Université de Montréal

Rule-based Quality Heuristics Formalization and Identification

Par

Fan Yang

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Mémoire présenté a la Faculté des Études Supérieures

en vue de l'obtention du grade de

Maître ès Science (M. Sc.)

en Informatique

Juin, 2007

Université de Montréal

Faculté des Études Supérieures

Ce mémoire intitulé

Rule-based Quality Heuristics Formalization and Identification

Par

Fan Yang

A été évalué par un jury composé des personnes suivantes :

Yann-Gaël Guéhéneuc
président – rapporteur

Houari Sahraoui
directeur de recherche

Nadia El-Mabrouk
membre du jury

Mémoire accepté le :

23 novembre 2007

Résumé

L'évaluation d'une conception par objects est habituellement effectuée par des experts en logiciel à travers une liste d'heuristiques basées sur leurs années d'expérience. Le logiciel qui satisfait cette liste est considéré comme acceptable. Cependant, cette démarche est rarement documentée pour être utilisée par des spécialistes novices, et même si elle l'est, il n'y a aucun consensus sur ce qui est considéré comme acceptable. De plus, l'analyse manuelle des logiciels de grande taille est au mieux fastidieuse, souvent infaisable.

Cette thèse propose une solution au problème de l'évaluation de conception, basée sur les règles. Nous rassemblons et raffinons en particulier un ensemble d'heuristiques de qualité de la littérature. Par la suite, nous définissons une approche pour formaliser ces heuristiques de qualité et les mettons en application sous forme de règles spécifiques aux conceptions décrites dans un méta-modèle de type UML. Ces règles sont automatiquement appliquées à l'information extraite à partir du code en recherchant des conformités aussi bien que des violations de bonnes pratiques.

Nous implantons notre solution dans un outil prototype. Nous avons appliqué cet outil au code source de logiciels industriels et académiques pour réaliser plusieurs études de cas. Ces expériences montrent que notre approche peut automatiquement détecter des conformités et des violations dans des logiciels à objects.

Mots-clés: Heuristiques de conception logiciel, heuristiques de qualité, retro ingénierie, le SDG et le moteur de règle.

Abstract

The evaluation of object-oriented design is usually made by software experts using a list of heuristics based on their years of experience. Software that satisfies these heuristics is considered as acceptable. However, expert's heuristics are rarely documented to be used by inexperienced software specialists and even if so, there is no consensus on what is considered as acceptable. Moreover, the manual analysis of large-scale software is fastidious at best and often infeasible.

This thesis describes a rule-based solution to evaluate the object-oriented design automatically. We particularly collect and refine a set of quality heuristics from the literature. Then, we propose an approach for formalizing these quality heuristics and implement them in the form of rules specific to software designs modeled in a UML meta-model. These rules are automatically applied to information extracted from the code by searching conformances as well as violations.

We present our solution into a prototype tool. We applied the tool in existing source code taken from industrial and academic fields for several case studies. These experiments show that our approach can automatically detect conformances and violations of the quality heuristics from the object-oriented systems.

Keywords: design heuristics, quality heuristics, reverse engineering, SDG and rule engine.

Acknowledgements

First and foremost I wish to thank my supervisor, Professor Houari Sahraoui, for his valuable courses during my studies at Université de Montréal and for his clear advice and encouragement during this project. I would not have been able to do this thesis without his support.

I also want to thank El Hachemi Aklicacem, for his constant guidance. I benefited greatly from formal and informal discussion with him at CRIM (Centre de recherche informatique de Montréal).

I thank CRIM which provided a wonderful research environment and financial support for this thesis.

I wish to thank the "Département d'informatique et recherche opérationnelle", Université de Montréal for the graduate courses and the research environment, and Mariette Paradis for easing the procedure of dealing with Department.

I would especially like to thank my wife Hong Hong and my son Hongyue, for their constant encouragement and support.

Finally, thanks to my parents, for everything.

Table of contents

List of Figures

List of Tables

1 Introduction

The demand for quality software continues to intensify due to our society's increasing dependence on software systems and the often devastating effect that a software error can have in terms of life loss, financial loss or time delays. Today's software systems must ensure consistent and error-free operation every time they are used. This demand for increased software quality has resulted in quality being more of a differentiator between products than it ever has been before. In a marketplace of highly competitive products, the importance of delivering quality is no longer an advantage but a necessary factor for companies to be successful.

1.1 Motivation

While there is uniform agreement that we need quality software, the question of how to measure and assure quality is far from a settled issue. Software metrics have been used to address this issue for several decades; many measures have been proposed in the literature to capture the structural quality of object-oriented code and design, e.g., [McCabe, 1976], [Fenton, 1991], [Chidamber and Kemerer, 1991], [Chidamber and Kemerer, 1994], [Li and Henry, 1999], and [Lorenz and Kidd, 1994]. These measures are being used to address not only different aspects of software quality such as maintainability, reliability, reusability and so forth, but also on the finer granularity of object-oriented properties such as cohesion, coupling and complexity. Once the necessary measurement instruments are in place, the assessment of even large software systems can be thus done very fast, at a low cost, with little human involvement. However, commercial software developers have made relatively little use of them. One of the main drawbacks of metrics is that results are provided in numeric value and are thus less intuitive than the guidelines derived from the practical experience of skilled developers for common software engineers to understand problems and how to locate and fix those problems. Another reason for this is that understanding and applying metrics can be very complicated and is generally only recommended to experienced developers. In addition, there is a lack of

association between the proposed metrics for evaluating the object-oriented design and the daily decisions made by developers.

