instance of a collapsed type hierarchy in a text editing system that handles ASCII and rich text documents. The class `Document` in the figure, provides the generic interface which defines common operations on generic documents (e.g. the methods `open()`, `copy()` and `paste()`). The implementation of the class makes use of a variable (e.g. `docType`) to track the current type of the document being processed. Its methods employ switch conditional constructs to inquire the current value of this variable in order to provide the needed specialized behavior.

A collapsed hierarchy makes understanding and changing individual specializations harder because their internal data and code are entangled in a single, bulky class. Also, it is hard to clearly distinguish general code from specialized code, and extend the hierarchy with new specializations.

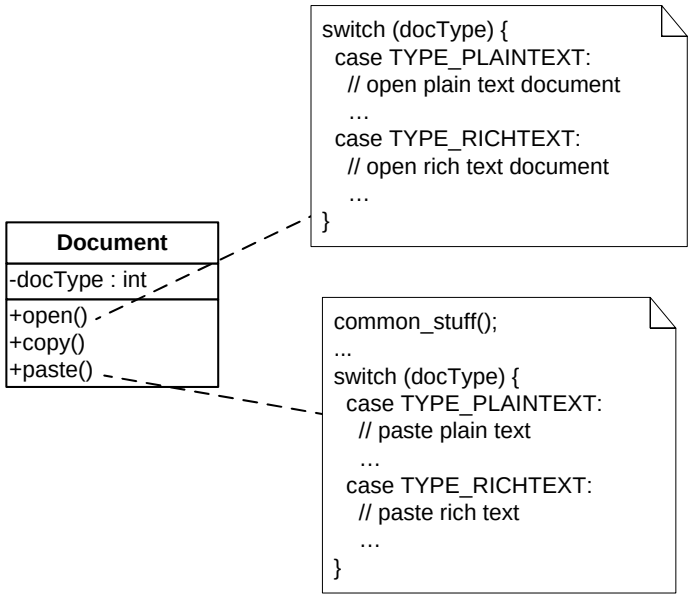


Figure A.1.: An example of *collapsed type hierarchy*

A.1.2. Context

Design intent

You need to express a specialization hierarchy of a class, that represents a valid abstraction in the design of application. Clients of the root class need to access specialized versions in a transparent way, using the interface defined by the generic abstraction.

Strategic closure

The number or implementation details of individual specializations is expected to change.

A.1.3. Imperatives

In order to maximize maintainability in the context described above, it is important to have a clean physical separation between the specializations themselves, as well as between what is common and what is characteristic for each specialization. This is most naturally achieved with the use of the inheritance relation. This will reduce the time needed to understand, add, change or remove individual specializations.

A.1.4. Pathological Structure

As exemplified in figure A.2, the entire hierarchy is collapsed into a single class which implements the root abstraction's interface. The implementations of the various operations

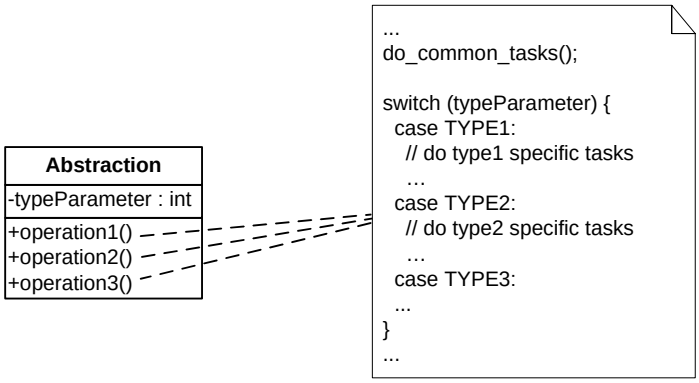


Figure A.2.: Pathological structure for *collapsed type hierarchy*

of this interface use a variable (either an attribute or a method parameter) for switching between the alternative behaviors. The attribute is usually initialized at the moment of instantiation and semantically embodies its concrete type. If method parameters are used, the clients pass arguments in order to request the expected behavior.

Although the concrete behavior of the abstraction instance is transparent to clients after its initialization, the design presented above has some serious drawbacks. The class that implements the abstraction is a monolith, in which specializations are tangled with one another in the implementation of every method defined by the generic interface. Because of this, the class increases in size and complexity. The entanglement on one hand and the increase in size and complexity on the other hand, make the design fragment hard to understand and change in the ways described in the context.

A.1.5. Reference Structure

The reference structure in the given context uses inheritance as the natural way to express a specialization hierarchy. As depicted in figure A.3, the type variable is no longer necessary, because the desired specialization is chosen through instantiation. Thus, the root abstraction received subclasses that correspond to each of the specializations. The large conditional constructs in the main abstraction have been dismantled branch by branch, and each branch has moved into the corresponding subclass. Each of the subclasses may implement any of the operations in the common interface in its own special way. Any

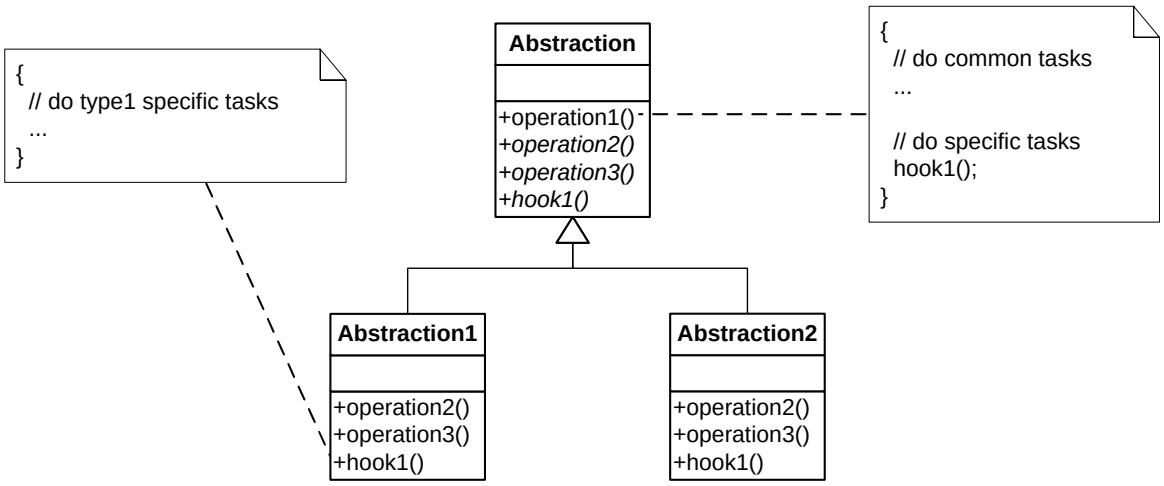


Figure A.3.: Reference structure for *collapsed type hierarchy*

behavior that was common to all specializations has migrated into special hook methods, in concordance with the template method design pattern.

A.1.6. Diagnosis Strategy

Search space

All classes of the system.

Initial filter

A non trivial class that contains at least two `switch` or equivalent `if-else` constructs, located in separate methods, not using runtime type identification, either on the same specific class attribute or using formal parameters that have the same name.

Indicators

Indicator 1: *The name of the parameter used in the conditional expressions contains the word "type", thus suggesting a type variable. Alternatively, the parameter is compared against a set of symbolic constants whose names contain such a word.*

Indicator 2: *Usage patterns of the parameter used in the conditionals suggest a type variable. In other words, there is no write access on the variable, anywhere in the class, with the exception of object constructors.*

Context matching

Question 1: *It must be confirmed that the class represents a valid abstraction in the design of the system.*

Question 2: *It must be confirmed that the parameter used in the conditional constructs represents a type code, used for implementing a specialization hierarchy of the abstraction represented by the class.*

Question 3: *It must be confirmed that the behavior being specialized semantically describes the abstraction in its entirety. In other words, the behavior that is being specialized corresponds to the abstraction modeled by the class in its entirety, not to a limited aspect of its implementation.*

Question 4: *It must be confirmed that the number or implementation details of individual specializations is expected to change.*

A.1.7. Reorganization Strategy

1: Based on the range of allowed values of the parameter, identify specializations of the class
2: **if** class has no subtypes **then**
3: Apply refactoring "replace type code with subclasses" [Fow99] for the identified specializations
4: Apply refactoring "replace conditional with polymorphism" [Fow99] for the newly created subclasses
5: **else**
6: Apply design pattern "bridge" [GHJV96] to extract the collapsed hierarchy into a parallel inheritance hierarchy
7: **end if**
8: Push up common behavior as high as possible in the newly created inheritance hierarchy, by creating template methods, in accordance with the design pattern "template method" [GHJV96]

A.2. Embedded Strategy

A.2.1. Description

In object oriented design, functionality is distributed among objects, based on the identity and perceived responsibilities of their corresponding classes. Thus, objects can be seen as actors that cooperate in order to realize the system's functions. Oftentimes, one class instance needs to be able to vary some detail of its behavior dynamically, based on the momentary needs of its clients. Normally, this can be achieved elegantly by employing the strategy design pattern. It allows defining a family of interchangeable algorithms in the form of a hierarchy. A client uses one member of this hierarchy in order to dynamically configure the class instance. Thus, the configurable instance is said to provide a context for the algorithm used as a parameter.

An embedded strategy is the situation in which the class providing the context, explicitly switches between alternative algorithms, whose implementations are all hard-coded into the class itself. Figure A.4 depicts an example instance of an embedded strategy, in a hypothetical text editing application.

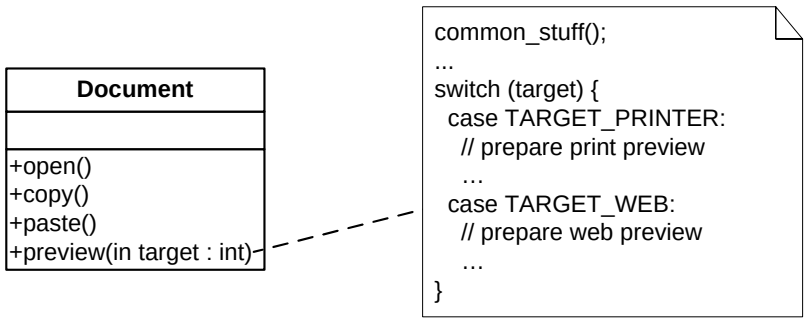


Figure A.4.: An example of *embedded strategy*

The class `Document` in the figure, provides the generic interface which defines common operations on generic documents, such as `open()`, `copy()`, `paste()` and `preview()`. Clients configure the preview operation by choosing between two alternative types of algorithms: one for print preview, the other for web preview. The `preview()` method uses a conditional construct in order to select the desired algorithm at runtime.

An embedded strategy makes understanding and changing both context class and individual algorithms harder, because their internal data and code are entangled in a single, bulky class. Also, it is hard to clearly distinguish general code from algorithm-specific code, as well as to implement additional algorithms.

A.2.2. Context

Design intent

Allow clients of a class that represents a valid abstraction in the application's design, to dynamically configure its instances with a family of interchangeable algorithms, that contribute to part of the services provided by the class.

Strategic closure

You expect the class providing the context, or the number and implementation of individual algorithms in the family to change independently.

A.2.3. Imperatives

From a maintainability standpoint, if the context and the algorithms are expected to change independently, it is important to have a clean separation between their implementations. Furthermore, in order to ease understanding and changing individual algorithms in isolation, it is important to cleanly separate their implementations from one another while simultaneously avoiding code duplication. Since the alternative algorithms semantically represent specialized versions of an abstract generic algorithm, we have a specialization hierarchy, that is best expressed using inheritance.

A.2.4. Pathological Structure

The pathological structure is depicted in figure A.5. All alternative algorithm implemen-

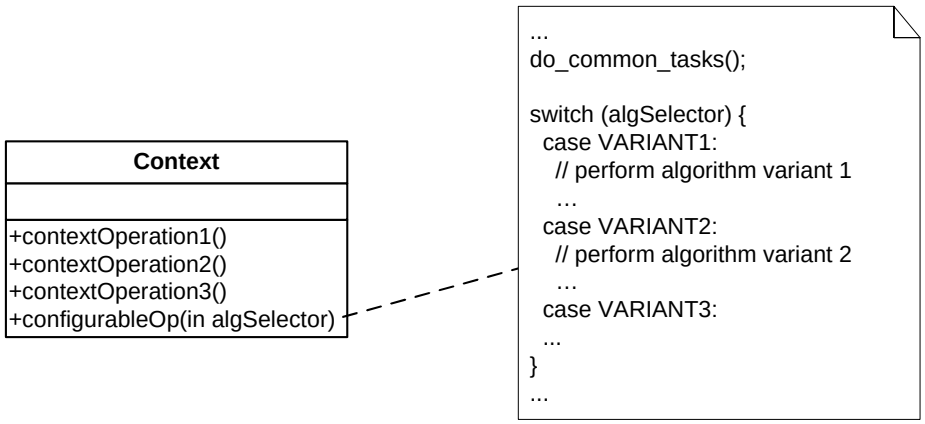


Figure A.5.: Pathological structure for *embedded strategy*

tations are contained within and mixed with the implementation of the context class. Clients choose between the various algorithms by means of selectors, defined either as class attributes or as parameters of those methods that support configuration. In order to select the desired behavior, the methods themselves use conditional constructs that explicitly check the value of the selector.

Although the class allows its clients to dynamically configure part of an object's behavior, the design presented in figure A.5 clearly disregards the imperatives described above. The implementation details of the algorithms are entangled with those of the context class, which impedes understanding and changing any of the two in isolation. In addition, the danger of having duplication between methods that rely on the same algorithms is very high.

A.2.5. Reference Structure

In the reference structure for the described context, the algorithm family is extracted out of the class providing the context, and modeled by a specialization hierarchy. This corresponds to the "strategy" design pattern. As depicted in figure A.6, the selector parameter

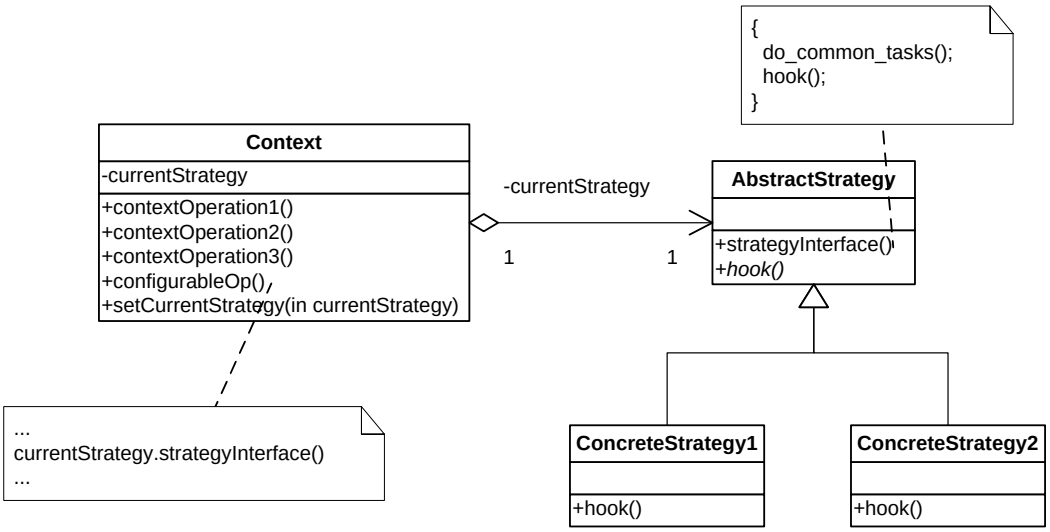


Figure A.6.: Reference structure for *embedded strategy*

is no longer needed, because explicit checks are now replaced by polymorphic calls to an abstract strategy interface. The individual branches of the former conditionals have migrated into the corresponding member of the newly defined specialization hierarchy. Clients can either use a dedicated method for configuring the context, as shown in the

figure, or can pass a reference to the desired strategy directly, upon each call. The use of inheritance allows extracting commonalities into the upper layers of the hierarchy, by employing the template method pattern.