Consequently, software developers are more inclined to rely on their intuition about the complexity of a system, rather than on some quantified metrics. The process of code or design review is accepted naturally by the majority of software engineers; many organizations execute design reviews by expert designers to improve the design of a large system and to avoid design flaws [Haynes, 1996].

For instance, once an object-oriented developer had completed a design regardless of the methodology used, the developer's main question was, "Now that I have my design, is it good, bad, or somewhere in between?" In asking an object-oriented guru, the developer was often told that a design is good when "it feels right." While this is of little use to the developer, there is a kernel of truth in such an answer. The guru runs through a subconscious list of heuristics, built up through his or her design experience, over the design. If the heuristics pass, then the design feels right, and if they do not pass, then the design does not feel right.

However, there are several concerns about the process of evaluating a design by consulting the object-oriented gurus. First, expert designers are hard to find and expensive to use. Second, this process, the identification of good or problematic OO software constructions, is very difficult to do manually for large systems. We can highlight the following reasons for this difficulty:

- Software systems that need to be reengineered are usually medium / large in size, making manual search for problems unfeasible.
- Systems are developed by different developers or teams. Design problems can be spread across several subsystems and thus cannot be detected locally.
- In most cases, the only reliable source for design information is the source code. Models, when available, either are out of date or are too superficial to support a design analysis. However, the manual analysis of source code limits

the scope of the problems that can be found in a timely and economically way.

- Developers often do not know what kind of problems they should be looking for when they have to evaluate their designs. A common knowledge-base containing potential design problems can provide a valuable support in this case.
- Expertise of gurus is generally designed to be used by human beings not for automated CASE tools.

1.2 General Methodology

To address the aforementioned problems, the general methodology is to analyze the legacy code, specifying frequent design problems or reusable designs as queries and locating the occurrences of these problems or reusable designs in a model derived automatically from source code.

The first step in the methodology is to parse the source code and to produce high-level design information. Doing so leaves the concrete implementation behind and moves towards a higher level of abstraction at which specifications of those problems or reusable designs are given. To be able to express and interpret the information gathered from source code, a meta-model for object-oriented systems has to be set up. This meta-model defines the different entities and relations that may occur in the design of an object-oriented program. The model of a legacy system that conforms to the meta-model can be stored as a graph, as entities and relations, or as predicates. This makes it possible to query and manipulate the model using different query languages.

To detect problematic or reusable structures in the design of a system, the second step in the methodology is to search for certain patterns representing those problematic and reusable designs in the meta-model built from the target system. This means that the methodology has to be able to specify problematic or reusable designs and to

query the model for the existence of a specified problematic or reusable design. The result of such a query is a piece of design specifying the location of the problematic or reusable design in the system. Such a piece of design in the meta-model is often referred to as a design fragment.

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

1.3 Our Approach

The three above-mentioned approaches represent different viewpoints about the same meta-model. Although they are equivalently powerful in a sense and the corresponding models and queries can be converted into each other, each of them has its advantages and shortcomings in certain tasks.

For our specific problem, we adopted the logic programming language approach, expert systems in other words. The reason is that expertise in design problems and reusable designs expressed in our model is mostly captured in literature in the form of natural languages and is primarily designed to be used by human beings. Expert systems, rule-based computer programs that capture the knowledge of human experts in their own fields of expertise, were a perfect solution for this problem. Though many expert systems have several major practical limitations such as a lack of causal knowledge[1] and a knowledge-acquisition bottleneck[2] [Giarratano and Riley, 1998], expert systems have been successful in dealing with real-word problems that conventional programming methodologies have been unable to solve, especially those

[1] Causal knowledge describes the expert systems do not really have an understanding of the underlying cause and effects in a system.
[2] Knowledge-acquisition bottleneck describes the problem of transferring human knowledge into an expert system, which is a time-consuming and labor-intensive task.

dealing with uncertain or incomplete information. According to a rule-based system's definition, Rule-based systems, or often called expert systems, are "software systems (or subsystems) that simulate as closely as possible the output of a highly knowledgeable and experienced human functioning in a problem-solving mode within a specific problem domain" [Lane, 1986]. The three main components of an expert system are the knowledge base (i.e., the expertise in a specific domain), the inference engine (i.e., the controlling mechanism), and the user interface (e.g., explanation facilities). In general, the inference engine applies the rules in the knowledge base on the facts in working memory to construct an agenda. The list of rules that could potentially be fired is stored on the agenda. The execution engine fires the rules from the agenda, thereby changing the contents of the working memory and restarts the cycle.