A.2.6. Diagnosis Strategy

Search space

All classes in the system.

Initial filter

A class contains at one or more methods that employ `switch` or equivalent `if-else` constructs, not using runtime type identification, which check a particular class attribute, or formal parameters having the same name. The checked attribute or method parameters will be referred to as selectors.

Indicators

Indicator 1: *The name of the selector contains the word "strategy" or "algorithm", thus suggesting a strategy configuration parameter. Alternatively, the selector is compared against a set of symbolic constants whose names contain such a word.*

Indicator 2: *Usage patterns of the selector suggest a strategy configuration parameter. In other words, the value of the selector is dictated from outside of the class. If it is a class attribute, clients change the attribute themselves, either directly or through an accessor method.*

Indicator 3: *The concern that is being specialized represents a small fraction of the class. In other words, a very small number of the class' non-accessor methods contain the conditional constructs described in indicator 1.*

Context matching

Question 1: *It must be confirmed that the classes represents a valid abstraction in the design of the system.*

Question 2: *It must be confirmed that the parameter used in the conditional constructs represents a type code, used for implementing a specialization hierarchy.*

Question 3: *It must be confirmed that the abstraction being specialized does not correspond to the concept modeled by the class as a whole, but to some limited aspect of its implementation. The rest of the class can be regarded as the context in which this family of related algorithms perform some limited task.*

Question 4: *It must be confirmed that changes to class providing the context as well as the number and implementation of individual algorithms in the embedded hierarchy are likely to happen in isolation.*

A.2.7. Reorganization Strategy

1: Identify all abstract strategy types based on the logic of the conditional structures
2: **for all** strategy interfaces IS_i **do**
3: Based on the range of values taken by the type parameter, identify all concrete strategies that correspond to IS_i
4: Apply refactoring "replace conditional logic with strategy" [Ker05] to implement the "strategy" design pattern [GHJV96] corresponding to IS_i
5: Push up common behavior as high as possible in the newly created strategy hierarchy, by creating template methods, in accordance with the design pattern "template method" [GHJV96]
6: **end for**

A.3. Explicit State Checks

A.3.1. Description

In object oriented programming, polymorphism is the universal mechanism that allows an abstraction, defined by its interface, to vary behavior transparently with respect to its clients. In particular, polymorphism is the most natural way of altering an object's behavior, as observed by its clients, based on its current "state". Normally, under the notion of "state", we understand a snapshot of the current values of all attributes defined in the class. In this case however, "state" must be understood in an abstract sense, as the state of a domain abstraction that is modeled by the class.

The design flaw "explicit state checks" refers to the situation in which an object uses explicit checks on some internal piece of data, in order to execute state specific behavior or manage its "state" transitions. The data that is checked represents the current "state" at any given time.

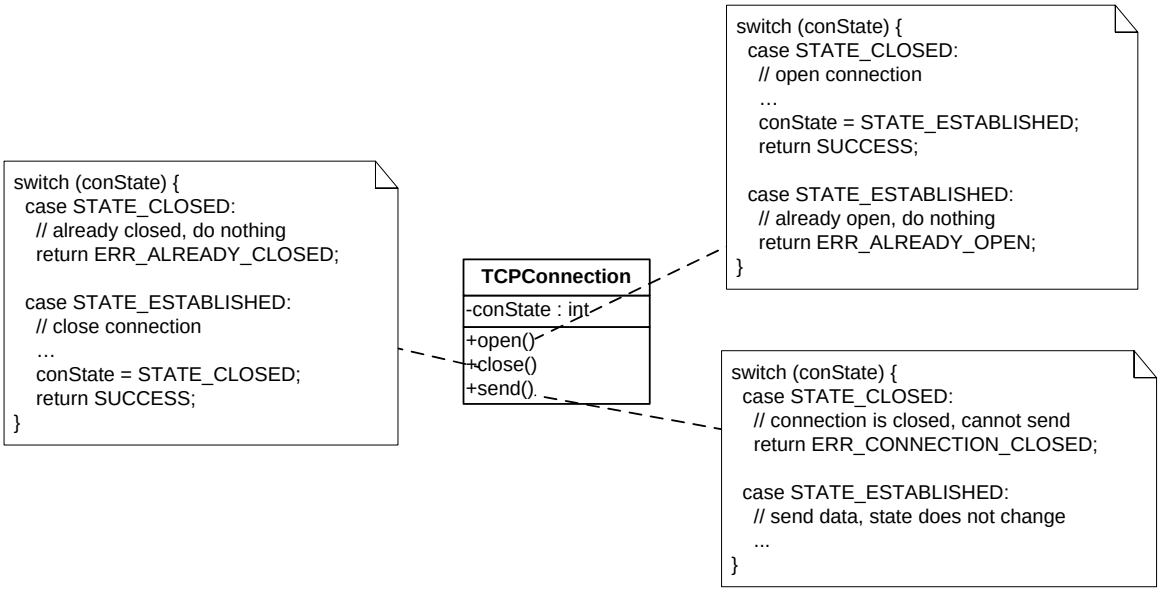


Figure A.7.: An example of *explicit state checks*

In the example depicted in figure A.7, the class `TCPConnection` uses the attribute `conState` to store the current connection state. The class implements a number of operations (e.g. `open()`, `close()`, `send()`) whose behavior varies according to this state. The selection occurs explicitly, by checking the attributes current value upon each

call. Some operations change the current state of the object by assigning a new value to the attribute.

Explicit state checks make it harder for the maintainer to add new states, as well as identify and change behavior that is specific to a given state. Furthermore, it is hard to distinguish state dependent from state independent code in the bloated classes that suffer from this design flaw.

A.3.2. Context

Design intent

Objects of a class that represents a valid abstraction in the application's design, need to vary their behavior dynamically, based on an abstract state, that can be managed either internally or externally.

Strategic closure

You expect the number of states to change, changes to the code that corresponds to individual states, or changes that would require the maintainer to distinguish state dependent from state independent behavior.

A.3.3. Imperatives

In order to maximize maintainability in the context described above, we must isolate on one hand, state dependent from state independent code from one another, and on the other hand, code that is specific for each individual state from one another. In addition, we can minimize the risk of code duplication by extracting commonalities in behavior among various states, in a specialization hierarchy.

A.3.4. Pathological Structure

Figure A.8 illustrates the most important characteristics of the pathological structure in the case of "explicit state checks". As shown in the figure, both state dependent and state independent code are contained inside a single monolithic class. The current state of an instance is held by an attribute that usually has some enumerated type. Throughout the implementation of the class, the value of this attribute is repeatedly checked inside typically large conditional constructs in order to select the desired behavior. State changes are carried out by assigning a new value to the attribute.

The pathological design clearly contradicts the imperatives described above. The entire functionality is entangled inside a single class, with several bloated methods. This leads to increased effort and error proneness in understanding and changing both state dependent and state independent code.

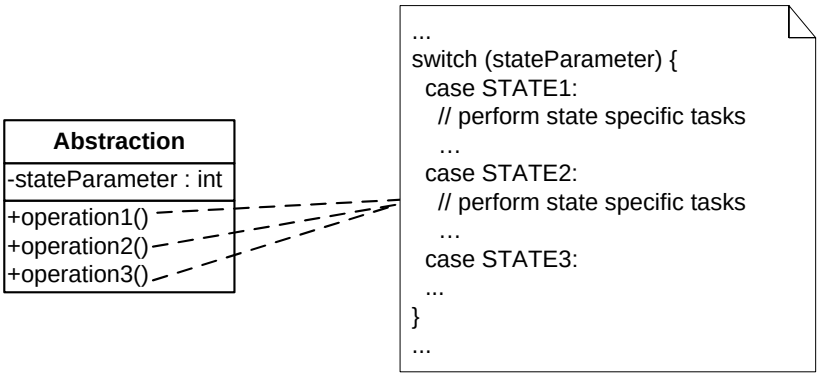


Figure A.8.: Pathological structure for *explicit state checks*

A.3.5. Reference Structure

In the given context, the reference structure corresponds to the design pattern “state”, as illustrated in figure A.9. In the structure presented in the figure, state specific behavior

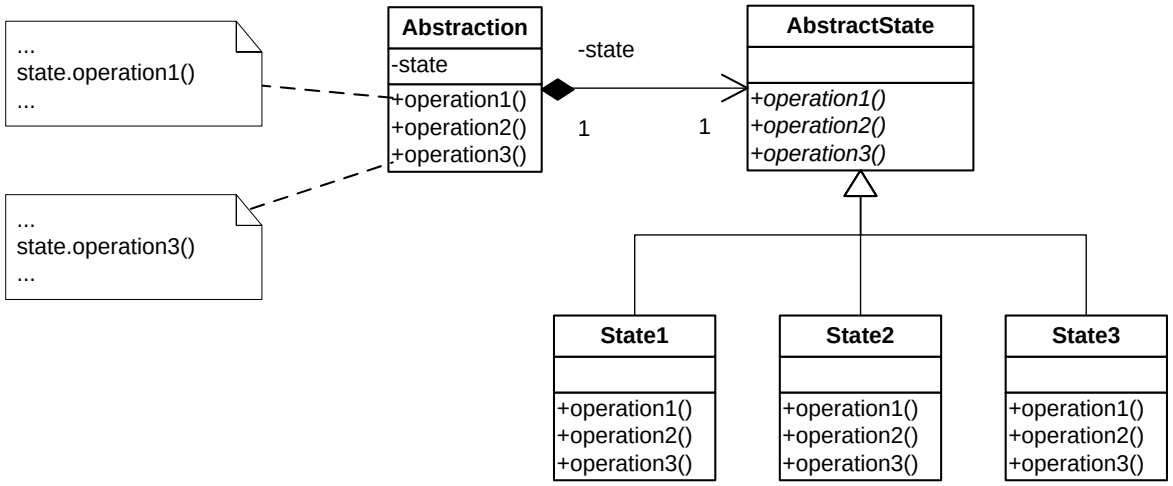


Figure A.9.: Reference structure for *explicit state checks*

has been isolated into separate classes, that form a specialization hierarchy. As a result, the large conditional constructs in the main class had been dismantled branch by branch, and each branch has moved into one of the concrete state classes. The main class aggregates the root of the specialization hierarchy, defining the common state interface. State management including instantiation of the state objects is generally best performed internally by the main class.

A.3.6. Diagnosis Strategy

Search space

All classes of the system.

Initial filter

A non trivial class that contains at least two `switch` or equivalent `if-else` constructs, located in separate methods, not using runtime type identification, on the same specific class attribute, which is not declared as final.

Indicators

Indicator 1: *The name of the checked attribute contains the word "state", thus suggesting a state variable. Alternatively, the switch parameter is compared against a set of symbolic constants whose names contain such a word.*

Indicator 2: *Usage patterns of the switch parameter suggest a state variable, in the sense that the value of the checked parameter is changed, either within branches of the conditional constructs, or from the clients that called the respective operation.*

Context matching

Question 1: *It must be confirmed that the classes represents a valid abstraction in the design of the system.*

Question 2: *It must be confirmed that the parameter used in the conditional constructs semantically denotes the state of the domain abstraction.*

Question 3: *It must be confirmed that the number or implementations of individual state specific behaviors are expected to change, or changes are expected that would require the maintainer to distinguish state dependent from state independent code.*

A.3.7. Reorganization Strategy

1: **if** state management occurs from within the conditional constructs described in indicator 1 **then**
2: Apply refactoring “replace state-altering conditionals with state” [Ker05] on the affected class
3: **else**
4: Based on the range of allowed values of the state parameter, identify the range of possible states
5: Apply refactoring "replace type code with state" [Fow99]
6: Implement a state management interface in the context class and adapt clients to use the context’s state management interface

7: If desired, optimize state management performance, by replacing on demand state object instantiation with pre-instantiated state objects, according to the “singleton” design pattern [GHJV96]
8: **end if**
9: Push up common behavior as high as possible in the newly created state hierarchy, by creating template methods, in accordance with the design pattern “template method” [GHJV96]

A.4. Dispersed Control

A.4.1. Description

One of the fundamental differences between the procedural and the object oriented paradigms, is the way in which complex operations are broken up into pieces that are allocated to various program functions. In the procedural world, we have a workflow, or activity based view, according to which a complex operation is broken up based on the logical steps in the workflow. In the object oriented world, a decomposition based on object identity and responsibilities is dominant while the activity based decomposition still plays an important role within classes.

There are however situations which justify concentrating bits of functionality from heterogeneous classes, giving priority to an activity rather than identity based decomposition. The design pattern "visitor" corresponds to such a situation, where the realization of functionalities on top of complex structures of related objects presumes a certain degree of orchestration between type specific behaviors. In order to make the implementation of complex operations that semantically pertain to an entire structure of objects maintainable, the individual bits of type specific behavior are encapsulated into a so called visitor class, that represents the high-level operation.

We have a case of "dispersed control", when in a situation such as the one described above, the implementations of structure related functionalities are broken up and dispersed between the individual types that belong to the structure. Figure A.10 illustrates an example of dispersed control in a hypothetical compiler. The heterogeneous structure is represented in this case by the abstract syntax tree, which is modeled as a collection of specialized `ASTNode` objects. Code generation represents the high level operation that is defined for the structure. Its implementation is dispersed throughout the `generateCode()` method in the entire `ASTNode` hierarchy. An external client to the hierarchy, called `CodeGenerator`, orchestrates the code generation functionality by iterating through the objects in the structure in a given way, and calling each object's individual version of `generateCode()`.

Since the types of individual nodes in the abstract syntax tree depend on the programming language that is being compiled, it is reasonable to expect that once implemented, the hierarchy will not change significantly. On the other hand, it is very probable that further global operations on the abstract syntax tree, such as type checking for example, will have to be added or changed frequently in the future. But maintaining such operations as well as adding new ones in this structure is difficult, because all descendants of `ASTNode` need to be adapted every time. In addition, individual types in the `ASTNode` hierarchy are harder to understand and changed, because their implementation is cluttered with the various bits of functionality that realize each global operation.

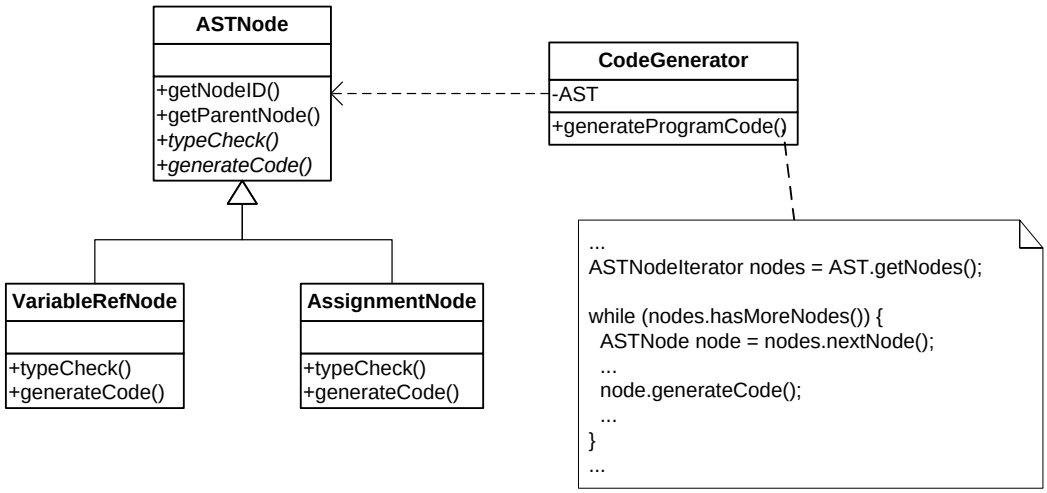


Figure A.10.: An example of *dispersed control*

A.4.2. Context

Design intent

You need to implement a number of operations that semantically pertain to a complex, heterogeneous structure, which consists of objects of classes, that represent a valid specialization hierarchy. The operations accumulate data from the structure elements, or perform some global function, by orchestrating between element specific bits of functionality.

Strategic closure

The number and implementation of the global operations is expected to change more frequently than the number and implementation of the individual element types forming the structure.