Thus using predicates is easy to map the expertise and easy to simulate experts decision making. Our approach is to build a rule-based tool that detects good and bad OO design constructions, i.e., constructions corresponding to standard solutions to recurring design problems (design patterns), or constructions that can result in future maintenance and reuse problems (design heuristics, anti patterns). Using this aid, it is possible to identify structures in a system that need to be modified to make it more flexible and reusable, and, by identifying existing design patterns, to facilitate the understanding system as a whole, including that of the badly documented systems.

Figure 1 shows the architecture of our approach. The whole process can be divided into the following steps:

- Expertise acquisition
- Design extraction
- Design facts generation
- Design analysis

The design-extraction process will parse source code and generate an intermediate representation of the source code – SDG (Semantic Directed Graph); then the design facts generation process will traverse SDG graph and produce facts representing

design information; those facts are then loaded into the working memory or saved into a repository to be used later on. These facts are stated in predicates corresponding to the constructions defined in the meta-model for object-oriented software. This definition of this meta-model was based on the UML semantic meta-model [UML, 1997].

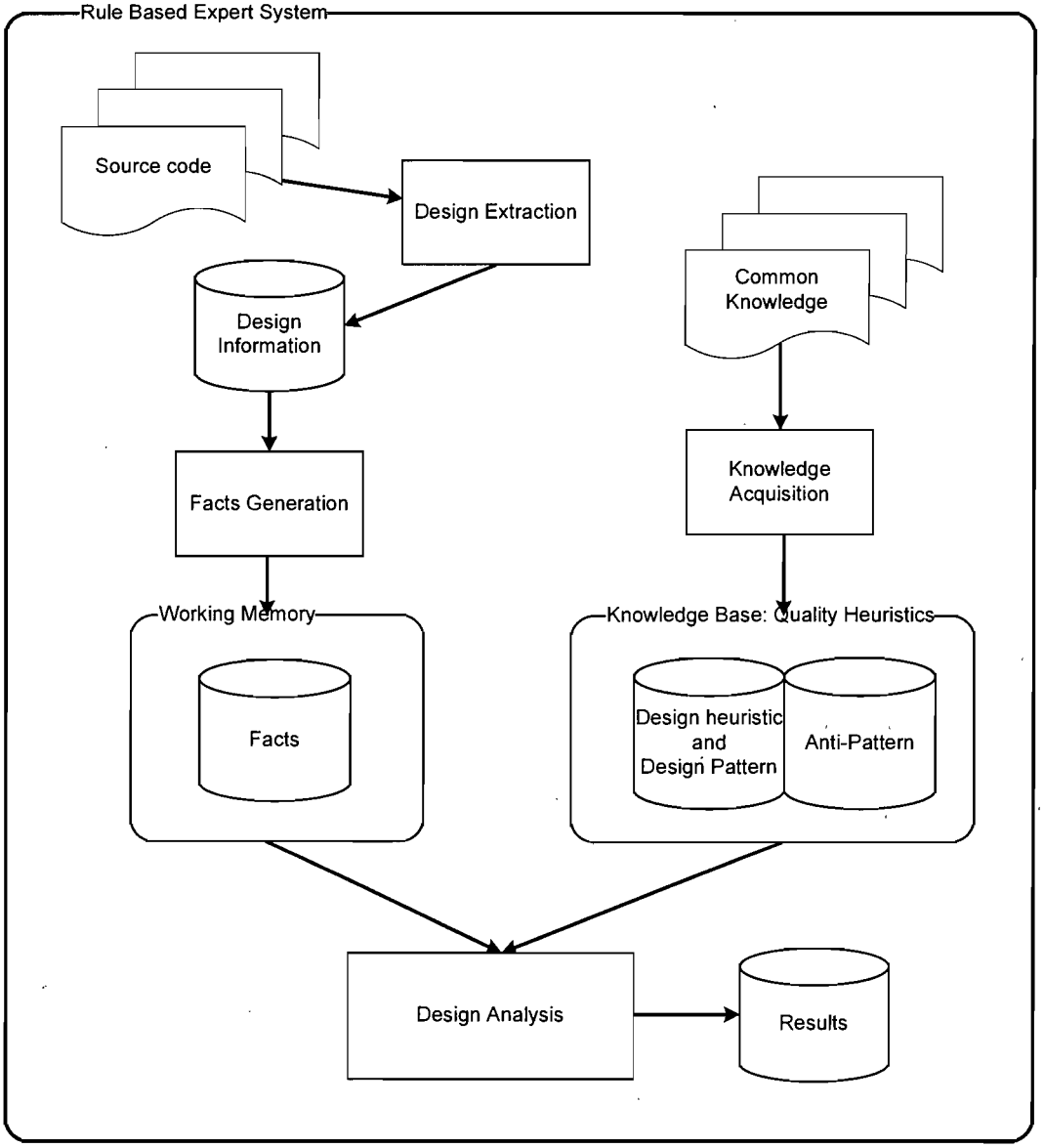


Figure 1 Design Architecture

Quality heuristics compose the knowledge base in our approach, which are confined from design heuristics, design patterns and anti-patterns (we name them as common

knowledge). The common knowledge, which shows what a good object-oriented system should look like, exists in the literature. Originally these guidelines were meant to be followed by a human developer when creating a new design, rather than by an automatic tool detecting violations of design rules in a given design of a legacy system. Quality heuristics are manually examined to see whether they could be used for automatically detecting problems in the CASE tool. Those proven quality heuristics are formalized into production rules and saved into the rule repository.