A.4.3. Imperatives

In order to maximize maintainability in the context described above, the implementation parts that are expected to change often (i.e. the global operations on the structure) must be decoupled from one another, and from those that are expected to change more rarely (i.e. the elements of the structure). In other words, the activity based view of complex operations should be given priority over the identity based view.

A.4.4. Pathological Structure

As shown in figure A.11, the inheritance hierarchy that describes the elements of the object structure, is also used for expressing the differences that exist between type specific bits of the global operation. In other words, every element's type interface is a mix of

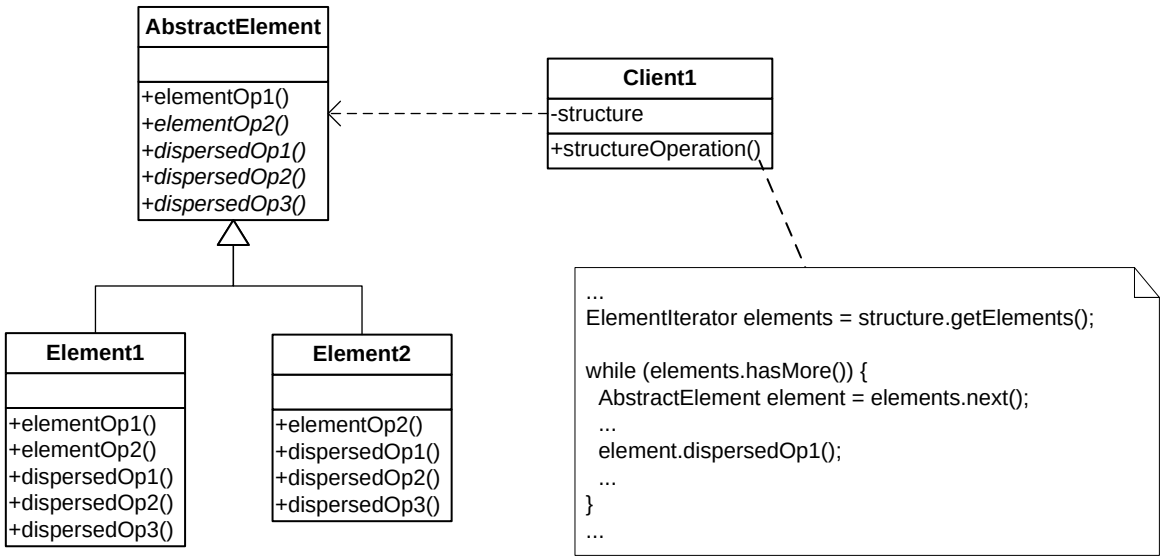


Figure A.11.: Pathological structure for *dispersed control*

both operations that are "local" to each element type, and operations that perform element specific bits of a greater, global operation that relates to the entire structure. The orchestration that is necessary in order to obtain the end result from the combination of individual element specific bits, is realized in an external client, which iterates over the structure in an appropriate way. Oftentimes, the purpose of the global operation is to derive some information that results from accumulating information from each structure element.

The pathological structure described above disregards the imperatives of this design flaw, because it mixes global, structure specific and element specific functionality, cluttering the implementation of all element types. Furthermore, individual structure related operations are hard to understand and change, because their implementation is dispersed throughout the entire element hierarchy. Adding a new global operation is hard, because it requires adding new methods to all element types.

A.4.5. Reference Structure

The reference structure is represented by the design pattern “visitor” and is illustrated in figure A.12. All individual bits defining a complex operation that pertains to the whole

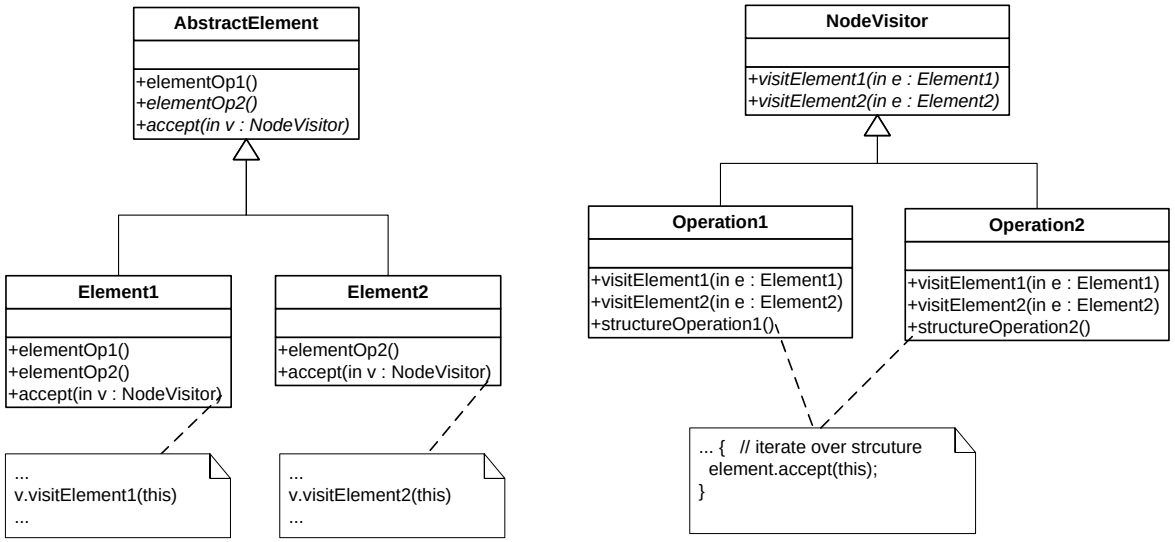


Figure A.12.: Reference structure for *dispersed control*

structure, have been encapsulated in separate visitor classes, which as special cases of visitors, are all part of a common specialization hierarchy. Each visitor class corresponds to exactly one global operation. Furthermore, a double dispatch mechanism has been implemented between the two hierarchies, which gives the maintainer complete flexibility in changing the number and implementations of individual visitors, without touching the element hierarchy. The element hierarchy defines a generic interface that allows a client to use an arbitrary visitor object, and each visitor object defines element specific methods, that are called during visitation. In addition, all element types need to provide an interface that allows a visitor to access element internal data. This break of element encapsulation is the price paid for the increased maintainability of the global operations.

A.4.6. Diagnosis Strategy

Search space

All type hierarchies in the system.

Initial filter

The root of the hierarchy defines a number of methods, either concrete or abstract,

that are either overridden or implemented in almost all terminal nodes of the hierarchy.

Indicators

Indicator 1: *The overridden methods represent bits of semantically unrelated global operations. Thus, there are no calls between any pair of such methods.*

Indicator 2: *The overridden methods represent bits of semantically unrelated global operations. Thus, clients do not call more than exactly one such method, from within any of their methods.*

Indicator 3: *The overridden methods are called from within contexts that suggest an orchestration, or an accumulation of information. An orchestration is probable if calls to many such methods occur from within a cycle, in which the call's target object is continually changed. Accumulation is probable if many such methods return information by means of a return value or output parameter.*

Context matching

Question 1: *It must be confirmed that the hierarchy represents a valid specialization hierarchy in the application's design.*

Question 2: *It must be confirmed that the identified group of methods represent bits of global operations that are semantically associated to a complex structure, formed with instances of the hierarchy. The clients that call methods in the identified group, orchestrate the calls to individual objects in the structure in order to accumulate information or otherwise perform a structure specific service.*

Question 3: *It must be confirmed that the number and implementation of the global operations is expected to change more frequently than the number and implementation of the individual element types forming the structure.*

A.4.7. Reorganization Strategy

1: **if** there is no double dispatch infrastructure in place **then**
2: Create abstract visitor class *AV*
3: **for all** types T_i that form the heterogeneous structure **do**
4: Create a corresponding `visit<...>` method in *AV*
5: Create a method named `accept(...)` that receives a reference of *AV*'s type as a parameter, and calls the proper `visit<...>` method on the received reference
6: **end for**
7: **else**
8: Let *AV* be the root of the existing visitor hierarchy

9: **end if**
10: **for all** dispersed operations O_i in the heterogeneous hierarchy **do**
11: Create a concrete visitor class V_i, as a subtype of AV
12: Apply refactoring "extract method" [Fow99] on the orchestration code within the client class, containing the call to O_i
13: Move previously extracted method to V_i
14: Replace the call to O_i with a call to the corresponding `accept(...)` method, passing the reference `this` as argument
15: Move all element specific method implementations that override or implement O_i, to the various `visit<...>` methods in V_i. If needed provide accessor methods to internal attributes of the heterogeneous element types
16: Remove all empty methods corresponding to O_i from the element hierarchy
17: **end for**

A.5. Misplaced Control

A.5.1. Description

The way in which functionality is broken up into subroutines or methods is one of the fundamental differences between the procedural and the object oriented paradigms. In the procedural world, we have a process, or workflow based view of the application's functionality, which is broken up into methods, based on the logical steps in the workflow. Thus, data is decoupled from the methods that use it. In the object oriented world, the predominant decomposition of functionality is based on object identities, and their associated responsibilities, while activity based decomposition should only play a role within classes. In other words, functionality and associated data are each divided and paired up, based on the responsibilities of each class. There are also notable exceptions to this principle, namely in those situations where the designer deliberately favors a workflow based decomposition by containing a complex process into a method, in order to make it easily maintainable. Such situations are present in the mediator, visitor and strategy design patterns.

Assuming a proper distribution of data among the classes of a system, we have a case of "misplaced control" in the situation where a piece of functionality is unjustifiably separated from the data its operates on. In other words, if a code fragment implements a service, based on data that is defined in a foreign class, that fragment is said to be misplaced, unless the placement is justified by the need to improve the maintainability of a complex operation that the fragment is part of. Figure A.13 illustrates an example instance of "misplaced control" in a hypothetical graphical editor.

The figure shows an excerpt of the method `weightAndPlaceNodes()`, of the strategy class `DiagramLayouter`. The method uses data that is strictly internal to the class `DiagramNode` in order to compute the weight for each node that needs to be placed in the current diagram. Assuming that there aren't alternative ways of computing a node's weight, and since the details of this process are not crucial from the standpoint of the layouting algorithm, the code block that computes node weights should naturally belong to the `DiagramNode` class. As can be seen from this example, the data encapsulation of the node class is broken in a brutal fashion, by having a data flow loop, in which data is first read out, processed and then written back to the foreign object.

Misplacing functionality in the way described above has many negative consequences for the understandability and flexibility of an object oriented design, because it favors the concentration of functionality in a few heavyweight classes (often referred to as behavioral god classes, or blobs). As a result, most of the other classes degenerate in "dumb" data holders, whose data encapsulation and hiding is abused systematically. Furthermore, this tends to be a self sustaining process, in the sense that adding new functionality to the lightweight classes becomes more and more difficult without major restructuring.

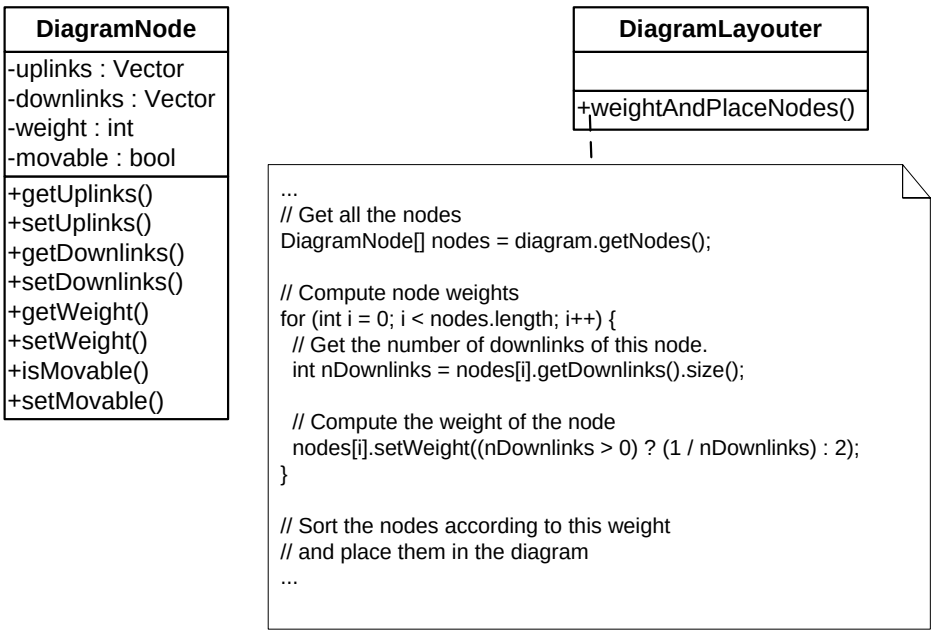


Figure A.13.: An example of *misplaced control*

The end result is a procedural style program with a lot of global data, and inherent maintenance problems.

A.5.2. Context

Design intent

Two or more classes, representing valid abstractions, that are not related through inheritance, hold data and/or responsibilities that are relevant in realizing a coherent unit of the application's functionality. Assuming a proper distribution of data among the classes, you want to distribute the implementation of the unit of functionality between them.

Strategic closure

You expect that maintenance activities are more probable to appertain to individual abstractions and their responsibilities, rather than the unit of functionality as a whole. Alternatively, you expect the need to specialize some of the abstractions involved in realizing the unit of functionality.

A.5.3. Imperatives

In order to maximize maintainability in the context described above, the responsibility based decomposition of behavior should takes priority over the workflow based decomposition. Furthermore, it is essential to maximize data hiding, thus eliminating unnecessary coupling between the classes that represent the cooperating abstractions. The bits of functionality that are assigned to each class have to naturally fit the class' identity and responsibilities. Assuming a proper distribution of data between the classes, the bits of functionality should be placed in the same class that holds the data that is relevant for the realization of the corresponding behavior.

A.5.4. Pathological Structure

The pathological structure for "misplaced control" is characterized by figure A.14, which shows the class
textttCentralClass, holding misplaced behavior, and its so called satellite class, holding data that is required by the misplaced behavior. This role assignment to the classes is

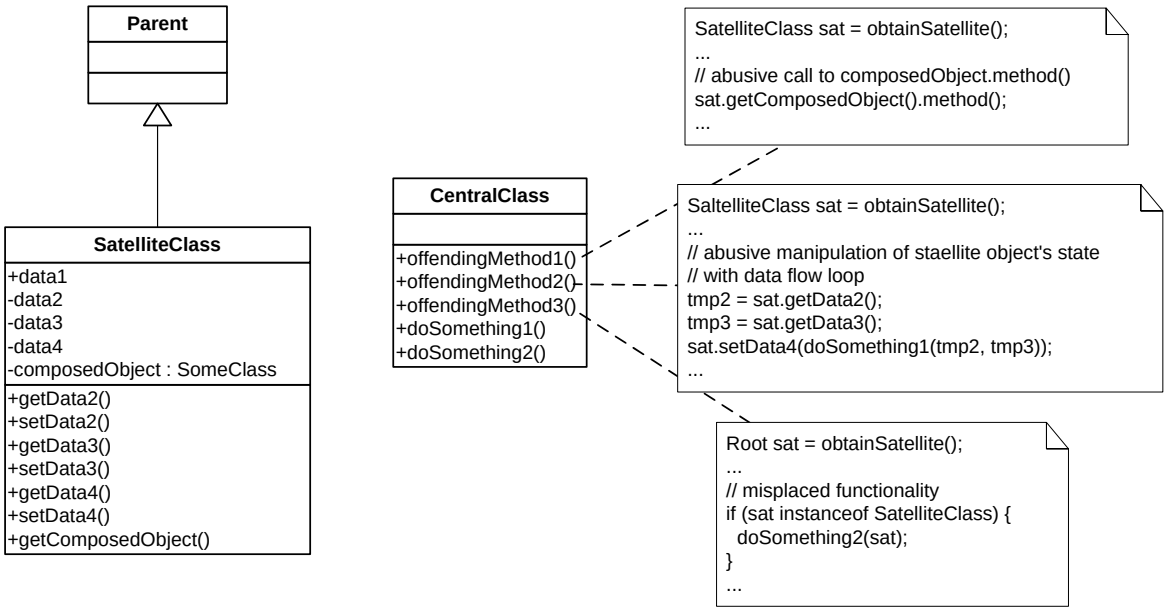


Figure A.14.: Pathological structure for *misplaced control*

only significant for the three concrete instances of the design flaw that are exemplified in `offendingMethod1()`, `offendingMethod2` and `offendingMethod3`. Any

class, including `SatelliteClass` may contain misplaced behavior, in which case the roles would be reversed.

In general, satellite classes are characterized by broken encapsulation, either by having public attributes or a lot of accessors. On the other hand, we have three examples of misplaced behavior in the figure. In the first example, the encapsulation of the satellite is broken by violating the "law of Demeter for methods"[1], by calling methods on an object reference, obtained from the satellite. In the second example, we have an information loop in the sense that the offending method pulls information from the satellite, transforms it in one way or the other, and then pushes the result right back into the satellite. The third and last example of misplaced behavior is when the offending method performs some processing on an instance of the satellite class, based on its concrete identity. In all three cases, the offending method contains code that can be regarded as providing some value added service on foreign data, and that would naturally belong in the satellite.

From the standpoint of the imperatives described above, data and associated behavior are not kept in the same class, which leads to broken encapsulation and unnecessarily high coupling between the central class and its satellite(s). This has a negative effect on the ease of understanding and ease of change for the classes involved.

A.5.5. Reference Structure

As depicted in figure A.15, the reference structure is characterized by the fact that the former satellite encapsulates and effectively hides its internal data from the former central class. It does this by providing a number of value added services on its data (the methods `doSomething1()`, `doSomething2()` and `performService()`), which contain code that was formerly placed in the central class. If we look at the implementations of the former offending methods, `Method1()` through `Method3()`, we see that all three forms of "misplaced control" have been defused by using delegation. In the case of `Method3`, the explicit type check has been replaced wit a polymorphic call to the method `doSomething2()`, defined in the satellite's parent class.

A.5.6. Diagnosis Strategy

Search space

All methods defined in all classes of the system.

Initial filter

Any non-trivial method, that is not a constructor or an accessor, and references a different type. Referenced types will be referred to as "satellites".

[1] see http://www.ccs.neu.edu/research/demeter/

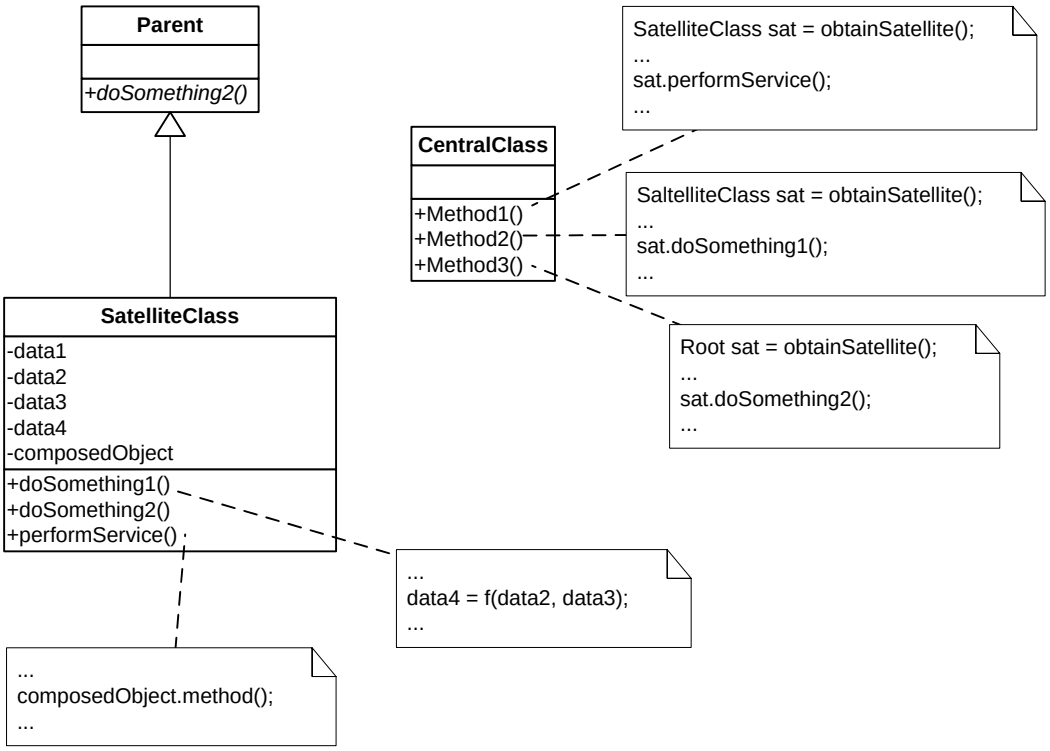


Figure A.15.: Reference structure for *misplaced control*

Indicators

Indicator 1: *The method accesses at least two foreign fields, either directly or through accessors. The accessed fields are declared in classes that are not superclasses of the one declaring the method.*

Indicator 2: *The method manipulates the internal state of another, unrelated class, by writing to one or more of its attributes, either directly or through a setter method.*

Indicator 3: *The method contains an information loop in the sense that information is pulled from instances of an unrelated class using accessors, transformed it in one way or the other, and the result is pushed back into the class either by writing to an attribute or as a parameter in a method call.*

Indicator 4: *Unrelated classes whose data are read or written to by the method, act as "dumb" data holders, by exposing a significant proportion of their data, either directly (i.e. public attributes) or through accessors.*

Indicator 5: *The method varies its behavior based on the identity of an object, using runtime type identification. In other words, it compares the type of a reference against*

an unrelated (i.e. through inheritance) type, adapting its behavior accordingly.

Context matching

Question 1: *The maintainer must confirm that the classes involved represent valid abstractions in the design of the system and the distribution of data among them is semantically sound.*

Question 2: *The maintainer must confirm that the code fragment that accesses data in the satellite class, can be seen as a service that is useful on its own, on data that belongs to the satellite class.*

Question 3: *The maintainer must confirm that future maintenance activities are more probable to appertain to individual abstractions and their responsibilities, rather than the unit of functionality as a whole. If the class in question implements a mediator, concrete visitor, or a concrete strategy, the user must ensure that extracting the misplaced unit of behavior is meaningful and desirable.*

A.5.7. Reorganization Strategy

1: Let m be the offending method
2: **for all** satellite objects Sat_i **do**
3: Based on the places in m where attributes of Sat_i are accessed, determine abstract services that can be logically seen as falling within the responsibility of Sat_i
4: **for all** Identified services Srv_j **do**
5: **if** the implementation of Srv_j does not correspond to the entire method body of m **then**
6: Apply refactoring "extract method" [Fow99] on code that corresponds to Srv_j
7: **end if**
8: Apply refactoring "move method" [Fow99] on the method that contains the implementation of Srv_j, to move it into Sat_i, or one of its superclasses, if appropriate
// This can for example be the case when satellites are siblings in an inheritance hierarchy and the method m uses reflection to check the exact type of the satellite instance in order to perform some type-specific action. The recommended solution is to move the code fragments involved to their corresponding satellite classes and consequently extract the commonality out of those operations into a common ancestor.
9: **end for**
10: Revise and if appropriate reduce the visibility of data members in Sat_i
11: **end for**

A.6. Schizophrenic Class

A.6.1. Description

In object oriented design, a class should not capture more than one key abstraction. Key abstractions are defined as the main entities within a domain model, and often show up as nouns within requirements specifications ([Rie96]). A key entity is an abstraction that stands on its own in the abstract model that results from modeling activities.

A "schizophrenic class" is a class that captures two or more key abstractions. Class schizophrenia is common in situations where a system is developed incrementally, without restructuring in-between increments. Thus, a class that is defined very early in the process, may prove to be too abstract later on, as it receives more and more responsibilities. Consequently, the class needs to be broken into fragments that capture more fine-grained abstractions.

Another possible scenario for the formation of schizophrenic classes is in the process of migrating a procedural system to an object oriented language. Large chunks of formerly global data are grouped together with functions that use this data into a single, large and noncohesive class.

In both scenarios mentioned above, the class encapsulates the data and behavior of two or more design abstractions. In addition, the encapsulated abstractions are described based on their identity. In other words, we have an object oriented topology of the underlying abstract model that results from modeling activities (i.e. the model describes the encapsulated abstractions as individual, cooperating actors).

Classes whose names contain words such as "system", "subsystem" or "manager" are likely candidates for class schizophrenia. However, there are also exceptional situations, in which a class is intended to provide a unified, simpler interface, to a complex set of interfaces that form a subsystem. This is the case of the "facade" design pattern. Nevertheless, a facade is primarily delegating to the responsible classes and does not aggregate all the data that define the abstractions in the subsystem.

Figure A.16 shows such an example instance, where the class `SmartHomeManager` implements three interfaces that define clearly separated responsibilities: the air conditioning system, the alarm system and the lighting system. The implementation of these three abstractions relies on partially overlapping data, but is mostly not related to one another. For example, the air conditioning shares the attribute `windowStates` with the alarm system, and the attribute `windowShadeStates` with the lighting system. The schizophrenic class doesn't always implement explicit interfaces, but we decided to include them in the example, because it is suggestive of the impersonation of various roles that the instance of a schizophrenic class does for various clients in the system.

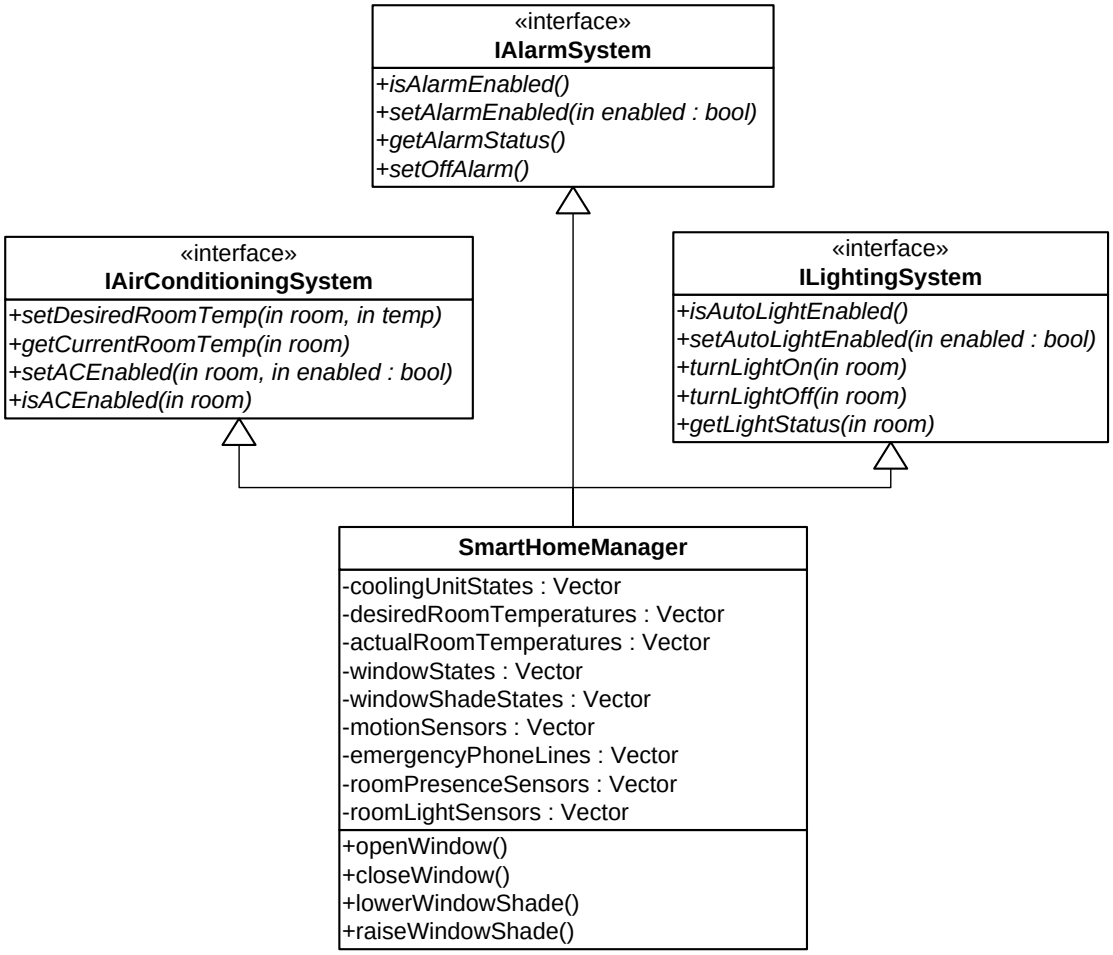


Figure A.16.: An example of *schizophrenic class*

A schizophrenic class negatively affects the ability to understand and change the individual abstractions that it captures, in isolation.

A.6.2. Context

Design intent

You want to express a set of individual design abstractions in a model having an object oriented topology, as classes in the system. Alternatively, you are migrating a procedural program to an object oriented language, and you need to define cooperating classes, from chunks of global data and functions.

Strategic closure

The design abstractions under consideration are expected to change independently from one another.

A.6.3. Imperatives

In general, in order to maximize maintainability, each class should capture no more than one key abstraction. Key abstractions are abstractions that stand on their own in the abstract model which determines design intent. In addition, all data that is related to one of the class' responsibilities, and all behavior that is related to the class' data should be kept together, in the same class [Rie96].

In order to maximize maintainability within the described context, we need to isolate those design abstractions from one another, that are expected to change independently. This implies that the functional decomposition of the original class' behavior needs to be replaced with an identity based decomposition that reflects the decomposition of data among the involved abstractions, so that data and associated behavior are kept together.

A.6.4. Pathological Structure

As illustrated in figure A.17, the pathological structure is characterized by the fact that the offending class encapsulates more than one key abstraction. As a consequence, we expect to find relatively isolated clusters of data and associated behavior, that represent the implementations of the corresponding logical interfaces. The interfaces can be either implicit, or declared explicitly, and they correspond to the various abstractions encapsulated by the class. In the latter case, they are not trivial, or so called marker interfaces. The first cluster in the generic structure presented in the figure, comprises the methods `method1()` and `method2()`, which use `attr1`, `attr2` and `attr3`, and the second cluster comprises methods `method3()` and `method4()`, which use `attr1` and `attr4`.

Figure A.17 depicts a rather favorable situation, in which the methods of the class can be assigned more or less unambiguously to clusters of data. This denotes an strong object oriented topology of the underlying abstract model. In the worse scenario, the class may appear to be functionally cohesive, in the sense that an unambiguous association of methods to the attribute clusters which describe the encapsulated concepts is not possible. Such instances are harder to diagnose reliably because heuristics must rely on indirect manifestations in the structure, such as class size and the way in which clients use the class. Thus, a schizophrenic class is likely to be large in terms of data that is defined, and to have a relatively large number of clients in the system.

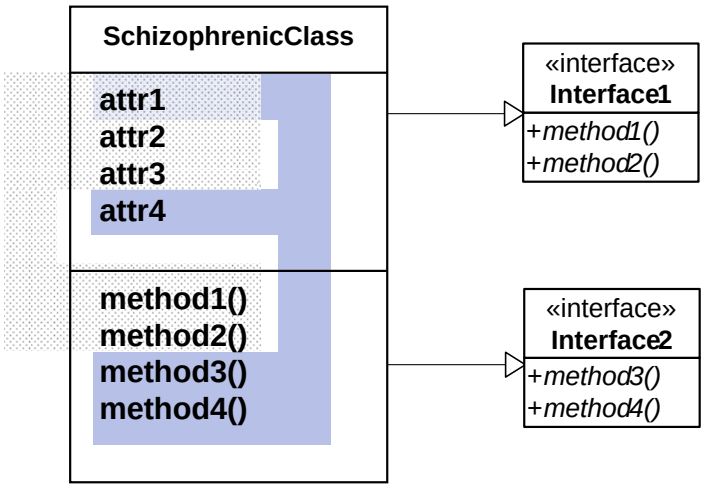


Figure A.17.: Pathological structure for *schizophrenic class*

The pathological structure described above has a negative effect on the maintainability of the individual abstractions contained within, because it hinders understanding and changing them in isolation.