Finally the design analysis process will apply quality heuristics rules to facts representing design information and will produce a report of violations and conformances found in the target system.

We have implemented a prototype tool according to our approach and have evaluated the prototype tool on several open-source systems. The results show that our approach can provide the following benefits to software engineers:

- Use the approach has been accepted and used naturally by software engineers.
- Define and formalize quality heuristics into the knowledge base.
- Show the location of a problem directly instead of showing a flood of metrics values that require further interpretation.
- Illustrate what kind of problem it belongs to.
- Comprehend the source code.
- Define a rule engine abstraction layer that allows us to develop our rules in a vendor-neutral language.
- Give a promising solution to the problem.

1.4 Structure of the thesis

The thesis consists of 7 chapters as follows.

Chapter 1, **Introduction:** talks about the motivation of our work and the general methodology; depicts architecture of our prototype; introduces main components in the prototype and finally shows the advantages of our approach.

Chapter 2, **State of the Art**: lists related work, discusses similarities and differences between their work and ours.

Chapter 3, **Quality heuristics**: presents the concepts and relationships of design heuristics, design patterns and anti-patterns; quality heuristic are refined from those raw sources.

Chapter 4, **Quality heuristics formalization and identification**: introduces production systems; defines selected UML constructs in production terms; and formalizes and identifies design quality heuristics rules in the production system.

Chapter 5, **Implementation**: describes the design architecture of our prototype tool and its design details; illustrates the functionalities and GUI of our prototype tool; and discusses key implementation issues.

Chapter 6, **Evaluation**: presents an evaluation procedure and shows evaluation results on selected open-source projects that have applied our prototype tool. Finally, this chapter presents a case study of the different versions of an open-source project.

Chapter 7, **Conclusion and Future work**: gives additional ideas outside the scope of the present thesis for future work with CASE tool and provides a conclusion for the work presented.

2 State of the Art

This chapter discusses related work; at first it provides a literature survey of works surrounding reverse engineering and research into quality heuristics formalization and identification, and presents a more detailed comparison of these with the work provided in this thesis.

2.1 Literature survey

Object-oriented methodology has dominated software development area for several decades. Along with this trend, a considerable number of methods have been introduced to help software engineers design and develop OO products such as [Rumbaugh et al. 1991] and [Booch, 1994] etc. Along with these methods, several tools became available (e.g., [Rational Rose, 1997], [Together, 2006]). The emphasis of these methods and tools has been on how to develop semantically correct OO models regarding the constructions available in modeling languages such as UML [UML, 1997], for example.

However, a correct model does not necessarily mean that it is flexible and reusable. The expertise of OO gurus and related research have bridged the gap between correct models and quality design, and have been captured in the literature on heuristics [Riel, 1996], [Martin, 2000], [Lieberherr, 1996] and design patterns [Gamma et al., 1995], etc. The heuristics cover important topics ranging from classes and objects with emphasis on their relationships to physical object-oriented design[3]. Heuristics can highlight a problem in one facet of a design while design patterns can provide the solution.

Although design heuristics and design patterns were originally supposed to be used by human developers, works introduced in [Brown, 1996], [Grotehen and Dittrich, 1997], [Bar and Ciupke, 1998], [Prechelt and Kramer, 1998], [Correa et al., 2000]

[3] Physical object-oriented design [Riel, 1996] involves the techniques used to map logical design (abstract constructs such as classes and relationships) onto given software and hardware platforms.

and [Wenzel, 2006] have demonstrated how to use them in an automatic manner, that is, how to integrate the expert's knowledge and thinking patterns into CASE tools. There are two main approaches to doing heuristics or pattern detection; one is by using graph matching, and the other way is by using production system.

2.2 Related work

Several works related to the reengineering of legacy object-oriented systems and the detection of design heuristics and design patterns have appeared in the last ten years. In this section, we compare them with our work in more detail.

2.2.1 MeTHOOD

MeTHOOD [Grotehen and Dittrich, 1997] is a framework that enables a design process that allows designers to review and improve object-oriented designs on the meta-model level. It consists of a design-knowledge base (containing definitions of measures, heuristics, and transformation rules) that works on a specification database containing conceptual design schemas. It applies the heuristics rule on conceptual design schemas to discover design flaws; it then uses transformation rules to create a proposal for an alternative design. Measures are used to deal with conflicts when two or more heuristic rules (as well as transformation rules) are possible in a given context at the same time.