A.6.5. Reference Structure

As shown in figure A.18, the original class has been broken up, based on the abstractions encapsulated by the class. Again, the figure illustrates the more favorable situation, in which we have a strong identity based decomposition of class members, based on the identities of the encapsulated abstractions.

With respect to attributes that are used by two or more functional clusters, we have two possibilities. In the first scenario, the attribute can unambiguously be assigned to one of the newly created abstractions (figure A.18 a). In this case, the attribute should be moved into its natural home, a possible guiding heuristic being that the abstraction that changes the attribute should also own it. All methods that use attributes belonging to a foreign class, may need to be split according to the class identities and associated responsibilities (see A.5). In case there are still foreign methods that need read access to the attribute, a getter accessor method can be created.

The second scenario is when one or more commonly used attributes really don't belong in any of the abstractions implementing the interfaces, but semantically forms a helper class for them (figure A.18 b). In this case as well, we should avoid providing accessor methods. Rather, the methods that use foreign attributes should be investigated, in order to identify possible higher level services that could be moved along with the attributes into

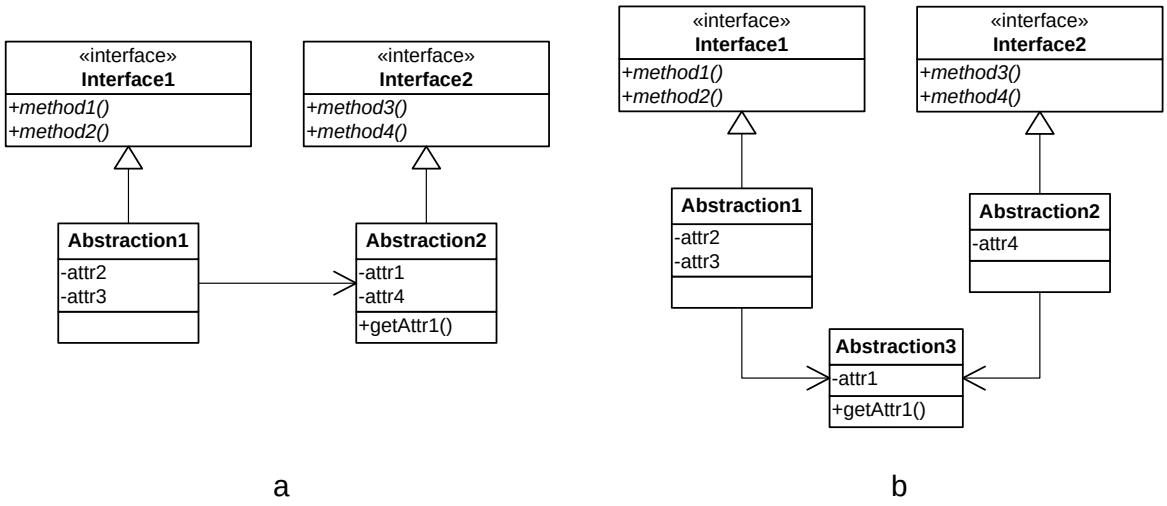


Figure A.18.: Reference structure for *schizophrenic class*

the helper class. Thus, we ensure that functionality is properly decomposed, according to each class' identity and associated responsibilities.

In any of the two situations the original class can provide a "facade" to the newly created classes.

A.6.6. Diagnosis Strategy

Search space

All classes in the system.

Initial filter

Any non trivial class, that is not cohesive with respect to data use of its methods.

Indicators

Indicator 1: *The class defines a large number of attributes.*

Indicator 2: *The class is heavyweight, in the sense that it is very large and has a high complexity.*

Indicator 3: *The class constitutes a "bottleneck", in the sense that a significant proportion of all classes in the system depend on it. A class depends on another if it references it.*

Indicator 4: *The class exhibits distinct personalities with respect to disjoint groups of clients, either by explicitly implementing two or more non-trivial interfaces, or*

by having disjoint groups of clients that use disjoint fragments of the class' public interface.

Context matching

Question 1: *The maintainer must confirm that the suspected class captures at least two key abstractions that are not related through specialization, with data and functional methods of their own.*

Question 2: *The maintainer must confirm that the two or more design abstractions captured by the class are expected to change independently from one another. If the class in question represents a facade or is intended as a library of utility functions for a subsystem, the user must ensure that separating the individual abstractions enclosed in the class is meaningful and desirable.*

A.6.7. Reorganization Strategy

1: Let O be the schizophrenic class
2: Check that we have an identity based decomposition of data in O, based on the identities of the encapsulated abstractions. // *An action oriented topology in the case of the encapsulated abstractions would require a complete redesign of the fragment, which is outside the scope of design flaws in general (see 6.1.1)*
3: Encapsulate all attributes in O with public accessors // *The public visibility is only temporary, in order to make moving functionality around easier*
4: Identify all the abstractions A_i, that need to be separated and establish their future interfaces
5: Create empty classes that correspond to each of A_i
6: **if** O has subtypes // *we assume that they contain valid specializations for one or more of the abstractions contained in O* **then**
7: Establish those abstractions from A_i that are affected by each specialization of O
8: Create appropriate subtypes for the classes that correspond to these abstractions
9: **end if**
10: **for all** attributes a_i in O **do**
11: Find the natural place for a_i in one of the newly created classes, including helpers, based on the A_i determined before and apply "move field" [Fow99]
12: **if** a_i is an array or collection **then**
13: Decide between keeping the structured type and having an association multiplicity of 1 to the host class, or increasing the association's multiplicity and replacing the collection or array with only one of its elements. In the latter case, the class interface and the implementation of the facade need to be adapted accordingly
14: **end if**
15: **end for**

16: **for all** methods m_i in O **do**
17: **if** m_i can be unambiguously assigned to one of the new classes **then**
18: Apply "move method" [Fow99] to move m_i's body to the respective class
19: **if** m_i is specialized in one of O's subclasses **then**
20: Apply "move method" [Fow99] to move the overriding method into the appropriate specialization
21: **end if**
22: **else**
23: Apply "extract method" and "move method" [Fow99] to break up the original method, based on the attribute clusters that determine the encapsulated abstractions, and reunite functionality with its associated data
24: **if** m_i was previously specialized in one of O's subclasses **then**
25: Apply "extract method" and "move method" [Fow99] to break up the original overriding method, based on the attribute clusters that determine specializations of the encapsulated abstractions, and reunite functionality with its associated data
26: **end if**
27: **end if**
28: **if** m_i had public visibility **then**
29: Implement "facade" [GHJV96] method in O, delegating to the appropriate abstraction(s).
30: **end if**
31: **end for**
32: Create initialization methods in the facade O, or adapt its constructors to instantiate and wire together all newly defined classes and their specializations
33: Reduce data and accessor visibility as much as possible in all of the newly created classes

A.7. Embedded Features

A.7.1. Description

Inheritance allows the definition of a hierarchy of specialized versions of an abstraction. By using a reference to the root type, other classes in the system can use any of the specialized variants without knowing their exact type. Furthermore, the object that the reference points to, may be dynamically exchanged with other objects of any of the specialized types.

But what if we wanted to dynamically give or withdraw new responsibilities or features, to one particular object? We cannot achieve this using inheritance alone, because object state would be lost every time we replaced instances. Furthermore, if the number of individual features were large, and we wanted to combine them, the number of specializations, and therefore classes in the system, would explode uncontrollably. The most flexible solution for this scenario involves combining both the inheritance and composition mechanisms, in accordance with the decorator design pattern.

A class is said to suffer from "embedded features", if it uses attributes that represent on/off switches for optional features of the class. The attributes are explicitly checked in order to choose the desired behavior in each case. Figure A.19 illustrates this situation.

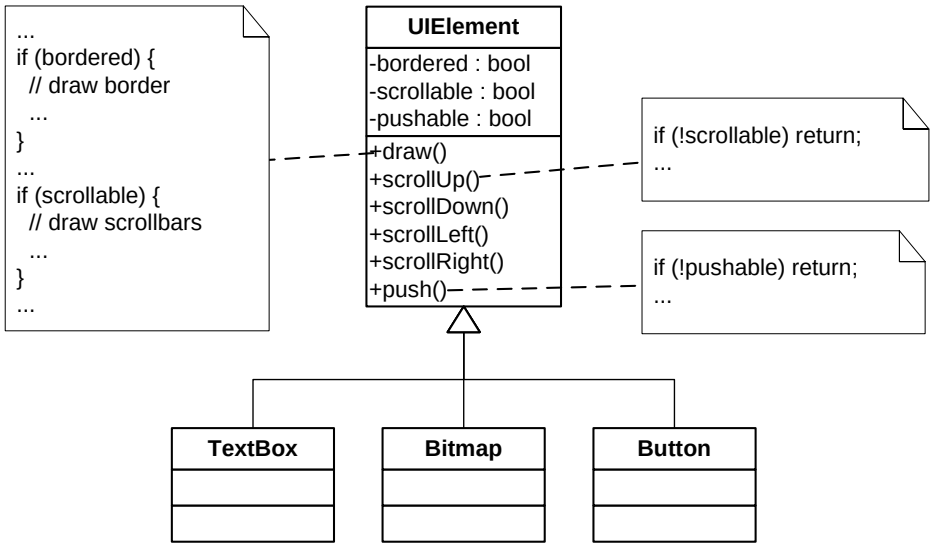


Figure A.19.: An example of *embedded features*

In the example shown in the figure, the abstractions that need to be configurable are elements of a graphical user interface, whose common root class is `UIElement`. The fea-

tures that an `UIElement` is expected to support are: drawing a 3D border around the element (the attribute `bordered`), scrolling support (the attribute `scrollable`), and support for push actions (the attribute `pushable`). As we can see in the figure, the implementations of the element's operations contain blocks that check the current status of each feature in order to provide the desired behavior.

The structure described above negatively affects maintainability, because it mixes code that belongs to the features with the code of the base abstraction. Thus, it becomes harder and harder to add new features, as well as understand and change any of the features in isolation. In addition, in the particular situation depicted in figure A.19, it is not possible to change the "layering" of the bordering and scrolling features, without changing the implementation. Thus, we must for example statically decide between having the scroll bars outside, or inside of the 3D border.

A.7.2. Context

Design intent

You want to allow clients to dynamically enable or disable one or more optional features on instances of a class, or family of classes. The class, or hierarchy of classes represent a valid abstraction or specialization hierarchy in the application's design.

Strategic closure

You expect further features to be added in the future, changes to occur to the existing features, or you need to be able to dynamically change the layering of the features. The public interface of the base abstraction is either not expected to suffer frequent changes, or changes are only expected in methods that are not related to any of the optional features.

A.7.3. Imperatives

In order to maximize maintainability in the context described above, we need to separate the implementation of the base abstraction from the implementations of the features, and the implementations of the features from one another. In order to maximize the flexibility in combining several features, the choice of layering should be left entirely to the clients, and not hardwired in the base abstraction.

A.7.4. Pathological Structure

The pathological structure of the flaw is very simple, and is schematically depicted in figure A.20. The operations of the base abstraction employ simple conditionals which

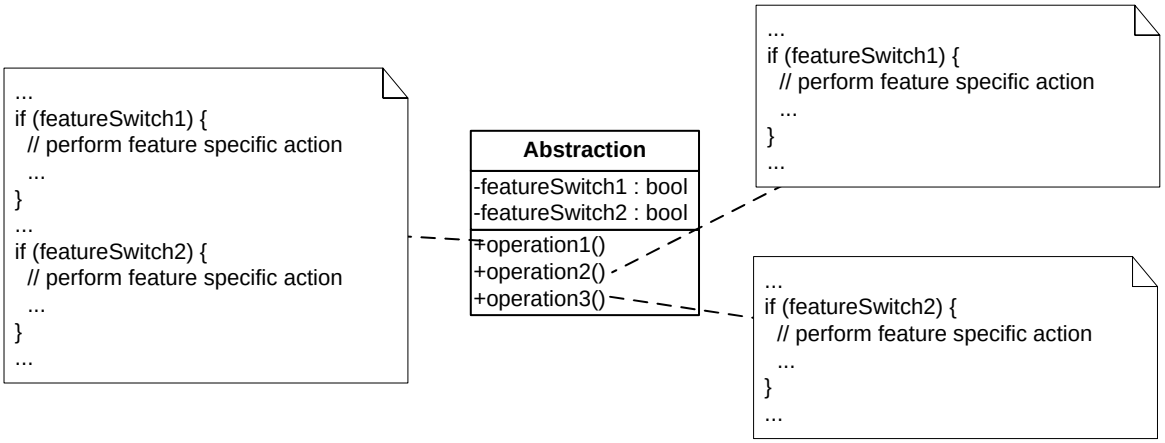


Figure A.20.: Pathological structure for *embedded features*

check the value of the corresponding on/off switch, every time feature specific behavior might be desirable.

The pathological structure is hardly maintainable, because it mixes the implementation of the base abstraction with the implementations of the optional features. In addition, the ordering of feature specific actions, such as those in `operation1()` is fixed, and therefore impossible to alter at runtime. These aspects contradict the imperatives defined in the previous section.

A.7.5. Reference Structure

The reference structure for the design flaw "embedded features" corresponds to the decorator design pattern ([GHJV96]), and is illustrated in figure A.21. The original abstraction is extended with the class `AbstractDecorator`, which is a degenerate composite, in the sense that it composes exactly one instance of its parent. `AbstractDecorator` also serves as a base class to a number of classes, each capturing a unique feature. The abstract decorator (and therefore all concrete decorators) fully support the interface of the base abstraction. The default behavior of these methods is to transparently delegate to the contained instance of the base abstraction. Each feature can override one or more such operations, as well as extend the inherited interface, but should also call the default implementation. This allows clients to wrap several decorators around the object, in order to enable a combination of features. The wrapping order determines the logical layering of the features. Clients are responsible for ensuring that potentially invalid feature combinations or layering are avoided.

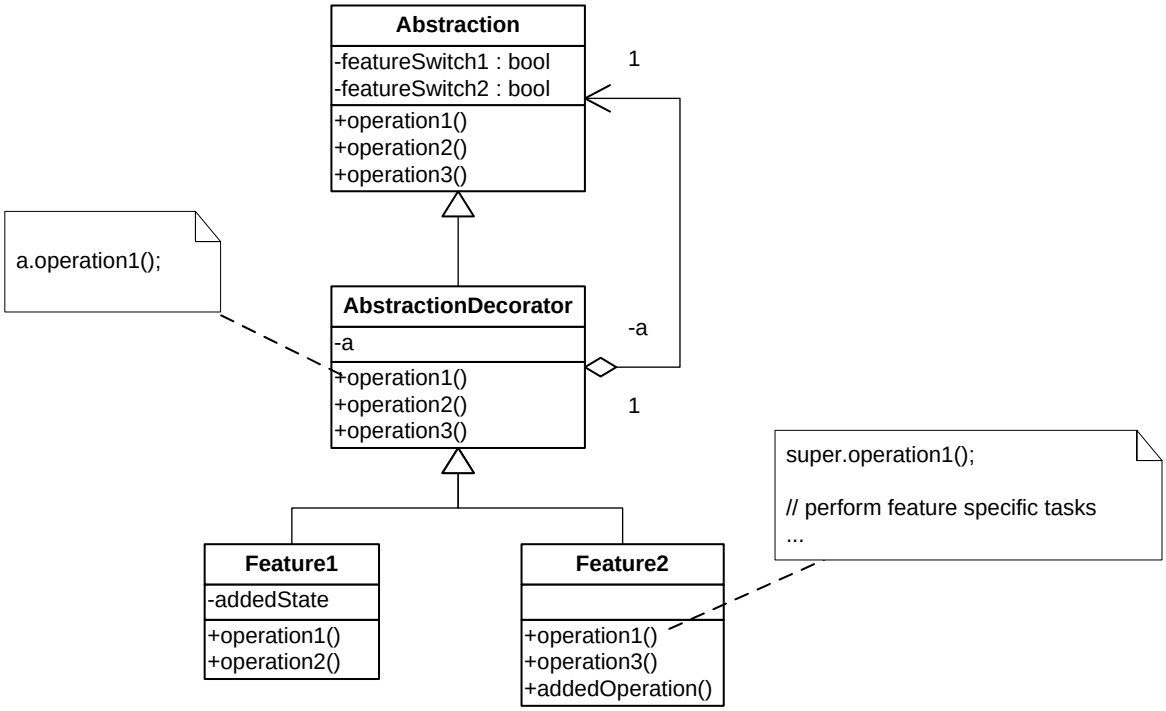


Figure A.21.: Reference structure for *embedded features*

A.7.6. Diagnosis Strategy

Search space

All classes in the system.