The overall design of MeTHOOD is that of an object-oriented database system. It uses an object-query language to express a design knowledge base. The targeted design models are presented as records in the database. Moreover, it uses measures to overcome heuristic rules conflicts manually. MeTHOOD's design meta-model has to be entered by using a special editor and cannot be discovered from source code automatically. So far, only a few rules are given formally. In addition, MeTHOOD is more general in the sense that is provides concepts for transforming designs and for resolving problems. Comparing MeTHOOD with our approach, we use a production system to represent the design knowledge base and the targeted design models; a production system (expert system) is originally designed to capture the human

knowledge and to simulate human thinking ability; solving rule conflict is an integral part of a production system. Another difference is that our approach can read a model from source code and comes with a set of formally defined rules that can be applied for problematic and reusable structures' detection.

2.2.2 GOOSE

GOOSE [Bar and Ciupke, 1998] is a reengineering tool set that helps the user to detect design problems in a legacy source code. It formalizes design heuristic rules, extracts design information from legacy source code and searches for violations of these rules automatically. It mainly provides a set of design heuristic rules that can be used in CASE tools automatically as well as those that not be. It uses the term "testability" to judge how precisely these design heuristics can be used as an automated search for violations.

The differences between GOOSE and our prototype are that the goal of GOOSE is to detect design problems; the design heuristic rules are mainly retrieved from [Riel, 1996]; its implementation uses Prolog. Our prototype not only detects design problems, but it also detects well-known good design structures to help end-users comprehend the designs. Both detected structures will ultimately be used as inputs to evaluate overall software quality. Quality heuristics are gathered from design heuristics, design patterns and anti-patterns, which are much broader than GOOSE. Finally, we use Jess [Jess, 2006] as our production system; it is reputed to be more efficient than the Prolog system.

2.2.3 KT

The first attempt to automatically detect design patterns was performed by Brown [Brown, 1996]. In this work, Smalltalk code was reverse-engineered in order to detect four well-known patterns from the catalogue by [Gamma et al., 1995]. The algorithm was based on information retrieved from class hierarchies, association relationships

and aggregation relationships, as well as the messages exchanged between classes of the system.

The KT tool focused on searching for Composite, Decorator, Template Method and the Chain of Responsibility. It noted that Strategy, State and Command would potentially be detectable, but that they would be ambiguous; so much so, that it would potentially be easy to obtain a false positive. Moreover, the KT tool is restricted to detecting design patterns in Smalltalk, since it regards only flows in VisualWorks for Smalltalk.

Our prototype is capable of detecting design patterns as well as design problems according to design heuristics and anti-patterns rules. Our prototype can detect more design patterns than the KT tool does. Also our prototype is a rule-based system that uses UML meta-model-based predicates to uniformly express software design information as well as the knowledge of design patterns, anti-patterns and design heuristics. Our prototype shares the same shortcoming as does the KT tool, which is a false positive result for the detection of Strategy and State design patterns.

2.2.4 OMT

OMT [Florijn, 1997] supports working with design patterns when developing or maintaining object-oriented programs. This tool provides three integrated views of a program: the source code view, design view and occurrences of design patterns in the program. The tool assists developers using patterns in three ways:

- Generating program elements (e.g., classes, hierarchies) for a new instance of a pattern, taken from an extensible collection of "template" patterns
- Integrating pattern occurrences with the rest of the program by binding program elements to a role in a pattern (e.g., indicating that an existing class plays a particular role in a pattern instance)
- Checking whether occurrences of patterns still meet the invariants governing the patterns and repairing the program in case of problems

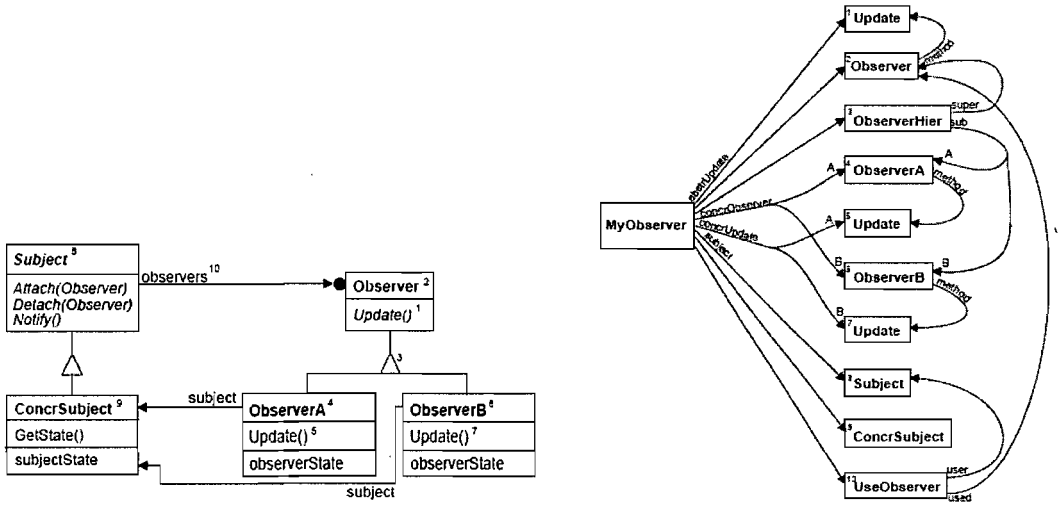


Figure 2 Observer instance and corresponding fragment structure

The tool proposed a mechanism for representing patterns as a set of fragments, called "The Fragment Model." This breaks down the pattern in terms of what it must provide: relationships with other classes (including inheritance), methods that must be present, and how they are connected with one another. Each one of these requirements is a fragment, and a collection of fragments together under one 'root' fragment defines a pattern. The aim of the fragment model is to provide a definition of the design patterns that can be used in a practical tool to allow the developer to instantiate patterns from scratch or from existing code. An example of a fragment is shown in Figure 2.