Initial filter

The class defines at one attribute that appears to represent an optional feature of the class. In other words, the attribute has a primitive, ordinal type, and it is written to either in a constructor, a method whose name contains "initialize", "setup" or "configure", or from outside the class (either directly or through accessors), but not from other non-accessor methods of the class. In addition, the attribute is checked exclusively in simple conditional statements (`if` or `if-else`), in methods of the class.

Indicators

Indicator 1: *One or more of such attributes have a boolean type.*

Indicator 2: *Code that corresponds to a branch in one or more of the identified simple conditionals, acts as a filter or otherwise changes the return value of the method in which it resides, by either containing a return statement or by writing to a variable*

that is used as the return argument of the method.

Indicator 3: *The suspected class has a higher than average cyclomatic complexity.*

Context matching

Question 1: *It must be confirmed that the classes represents a valid abstraction in the design of the system.*

Question 2: *The maintainer must confirm that the class attributes identified by the initial filter represent on/off switches for optional features provided by the suspected class.*

Question 3: *The maintainer must confirm that new features are expected to be added in the future, changes are expected to occur to the existing features, or the need to dynamically change the logical layering of the features may arise.*

Question 4: *The public interface of the class is either not expected to suffer frequent changes, or changes are only expected in methods that are not related to any of the optional features.*

A.7.7. Reorganization Strategy

1: Let C be the class containing the embedded features, and the so called enclosure type be the interface that declares all of the public methods needed by clients of C
2: **if** E is defined by a superclass or implicitly defined by C **then**
3: Extract an interface I defining the enclosure type
4: Make C explicitly implement I
5: **end if**
6: Identify F, the set of optional features implemented in C
7: **if** F contains more than one feature **then**
8: Define a base decorator class D (may be abstract), which implements I and delegates to an internal instance of the type C
9: Define concrete decorator classes for all features in F, as subclasses of D D
10: **else**
11: Define a concrete decorator class, which implements the interface of the enclosure type and has delegation methods to an internal instance of type C
12: **end if**
13: **if** C has subclasses that override optional features in C **then**
14: Move feature overriding code from the subclass into specializations of the corresponding concrete decorator classes
15: **end if**
16: **for all** features f_i in F **do**

17: Move the corresponding code blocks out of the conditional statements from *C* into the corresponding concrete decorator class. If there are methods that are called exclusively from such code blocks, move these methods to the decorator class as well
18: Create appropriate constructors in the concrete decorator class
19: **end for**
20: Remove the configuration parameter from *C*
21: Adjust clients by replacing object parametrization with proper instantiation of the decorator and decorated classes. If several decorators need to be layered and there are invalid combinations provide and make use of factory methods in the decorated class
22: Adapt all clients that rely on the identity of the decorated object to eliminate this dependency // *This is necessary because decorated objects share the interface of the original object, but not its identity*

A.8. Containment by Inheritance

A.8.1. Description

The object oriented paradigm provides two basic mechanisms to facilitate the reuse of behavior from an already implemented class, in the implementation of a new class. The first mechanism is the inheritance relation, intended to express a vertical generalization between an abstraction and its specialization. From the clients' point of view, the two classes are said to be of the same kind, because the derived class automatically inherits and supports all operations defined in the interface of its base class. In addition, their instances can be used interchangeably by referring to them through the common interface.

The second mechanism is composition, which expresses the containment and the "uses" relations, between the container class and the contained class. In this case, a container instance "uses" an instance of the contained class in order to realize its own responsibilities.

The design flaw "containment by inheritance" refers to the situation in which a class inherits from another class, but the inheritance relation does nor represent a valid specialization. In other words, the derived class "uses" the inherited bits of the base class' implementation, but does not need, or even want to support its public interface. Using instances of the two classes interchangeably would not be meaningful from the viewpoint of other classes in the design.

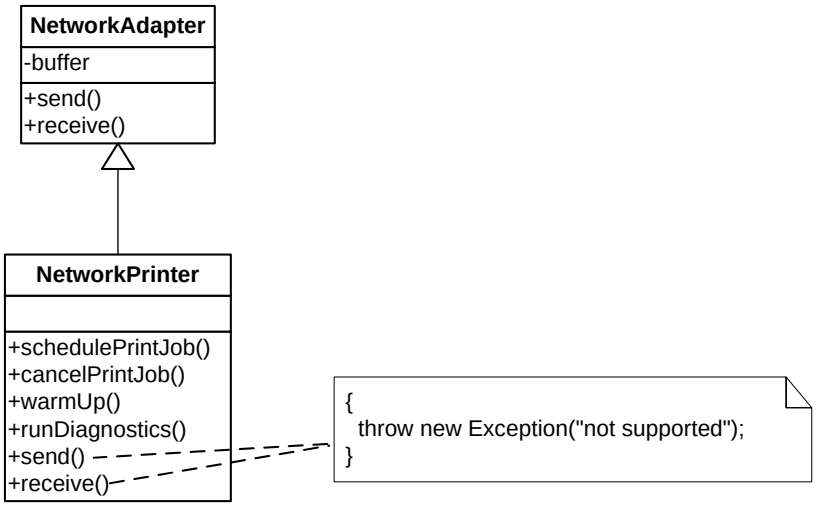


Figure A.22.: An example of *containment by inheritance*

An example instance of this design flaw is depicted in figure A.22, in which a

`NetworkPrinter` uses a `NetworkAdapter` in order to accomplish the responsibilities, assigned to it by design. The inherited operations are logically not meaningful for the derived class, and have been overridden with methods that always return an error. The example shown in the figure employs what is called public inheritance, which is the standard type of inheritance in the object oriented paradigm. In certain languages, such as C++, dedicated variants of inheritance may be provided for facilitating implementation inheritance, but without inheriting the base class interface. In this case, the two overrides would not be necessary.

The structure presented above contradicts the semantics of inheritance and therefore may lead to confusions in understanding a design. In addition, someone who wants to understand the implementation of the derived class is forced to also check the base class. Finally, details of the base class implementation are needlessly revealed to semantically unrelated classes. This means that changes to the internals of the base class are harder to accomplish, because they may potentially affect the derived class, its subclasses, and possibly their users throughout the system.

A.8.2. Context

Design intent

While implementing a new class, you want to reuse (part of) the behavior of an already implemented class. The two classes represent valid abstractions in the application's design. From the perspective of your class' users and within the confines of the application domain, the relationship between the two abstractions is rather a "has" and "uses" type relation, not a valid specialization.

Strategic closure

You expect changes in the original class, or any other maintenance activities that would require an understanding of the design fragment involving the two classes.

A.8.3. Imperatives

One of the fundamental principles in object orientation is the separation of the public interface from the details of the implementation. Therefore, a black box style design, where the publicly exposed details are confined to a minimal interface, is always preferable to a white box design, which increases the risk of unwanted dependencies from other classes in the system. In the context described above, hiding the internals of the original class from the new class as well as from the users of the new class, is important in order to decrease design fragility due to unwanted coupling. A black box design also minimizes the effort spent in understanding the new class, because it limits the number of abstractions (interfaces) that need to be understood, to one.

A.8.4. Pathological Structure

The pathological structure that characterizes the present design flaw is illustrated in figure A.23. The names of the classes reflect their roles with respect to the containment relation

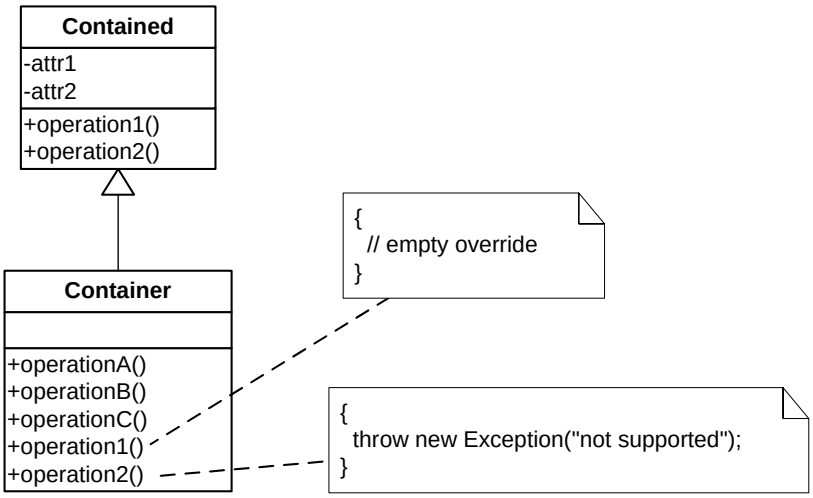


Figure A.23.: Pathological structure for *containment by inheritance*

described in the context. The `Container` extends the `Contained` class in order to acquire its implementation. It also inherits the latter's public interface, which it does not need/want to support. The methods of this interface may therefore be overridden with empty methods, and methods that throw a specific exception or otherwise return an error.

Since the inheritance mechanism employed in the pathological structure corresponds to a white-box reuse of the contained class' implementation, the pathological structure stands in clear contradiction to the imperatives defined above. As already mentioned in the description, certain languages (such as C++) define private and protected inheritance, which facilitate implementation inheritance without publishing the base class interface. According to [Rie96] however, these represent warped forms of containment that may produce confusion, are harder to maintain, and should therefore be avoided.

A.8.5. Reference Structure

As illustrated in figure A.24, inheritance is replaced by composition. This represents the most natural way of expressing the containment relation between a class that wants to use already implemented behavior, and the class that provides the wanted behavior.

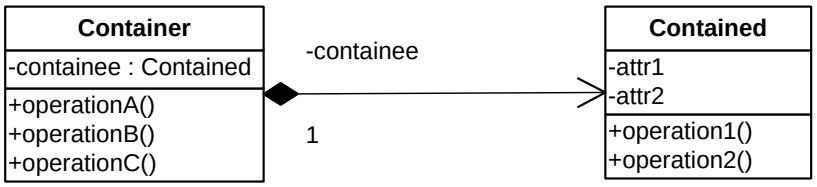


Figure A.24.: Reference structure for *containment by inheritance*

A.8.6. Diagnosis Strategy

Search space

All pairs of classes.

Initial filter

Classes must be related through a direct inheritance relation.

Indicators

Indicator 1: *The derived class is a "tradition breaker" in the sense that it adds a significant amount of new methods, or it extends several supertypes.*

Indicator 2: *The presence of "refused bequest" (interface form) [Fow99] in at least one method that is part of the inherited public interface of the base class. There are two possible scenarios. In the first one, the derived class employs private or protected inheritance. In the second scenario, the derived class employs public inheritance but overrides the inherited public interface with methods that are either empty, are limited to throwing exceptions, or returning an error code.*

Indicator 3: *The base class stores library code. We have the following two possibilities: either it is intended as a library for arbitrary classes, or the subclass uses it as such. In the first case, the base class is large and complex, yet has a very low internal cohesion. In the second case, the superclass is not simple (i.e. it has a certain minimal size and complexity), yet the subclass does not override any of the inherited services.*

Context matching

Question 1: *It must be confirmed that the two classes represent valid abstractions in the design of the system.*

Question 2: *It must be confirmed that the inheritance relation between the two classes does not represent a valid specialization ("kind of" relation).*

Question 3: *It must be confirmed that from the viewpoint of the class' users, the logical relation between the two abstractions is a "uses"-type relation, which can be modeled through containment.*

Question 4: *It must be confirmed that changes in the original class, or any other maintenance activities that would require an understanding of the design fragment involving the two classes, are expected to happen in the future.*

A.8.7. Reorganization Strategy

1: Let B be the subclass playing the role of the container class.
2: **if** B only uses part of the superclass' functionality **then**
3: Apply refactoring "extract class" [Fow99] on the needed functionality needed by B. Let this new class be A
4: **else**
5: Let A be the superclass, or the class extracted from it that plays the role of the contained class, and
6: **end if**
7: If A is abstract and B is concrete, create an implementation class as a subtype of A, by extracting all implementations of abstract methods from B into the newly created subtype [Fow99]
8: Replace the inheritance relation between A and B with composition.
9: **if** B had subclasses that were overriding methods in A, or A had behavior specific to subtypes of B **then**
10: Create corresponding subtypes of A, containing the overridden versions of the methods or type specific behavior, in accordance with the design pattern "bridge" [GHJV96]
11: Create factory methods that allow clients to instantiate the desired class combination
12: **else**
13: Implement the instantiation of A or its concrete implementation in B's constructor
14: **end if**
15: Adapt the visibility of attributes and methods in A and its subtypes, to provide needed services to B
16: If the objects of A are and are expected to remain stateless, turn the class into a "singleton" [GHJV96]

A.9. Premature Interface Abstraction

A.9.1. Description

Data, behavior and interface should be defined at the proper level in any inheritance hierarchy. There are two possible ways of going wrong. In the first scenario, some of the data, behavior and/or interface may be defined too low in the hierarchy. In other words, we have commonality that has not been recognized as such, and the result is duplication. Riel tells us to "factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy" [Rie96]. In the second scenario, commonality is defined too high, which may result in its rejection in some of the members of the hierarchy. Of course, throughout the life cycle of a non trivial system, it may happen that a hierarchy starts off well designed, but "evolves" into one of the two cases, and consequently needs to be restructured.

The design flaw "premature interface abstraction" corresponds to the second scenario, in which a piece of the public interface (i.e. a method) is defined too high in the inheritance hierarchy, as illustrated by the example in figure A.25, and are consequently rejected (i.e. "refused bequest", [Fow99]) by some of the concrete specializations.

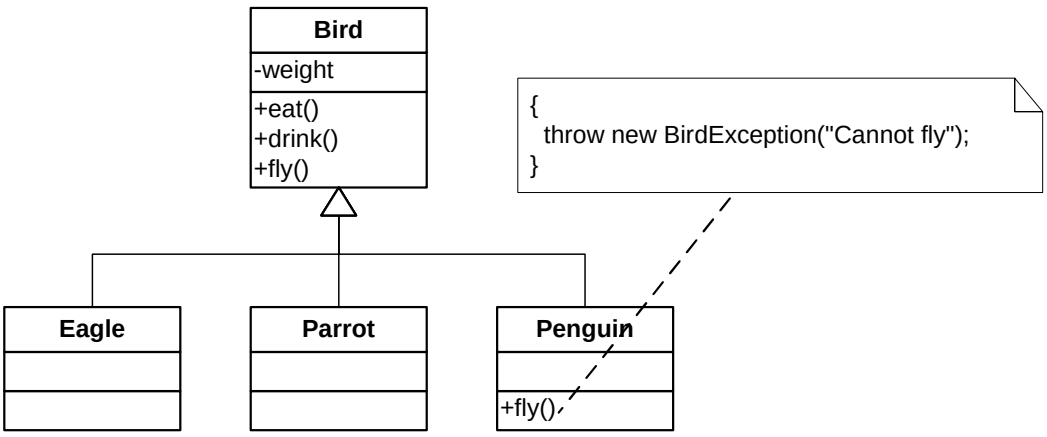


Figure A.25.: An example of *premature interface abstraction*

Suppose our design originally contained the superclass `Bird`, and the two subclasses `Eagle` and `Parrot`. By using the principles and heuristics of object orientation, the common interface containing `eat()`, `drink()` and `fly()`, has been factored in the base class. At a later time, the class `Penguin` was added. Since penguins are a special kind of birds, the class belongs at the bottom of the hierarchy. The developer then noticed that the default implementation of `fly()` caused unexpected behavior, and he decided to override this method with one that signals this error, by throwing an exception.