Our prototype is a rule-based system, it uses UML meta-model-based predicates to express software design information instead of "The Fragment Model" in OMT. It will detect design problems as well as well-known good design structures. OMT does not offer support for the automatic search of design patterns; it can generate source code when a user selects a pattern template and does refactorings, a suite of transformations that restructure and extend the program on a design level according to designated patterns.

2.2.5 SAD

SAD [Moha and Gueheneuc, 2006], proposes a language (Rule Card) and a framework (SAD) to express design defects synthetically and to generate detection algorithms automatically.

Rule card specifies design defects synthetically. Rule cards are expressed using a BNF grammar. A rule card is identified by the keyword RULE_CARD, followed by a name and a set of rules specifying this specific design defect as a set of code smells. A rule describes a code smell as a list of properties (metrics, structural, or semantics).

The automated generation of detection algorithms relies on the SAD framework. SAD provides the building blocks common to all detection algorithms. It includes the PADL and SADDL meta-models, which represent object-oriented programs and provide constituents related to design defects to describe models of rule cards respectively.

Finally, SAD includes algorithms to visit models of rule cards and to generate detection algorithms from these models.

Our prototype is a rule-based system; it detects design problems as well as well-known good design structures. Their meta-model and our meta-model are quite similar. The rule card is interesting for us; we can use it in our knowledge-acquisition process to automatically generate quality heuristic rules from common knowledge.

3 Quality Heuristics

Designing object-oriented software is difficult; designing reusable object-oriented software is even more difficult. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements.

Experienced object-oriented designers do make good designs. Experienced designers evidently know something that inexperienced designers do not. What is it that they know? One thing expert designers are aware of is a list of guidelines for good or bad designs based on their years of experience. These guidelines help them develop good design and improve design quality. These guidelines are called design heuristics. Experts also know not to solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, recurring patterns of classes and communicating objects are found in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. Design pattern is the name of these reusable good solutions or recurring patterns. The knowledge of experts not only includes the patterns related to good design, but also the patterns related to design problems, such as reusable resistant structures, good patterns applied in wrong contexts, etc. Anti-pattern is the name of these bad solutions. Quality heuristics are developed from design heuristics, design patterns and anti-patterns, which are the core of our prototype. We discuss design heuristics first and, thereafter, we talk about design patterns and anti-patterns. At the end of this chapter, we explain relations among three of them.

3.1 Design Heuristics

Experienced OO developers can look at source code or UML diagrams directly and identify design heuristics that influence the system's design. They then exert judgement regarding the balance of these design heuristics to form an opinion about the quality of the component or system in question. This process is largely influenced by the opinions of individual developers and involves a number of aesthetic components.

Many publications have attempted to capture the expertise of skilled OO developers, such as [Meyer, 1988], [Martin, 1996a], [Martin, 1996b], [Martin, 1996c], [Riel, 1996], and [Lieberherr, 1996], etc. As explained in [Meyer, 1988] and [Martin, 1996a]. For instance, "Classes should be open for extension, but closed for modification." The goal of this design heuristic is to allow classes to be easily extended to incorporate new behaviour without modifying existing code. In other words, designs should be resilient to change and flexible enough to take on new functionality to meet changing requirements. In addition, Riel, for example, documented 61 golden rules of OOP [Riel, 1996]. Every rule links to a potential problem in the design where the rule was violated. He describes them in the following way: "not hard and fast rules that must be followed under penalty of heresy. Instead, they should be thought of as a series of warning bells that will ring when violated. The warning should be examined, and if warranted, a change should be enacted to remove the violation of the heuristic. It is perfectly valid to state that the heuristic does not apply in a given example for one reason or another." He classified all the heuristics into 8 categories: Classes and Objects, Topologies of Action-Oriented Versus Object-Oriented Applications, Relationships Between Classes and Objects, Inheritance Relationship, Multiple Inheritance, Association Relationship, Class-Specific Data and Behaviour and Physical Object-Oriented Design.

Another example is K. Beck and M. Fowler's collection of code smells [Fowler, 1999]. Code smells are used to help software developers identify problematic code

and to decide when this code needs to be improved by refactoring. The authors' choice of the term "smells" emphasises the vague and subjective nature of heuristics.

There are several possible types of relationships between heuristics. The most important two are implication and contradiction [Bar and Ciupke, 1998]. The term "implication" is used to mean that the conformance to one heuristic indicates the conformance to another. For example, Riel's Heuristic RH2.1 "All data should be hidden within its class" implies the Information Hiding Principle. This principle suggests that the details of an object that are most likely to change, or "do not contribute to its essential characteristics" [Booch, 1994], should be hidden.

Many contradicting heuristics are derived from differing opinions about good OO design. For example RH5.7, "All base classes should be abstract" discourages concrete base classes; however satisfying this heuristic could result in a Lazy Class Smell. Other contradictions result from conflicting forces of design. The simplest example is RH5.4 and RH5.5, which state, "in theory, inheritance hierarchies should be deep – the deeper the better" and, "in practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this number is 6", respectively.

When faced with contradicting heuristics, the developer should examine the design further to determine whether or not both of them are applicable in their particular situation, and if they are, decide which one "plays the more important role" [Riel, 1996].

3.1.1 Human Factors

Heuristics are expressed in natural language, and, thus, the process of evaluating OO designs with respect to these heuristics can be very subjective. Human factors, such as experience, role, and knowledge of the design in question, all contribute to an individual's interpretation. As Beck and Fowler put it: "in our experience, no set of metrics rivals informed human intuition" [Fowler, 1999]. From this, it is apparent that

automated heuristics cannot supplant the judgement processes of experienced developers, but instead, should be used to facilitate developers' work (novice or experienced).

3.1.2 Relation to Design Metrics

"A heuristic is not a metric" [Gibbon and Higgins, 1996]. Heuristics are rules and guidelines derived from the practical experience of skilled developers. They are expressed using natural language and conventionally have very vague, subjective definitions. Metrics, on the other hand, are very formal and precisely defined measures of software. They are typically, but not always, derived from sound conceptual and theoretical information.

It is common for metric results to be in the form of data values that can be displayed using appropriate measurement scales. Examples of such scales include: nominal, ordinal, interval, and ratio. These results can be effectively used in identifying problem areas in code; however, once a problem has been detected, metrics fail to provide developers with the guidance required to resolve the problem.

3.2 Design Pattern

In order to avoid redesigning, or at least to minimize it, experienced object-oriented designers explain you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Meanwhile, new designers are overwhelmed by the options of design methodologies available; those designers who come from a procedure-oriented background tend to fall back on non-object-oriented techniques they have used before. It takes a long time for novices to learn what good object-oriented design is all about.

We all know the value of design experience. How many times has one had design déjà-vu—that feeling that one has solved a problem before but not knowing exactly where or how? If one could remember the details of the previous problem and how it was solved, then the experience could be reused. Design experience helps designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them. An analogy helps illustrate the point. Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns like the pattern of "represent states with objects" and "decorate objects so you can easily add/remove features." Once you know the design pattern, many design decisions follow automatically.

Design patterns have been one of the most significant developments in software engineering in the past decade. The aim of this field is to identify and catalogue the knowledge and expertise that has been built up over many years of software engineering. Design patterns can be identified in all parts of the development process: architecture, analysis, design, coding, reengineering, as well as in specific application areas such as real-time programming or in user-interface construction. Design patterns are in no way invented; they are discovered or "mined" from existing systems. The motivation is to uncover proven designs that experts have already used and reused and to distil from these the essence of the solution with domain-specific detail removed. The resulting nugget of design wisdom can then be documented and made generally available. This pattern can be assimilated by other designers and applied in other domains.

The notion of a design pattern in software was borrowed from the work of the architect Christopher Alexander, who described the process of architecting living space (be it the corner of a room or an entire city) in terms of patterns. He defined the notion of a pattern in the following way:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

Varying definitions of the term pattern abound, but this "three-part" version suits our current purposes. Gabriel puts the Alexandrian definition into a software context in this way:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves [Gabriel, 1995].

In contrast to the design heuristic, the available design patterns today in the literature are described in explicit and organized form. In a general way, all descriptions of a design pattern must contain the following information:

- **Pattern Name and Classification:** Gives the pattern a name that becomes part of the designer's vocabulary and conveys the essence of the pattern succinctly.
- **Intention:** Explains what the design pattern does and what particular design problem it addresses to.
- **Also Known As:** Gives other names used for this pattern.
- **Motivation:** Illustrates the design problem and how the classes and object structures solve the problem.
- **Applicability:** Talks the design pattern can be applied in, in which situations, and how to recognize these situations.
- **Structure:** Presents a graphical representation of the participating classes in the pattern, using a graphic notation similar to OMT. It also describes the sequences of requests and collaborations among objects by means of interaction diagrams.
- **Participants:** Describes the classes and objects participating and their responsibilities.
- **Collaborations:** Describes how the participants collaborate to carry out their responsibilities.

- **Consequences:** Describes how the design pattern supports its objective and what aspects of the systems structure it lets vary independently.
- **Implementation:** Explains what techniques should be taken into consideration during the implementation period.
- **Sample Code:** Illustrates how to implement the pattern in C++ or Smalltalk.
- **Known Uses:** Presents a short enumeration of the existing designs where the pattern has been used.
- **Related Patterns:** Shows what design patterns are closely related to this one, what their differences are, and which other patterns this pattern could be used with.