The design presented above violates the Liskov substitution principle of object oriented design, which forms the basis for the mechanism of polymorphism. The interface of `Penguin` does not reflect what the class actually does, and may therefore confuse its users. Also, if further non flying birds are added to the hierarchy, the problem expands to them as well. If the maintainers of the system later decide to change the exception that is thrown, they must remember to update all affected classes.

A.9.2. Context

Design intent

Given a valid taxonomy expressed using an inheritance hierarchy, you want to achieve a distribution of interface declarations throughout the types that form the hierarchy, in a way that naturally reflects the original taxonomy.

Strategic closure

You either expect changes to the type that rejects part of the inherited interface, or any change that would require the maintainer to understand and use the specialization hierarchy.

A.9.3. Imperatives

In order to maximize maintainability in the context described above, the public interfaces of the classes that form an inheritance hierarchy, should be defined at the proper level in the inheritance tree. The proper level is the one that guarantees the absence of logical contradictions between an abstraction's identity and the characteristics inherited from its generalizations.

A.9.4. Pathological Structure

As illustrated in figure A.26, a type defines one or more methods that are then rejected by one or more of its subtypes (pictured in gray). Starting from each rejecting type, we can form a subtree whose root type is a subtype of the abstraction that defines the rejected interface. These subtrees are encircled in the figure. The first one contains an abstract class and two concrete classes, which both reject the interface formed by `operation2()` and `operation3()`, defined by `Abstraction`. The two rejecters are grayed. The second subtree is formed by its root alone, the class `Concrete4`, which also rejects the two methods. Class `Concrete3` doesn't reject the inherited methods.

At this point, we must impose a restriction: all concrete types (i.e. types that can be instantiated) that are part of the subtrees formed as described above, must reject the interface. Why do we need this restriction? Because if we allowed concrete classes in two

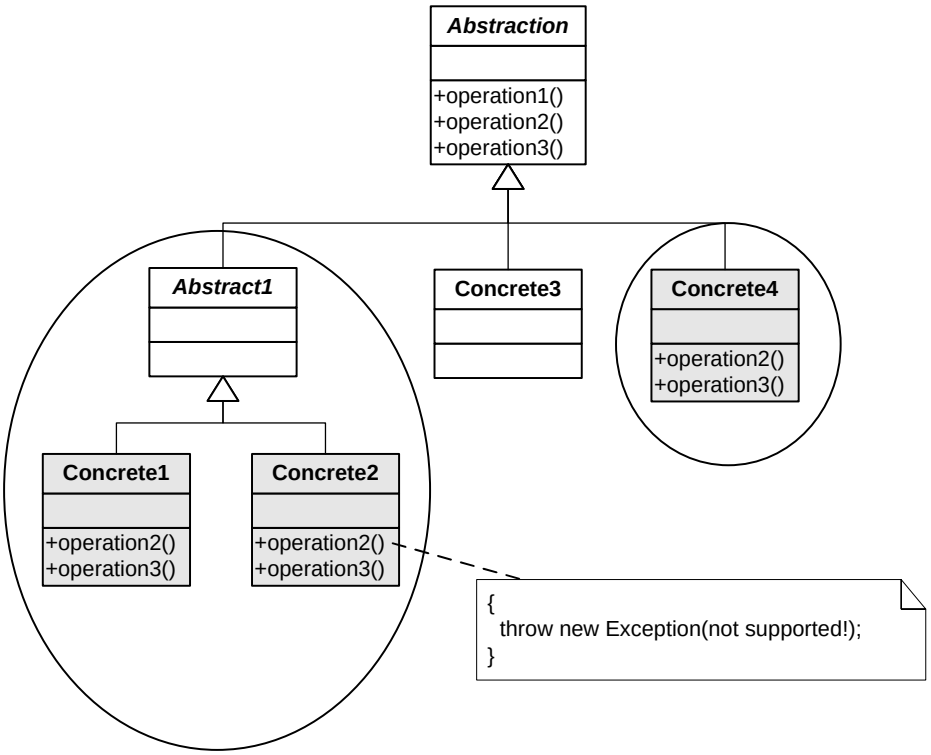


Figure A.26.: Pathological structure for *premature interface abstraction*

or more subtrees to *not* reject the interface, it would mean that the interface represents optional responsibilities, that some of the subtypes may or may not support. These optional responsibilities are therefore not semantically tied to the core responsibilities of `Abstraction`. For example, if the taxonomy referred to birds, as in figure A.25, then a rejected interface dealing with object serialization would obviously not be logically related to the concept of a bird. This situation would warrant the extraction of the corresponding method declarations into a separate dedicated interface for serialization, which could then be implemented only by those classes that need it. But this scenario is not within the scope of the present design flaw, hence the restriction.

The rejection of an inherited interface fragment in a type, signals a logical contradiction between the type's identity and the characteristics inherited from its generalizations, which has a negative effect on the maintainability of the design.

A.9.5. Reference Structure

The reference structure is depicted in figure A.27. Please note that the rejected interface, formed by methods `operation2()` and `operation3()`, has been pushed further down in the hierarchy. If we only had one "adopting" subtree, we could have simply

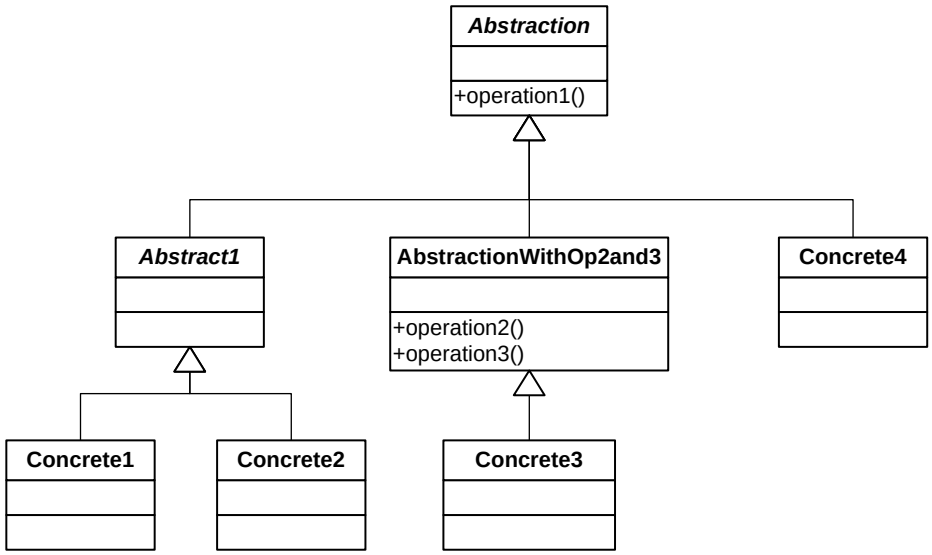


Figure A.27.: Reference structure for *premature interface abstraction*

moved the declarations into the root of this subtree. The figure however shows the general situation, in which a new type (`AbstractionWithOp2and3`) is defined.

A.9.6. Diagnosis Strategy

Search space

All types defined in the system.

Initial filter

The type must have at least two direct subtypes, and must declare an interface (i.e. one or more methods) that is rejected by at least one of its direct or indirect subtypes, which may either be an abstract or a concrete type. Rejection is indicated by the presence of "refused bequest" (interface form) [Fow99] in all the methods that form the inherited public interface, by overriding the inherited interface with methods that are either empty, are limited to throwing exceptions, or returning an error code. Every rejecting type determines a subtree whose root node is a direct subtype of the initial type.

Indicators

Indicator 1: *All concrete classes contained in all of the subtrees described above, reject the interface, either explicitly, or implicitly, by inheriting such overridden methods from an abstract superclass.*

Context matching

Question 1: *It must be confirmed that the inheritance hierarchy represents a valid taxonomy in the application's design.*

Question 2: *It must be confirmed that the rejection of the interface is semantically justified for those subtrees that reject it, and that the interface is meaningful for the adopting subtrees.*

Question 3: *It must be confirmed that changes to the type that rejects part of the inherited interface, or any change that would require the maintainer to understand and use the specialization hierarchy, are expected to happen in the future.*

A.9.7. Reorganization Strategy

1: Let C be the class defining the rejected interface I
2: **if** I is adopted by two or more subtrees determined as described above **then**
3: Create an intermediary type T, as a direct subtype of C. T can be either an interface or an abstract class, as appropriate.
4: Make all adopting subtrees inherit from T
5: Apply refactoring "push down method" [Fow99] on the methods composing I to move them into T
6: **else**
7: Apply refactoring "push down method" [Fow99] on the methods composing I to move them into the root of the adopting subtree
8: **end if**
9: Remove the methods corresponding to I in all rejecting types

A.10. Collapsed Method Hierarchy

A.10.1. Description

Data, behavior and interface should be defined at the proper level in any inheritance hierarchy. One possible mistake is to define some of the data, behavior and/or interface too low in the hierarchy. This results in commonality that has not been recognized as such, and is therefore duplicated in the nodes of the hierarchy. Riel tells us to "factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy" [Rie96].

The other possibility is to define the commonality too high in the inheritance tree. In section A.9 of this catalog, we described a flaw that captures the case of prematurely defined interfaces.

The design flaw "collapsed method hierarchy" deals with a similar kind of problem, but regarding behavior (i.e. parts of a method hierarchy's implementation). As in the case of interfaces, this flaw is not always the result of poor initial design, but sometimes of the system's evolutionary history. Figure A.28 presents us with an example instance of the "collapsed method hierarchy" design flaw, in a hypothetical logical expression evaluator.

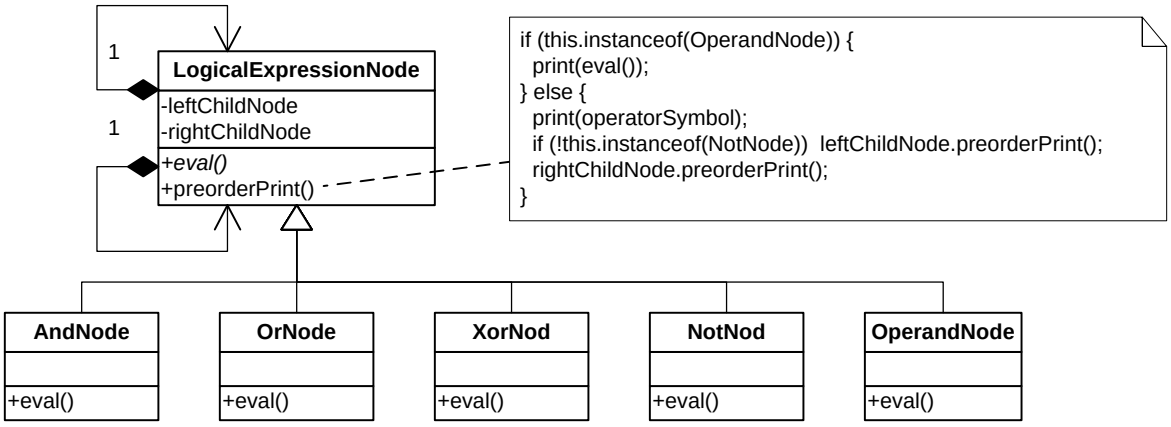


Figure A.28.: An example of *collapsed method hierarchy*

In the example, logical expressions are modeled as trees, that consist of `LogicalExpressionNode`s. Each node may have two, one or no child nodes, in case they represent a binary operator, unary operator or operand, respectively. The hierarchy's root defines an abstract `eval()` method, which is implemented in each specialized node, according to its particularities. Developers then decide to add the `preorderPrint()` operation. Since the implementation of this operation would be

identical for most of the concrete node types, the developer chooses to implement it once, in the superclass, treating the two exceptional cases explicitly. By doing that, he pulls specific behavior up into higher levels of the hierarchy.

"Collapsed method hierarchies" have a negative influence on maintainability because they concentrate specialized behavior from potentially several subclasses in the superclass implementation, making the operation harder to understand and change. In addition, the superclass becomes dependent on its subtypes. Thus, changes to the lower levels of the hierarchy may require changing methods in the superclass as well.

A.10.2. Context

Design intent

Given a semantically valid specialization hierarchy, modeled as an inheritance hierarchy, you want to distribute behavior (implementation) throughout the types that form the hierarchy, in a way that naturally reflects each type's responsibilities.

Strategic closure

You expect changes to occur that would require the maintainer to understand or modify the superclass or its subclasses.

A.10.3. Imperatives

In order to maximize maintainability, the distribution of behavior (i.e. implementation) among the classes that form a specialization hierarchy, should be in accordance with the specialization relationships between these classes. In other words, a class should not implement behavior that falls within the responsibilities of one or more of its descendants. A class should only contain code that realizes behavior that is common to all its subclasses. A superclass should not be dependent in any way on any of its subclasses [Rie96].

A.10.4. Pathological Structure

As depicted in figure A.29, the pathological structure is characterized by the fact that a superclass contains code that is specific to one or more of its direct or indirect subclasses. In order to guarantee that such code is only executed for the right subtype, the implementation contains explicit checks on the object's type, by using a reflection mechanism, as in the case of method `operation3()`.

The structure depicted in the figure is in contradiction with the imperatives stated above, because the more general abstraction contains code that is not common for all its subclasses. In order to do that, it must "know" at least some of its subtypes, therefore introducing an unwanted downward dependency (i.e. from general to special).

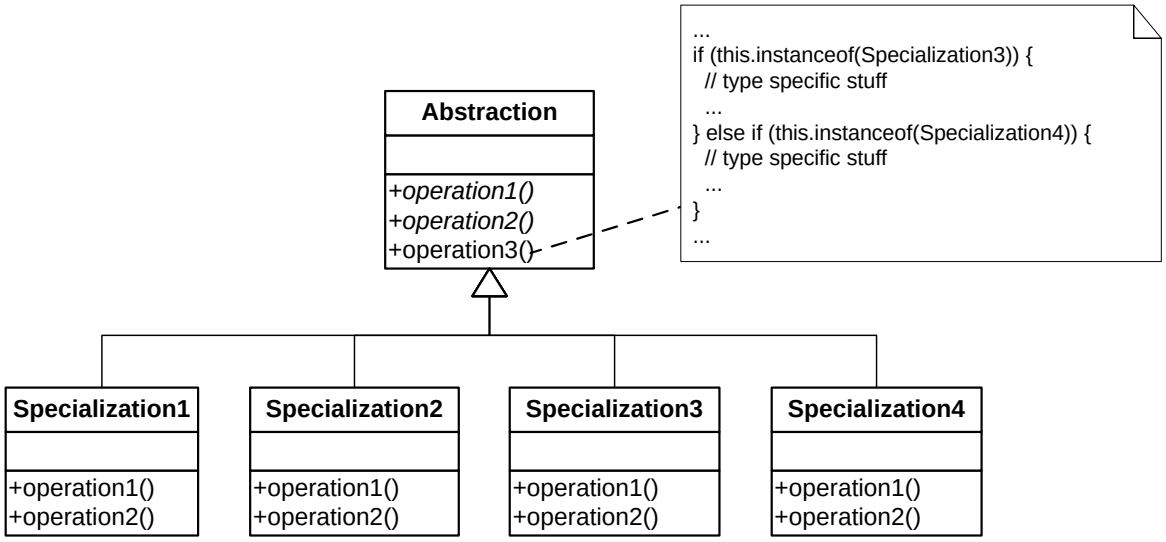


Figure A.29.: Pathological structure for *collapsed method hierarchy*

A.10.5. Reference Structure

The reference structure is illustrated in figure A.30. The explicit checks have been replaced with a call to a hook method, as described by the "template method" design pattern [GHJV96]. Those subclasses that need special treatment override the hook methods as desired. Thus, type specific behavior is moved from the superclass to the appropriate specializations in the hierarchy.

A.10.6. Diagnosis Strategy

Search space

All classes in the system.

Initial filter

The class must have at least one descendant. In addition, it contains at least one conditional statement that uses runtime type identification in order to check the self object's type, comparing it against the type of one of its descendants.

Indicators

Context matching

Question 1: *It must be confirmed that the supertype and the subtypes it depends on represent valid abstractions in the design of the system, and that the inheritance relations that link them represent valid specializations.*

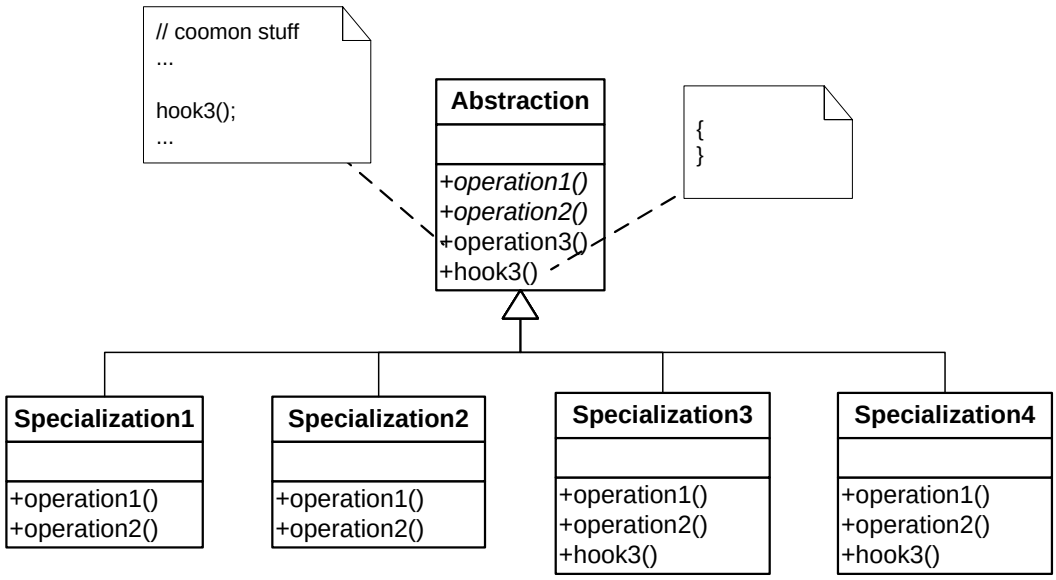


Figure A.30.: Reference structure for *collapsed method hierarchy*

Question 2: *It must be confirmed that changes that would require the maintainer to understand or modify the superclass or its subclasses are expected to happen in the future.*

A.10.7. Reorganization Strategy

1: Let C be the class containing conditional constructs identified by I1
2: **for all** methods m_i in C, that contain the conditional constructs identified by I1 **do**
3: Let S be the set of subclasses that m_i depends on
4: Remove those classes s_i from S, that override m_i without calling the superclass version
5: **for all** conditional structures in m_i **do**
6: Create an empty hook method in C, in accordance with the design pattern " template method" [GHJV96]
7: Move type-specific behavior from the current conditional structure into the corresponding overrides of the default hook method, in the respective subtypes in S
8: If the type-specific behavior from m_i calls other methods in S, check to see if those methods can be also moved into the subtype
9: Replace current conditional structure in m_i with a call to the current default hook method

10: **end for**
11: **end for**
12: If default hooks are not meaningful for some of the subtypes, define an intermediate level in the hierarchy, in order to group together those specializations that do not need special treatment under a common supertype. The same can be done with those that need it.

Bibliography

[AFC98] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th IEEE International Workshop on Program Comprehension (IWPC)*, pages 153Ű–160, 1998.

[Bae99] Holger Baer et al. The FAMOOS object–oriented reengineering handbook, 1999.

[Bas92] V. R. Basili. Software modeling and measurement: The goal question metric paradigm. Technical Report UMIACS-TR-92-96, University of Maryland, College Park, MD, September 1992.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd Internatioinal Conference on Software Engineering (ICSE)*, pages 592–605, 1976.

[BC87] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. In *OOPSLA '87 Workshop on Specification and Design for Object-Oriented Programming*, 1987.

[BCR94] V. Basili, G. Caldiera, and H. D. Rombach. *Encyclopedia of Software Engineering*, chapter The Goal Question Metric Approach, pages 528–532. John Wiley and Sons, Inc., 1994.

[BD02] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. In *IEEE Transactions on Software Engineering*, volume 28, pages 4–17, January, 2002.

[BK95] J. M. Bieman and B. K. Kang. Cohesion and reuse in an object oriented system. In *Proceedings of the ACM Symposium on Software Reusability*, 1995.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[BMP87] D. Bell, I. Morrey, and J. Pugh. *Software Engineering - A Programming Approach*. Prentice-Hall, New Jersey, 1987.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[BNL03] D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE)*, 2003.

[BS99] D. Blostein and A. Schürr. Computing with graphs and graph rewriting. *Software - Practice & Experience*, 29(3):1–21, 1999.

[BT04] Markus Bauer and Mircea Trifu. Architecture-aware adaptive clustering of OO systems. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR)*, 2004.

[Cas92] E. Casais. An incremental class reorganization approach. In *Procedings of the 6th ECOOP Conference*, 1992.

[Cas94] E. Casais. *Object-Oriented Systems*, chapter Automatic Reorganization of Object-Oriented Hierachies: A Case Study. Chapman & Hall, 1994.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[Cin00] Mel Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, Trinity College, Dublin, 2000.

[Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30*, pages 18–32, 1999.

[Ciu01] Oliver Ciupke. *Problemidentifikation in objektorientierten Softwarestrukturen*. PhD thesis, Universität Karlsruhe, 2001.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CY91] Peter Coad and Edward Yourdon. *Object–Oriented Design*. Prentice Hall, London, second edition, 1991.

[DDN03] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[DeM82] Tom DeMarco. *Controlling Software Projects*. Yourdon Press, 1982.

[DRD99] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1999.

[Dro96] R. G. Dromey. Cornering the chimera. *IEEE Software*, 13(1):33–43, 1996.

[DS98] N. R. Draper and H. Smith. *Applied Regression Analysis*. John Wiley and Sons, 1998.

[DW03] T. Dudzikan and J. Wlodka. Tool-supported discovery and refactoring of structural weaknesses. Master's thesis, TU Berlin, 2003.

[Eas93] A. Eastwood. Firm fires shots at legacy systems. *Computing Canada*, 19(2):17, 1993.

[EL96] Karin Erni and Klaus Lewerentz. Applying design–metrics to object–oriented frameworks. In *Proceedings of METRICS*. IEEE, 1996.

[EPS00] F. B. E Abreu, G. Calo Pereira, and P. Ssousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR)*, 2000.

[Erl00] L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, May/June:17–23, 2000.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter The AGG Approach: Language and Environment, pages 551–603. World Scientific, 1999.

[FKvO98] L. M. Feijs, R. L. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Software - Practice & Experience*, 28(4):371–400, 1998.

[Fow99] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FP96] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, second edition, 1996.

[Fre03] Uwe Freese. Werkzeuggestützte problemidentifikation und -behebung in objektorientierten systemen. Master's thesis, Universität Karlsruhe, 2003.

[Fri95] Lisa Friendly. The design of distributed hyperlinked programming documentation. In *Proceedings of the International Workshop of Hypermedia Design (IWHD)*, 1995.

[GAA01] Yann-Gaél Guéhéneuc and Hevré Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the TOOLS 39*, pages 296–305, 2001.

[Gau21] C. F. Gauss. *Theoria combinationis observationum erroribus minimis obnoxiae*. 1821.

[GBG+06] Rubino GeiSS, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. Grgen: A fast spo-based graph rewriting tool. In *Proceedings of the ICGT*, pages 383–397, 2006.

[GC03] S. Grant and J. R. Cordy. An interactive interface for refactoring using source transformation. In *Proceedings of REFACE03, WCRE Workshop on REFactoring: Achievements, Challenges, Effects*, 2003.

[Gen04] Thomas Genssler. *Werkzeuggestützte Adaption objektorientierter Programme*. PhD thesis, Universität Karlsruhe, 2004.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison-Wesley, 1996.

[Har75] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[HHHL03] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *Proceedings of the 11th IWPC*, 2003.

[HJ01] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[HR96] L. E. Hyatt and L. H. Rosenberg. A software quality model and metrics for identifying project risks and assessing software quality. In *Proceedings of the Product Assurance Symposium and Software Product Assurance Workshop*, 1996.

[HS96] Brian Henderson-Sellers. *Object Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[Huf90] S. Huff. Information systems maintenance. *The Business Quarterly*, 55:30–32, 1990.

[IEE90] IEEE standard glossary of software engineering terminology. The Institute of Electrical and Electronics Engineers, 1990.

[JCS] Mark Harman Robert Hierons Bryan Jones Mary Lumkin Brian Mitchell Spiros Mancoridis Kearton Rees Marc Roper John Clarke, Jose Javier Dolado and Martin Shepperd. Reformulating software engineering as a search problem.

[JF88] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1(2):22–35, 1988.

[JLB02] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in java programs. In *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*. IEEE, 2002.

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007.

[KA83] M. Kogure and Y. Akao. Quality function deployment and cwqc in japan. *Quality Progress*, pages 25–29, 1983.

[KC06] M. Ó Keeffe and M. Ó Cinnéide. Search-based software maintenance. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2006.

[Ker05] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

[KP96] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE)*, pages 208Ű–215, 1996.

[Lak96] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained and Evolutionary Software Visualization*. PhD thesis, University of Berne, 2003.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[LCP01] F. Losavio, L. Chirinos, and M. A. Perez. Quality models to design software architectures. In *Proceedings of the Technology of 38th Object-Oriented Languages and Systems (TOOLS)*, pages 123–135, 2001.

[LD01] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[LD03] M. Lanza and S. Ducasse. Polymetric views: A lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[LD05] Michele Lanza and Stéphane Ducasse. Codecrawler–an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, 2005.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[LHMI07] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 106–115, 2007.

[Lie96] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[LJ05] Seunghak Lee and Iryoung Jeong. Sdd: high performance code clone detection system for large scale source code. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 140–141, New York, NY, USA, 2005. ACM Press.

[LK94] M. Lorenz and J. Kidd. *Object Oriented Software Metrics*. Prentice Hall, 1994.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[LN03] Claus Lewerentz and Andreas Noack. *Graph Drawing Software*, chapter CrocoCosmos - 3D Visualization of Large Object-Oriented Programs, pages 279–297. Springer-Verlag, 2003.

[LS81] B. P. Lientz and E. Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[Lud02] Andreas Ludwig. *Automatische Transformation großer Softwaresysteme*. PhD thesis, Universität Karlsruhe, 2002.

[LW93] B. Liskov and J. Wing. Family values: A behavioral notion of subtyping. Technical Report MIT/LCS/TR-562b, 1993.

[Mar00] Robert C. Martin. Design principles and design patterns. Object Mentor, 2000.

[Mar01] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 39*, pages 173–182, 2001.

[Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2002.

[Mar03] Robert C. Martin. *Agile Software Development. Principles, Patterns and Practices*. Prentice Hall, 2003.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings. 20th IEEE International Conference on Software Maintenance (ICSM)*, 2004.

[MBG06] Naouel Moha, Saliha Bouden, and Yann-Gaël Guéhéneuc. Correction of high-level design defects with refactorings. In *Proceedings of the 7th International ECOOP Workshop on Object-Oriented Reengineering (WOOR)*, 2006.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(3), 1976.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[Mey06] Matthias Meyer. Pattern based reengineering of software systems. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), Doctoral symposium*, 2006.

[MGL06] Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE)*, 2006.

[Mit02] B. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel University, 2002.

[MM01] B.S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2001.

[Moo96] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1996.

[MRG+06] Naouel Moha, Jihene Rezgui, Yann-Gaël Guéhéneuc, Petko Valtchev, and Ghizlane El Boussaidi. Using fca to suggest refactorings to correct design defects. In *Proceedings of the 4th International Conference On Concept Lattices and Their Applications (CLA)*, 2006.

[MRW77] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. *National Technical Information Service*, 1,2,3, 1977.

[MSG99] Thierry Miceli, Houari A. Sahraoui, and Robert Godin. A metric based technique for design flaws detection and correction. In *Proceedings of the International Conference on Automated Software Engineering*, pages 307–310. IEEE, 1999.

[MW99] K. Mens and R. Wuyts. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of Technology of Object-Oriented Languages and Systems Europe*, pages 33–45, 1999.

[Neu00] Rainer Neumann. *Vermeidung spezialisierungsbedingter Probleme in objektorientierten Systemen.* PhD thesis, Universität Karlsruhe, 2000.

[NSW+02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Par94] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PP96] A. Pettorossi and M. Proietti. Future directions in program transformation. *ACM Computing Surveys*, 28(4), 1996.

[PUPT98] L. Prechelt, B. Unger, M. Philippsen, and W. Tichy. Two controlled experiments assessing the usefulness of design pattern information during program maintenance. *Empirical Software Engineering - An International Journal*, 1998.

[PUT+01] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27(12):1134–1144, 2001.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

[Rie96] Arthur J. Riel. *Object–Oriented Design Heuristics*. Addison–Wesley, first edition, 1996.

[RJ04] J. Rajesh and D. Janakiram. Jiad: A tool to infer design patterns in refactoring. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 227–237, New York, NY, USA, 2004. ACM Press.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[RRHK00] D. RAYSIDE, S. REUSS, E. HEDGES, and K. KONTOGIANNIS. The effect of call graph construction algorithms for object-oriented programs on clustering. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC)*, 2000.

[Rug00] Spencer Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9, 2000.

[Sen07] Olaf Seng. *Suchbasierte Strukturverbesserung objektorientierter Systeme*. PhD thesis, Universität Karlsruhe, 2007.

[SGM00] Houari Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of OO design quality and its automation? In *Proceedings of the International Conference on Software Maintenance 2000*, pages 154–162. IEEE, 2000.

[SGMZ98] Benedikt Schulz, Thomas Genssler, Berthold Mohr, and Walter Zimmer. On the computer-aided introduction of design patterns into object–oriented systems. In *Proceedings of the 27^{th} TOOLS conference*, 1998.

[Som00] Ian Sommerville. *Software Engineering*. Addison–Wesley, sixth edition, 2000.

[SSB06] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM Press, 2006.

[SSC96] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 387Ű–396, 1996.

[Sta84] T. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984.

[TK04] Ladan Tahvildari and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: A metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 2004.

[TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering, Benevento, Italy*, 2003.

[Tok99] Lance Tokuda. *Evolving Object-Oriented Designs with Refactorings*. PhD thesis, The University of Texas at Austin, 1999.

[Tri01] Adrian Trifu. Using cluster analysis in the architecture recovery of object-oriented systems. Master's thesis, Universitatea Politehnica Timişoara, Romania, 2001.

[Tri06] Adrian Trifu. Verfahren und werkzeuge zur realisierung von lösungsstrategien. QBench project deliverable, `http://www.qbench.de/QBench/CMS/Members/trifu/AP4EN7undAP6EN2`, 2006.

[TS05] Mircea Trifu and Peter Szulman. Language independent abstract metamodel for quality analysis and improvement of oo systems. In *Proceedings of the 7th German Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*, 2005.

[TSK05] Mircea Trifu, Peter Szulman, and Volker Kuttruff. Qbench systemmetamodell. QBench project deliverable, http://www.qbench.de/QBench/CMS/Members/mtrifu/Systemmetamodell-QBench-V4.0.pdf, 2005.

[vG91] John van Gigch. *System Design Modeling and Metamodeling (The Language of Science)*. Plenum, 1991.

[Vis05] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[Zim97] Walter Zimmer. *Frameworks und Entwurfsmuster*. PhD thesis, Universität Karlsruhe, 1997.

ISBN: 978-3-86644-274-0

www.uvka.de