As the number of discovered design patterns grows, it make sense to partition them so that we can organize them, narrow our searches to a subset of all Design patterns, and make comparisons within a group of patterns. The most well-known scheme, which was used by the first pattern catalogue, partitions the patterns into three distinct categories based on their purposes [Gamma et al., 1995]:

- Creational patterns involve object instantiation and provide a way to decouple a client from the objects it needs to instantiate, such as Singleton, Abstract Factory, Factory method, ...
- Behavioural patterns are concerned with how classes and objects interact and distribute responsibility, such as Visitor, Observer, State, Iterator, etc.
- Structural patterns allow for composing classes or objects into larger structures, for example, Adapter, Composite, Decorator and Façade are in this category.

Design patterns in [Gamma et al., 1995] are more abstract and focus on problems in object-oriented software in general. For example, The Observer pattern defines a one-to-many dependency between objects, so that when one object changes state, all of its dependents are notified and updated automatically. Its structure is shown in Figure 3. The clients use Subject interface to register observers and also to remove observers; observer interface has one method, update(), that gets called up when the subject's state changes; a concrete subject always implements the Subject interface. Concrete

observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive an update. It provides an object design where subjects and observers are loosely coupled, that is, they can interact, but have very little knowledge of each other. It is the building block of the well-known MVC (Model, View and Controller) pattern.

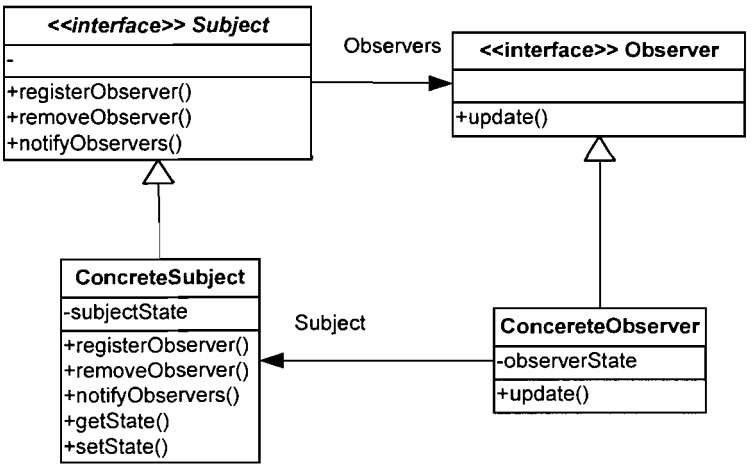


Figure 3 Observer Design Pattern

The present thesis focuses on the automatic detecting of occurrences of design patterns as well as of the design problems. We choose to work with patterns at the design level for two reasons:

- It is a richer set than the program-language specific patterns found at the coding level.
- They are more concrete than those found at the analysis level, so that detecting the occurrences of these design-level patterns automatically from source code is realistic.

The notions of formalization and automation are not generally welcomed in the patterns community. James Coplien expressed this distaste clearly in [Coplien, 1996]: "Patterns are not designed to be executed or analyzed by computers, as one might imagine to be true for rules: patterns are to be executed by architects with insight, taste, experience, and a sense of aesthetics." We have to agree his claim in terms of the first two parts of Gabriel's definition. How it is decided that a context is appropriate for the application of a pattern and that assess the forces acting in this context will be resolved by the pattern is a matter of "insight, taste, experience, and a

sense of aesthetics." However, the third part of the pattern definition, that of applying the software configuration that resolves the forces, is clearly a potential candidate for automation.

In Chapter 4, we will present a methodology for the development of automated design pattern transformations where the designer defines the context to which the pattern is to be applied and the actual application of the software structure is automated. Other work in the area of automated pattern application is considered in that chapter as well. Thus in this chapter we focus on other uses of formalization and automation in the context of design patterns.

Since software systems are progressively becoming larger and more complex, the task of understanding while developing and especially while maintaining software is becoming more and more difficult. Therefore, the use of patterns has become a helpful methodology to develop software in a more structured and understandable way. A key reason for using a pattern is that it helps describe the system, as well as implement it. Thus, when a pattern is used (and documented) in a code base, it aids other developers looking to extend the system.

3.3 Anti-Pattern

A design pattern gives a general solution to a recurring problem in a particular context; the universe just would not be complete if we only had positive parts and no negative part. The complementary part is anti-pattern, which tells you how to go from a problem to a BAD solution. Brown et al. give the expression "anti-pattern" the formal definition of "a literary form that describes a commonly occurring solution a problem that generates decidedly negative consequences. An anti-pattern describes a general form, the primary causes which led to the general form; symptoms of the general form; and a refactored solution describing how to change the Anti-pattern into a healthier situation" [Brown et al., 1998].

The concept of anti-pattern was first formally introduced from "Antipattern Session Notes" presented in the Object World West conference in 1996 by Michael Akroyd.

The discussion of the usefulness of anti-patterns began almost in parallel with the introduction of patterns. Similar work on providing software guidance based on dysfunctional behaviour and refactoring a solution has been documented by B. Webster [Webster 95], and J. Coplien [Coplien, 1996] and [Brown et al., 1998].

Like design patterns, there are many types of anti-patterns:

- Development anti-patterns that comprise technical problems and solutions that are encountered by programmers.
- Architectural anti-patterns that identify and resolve common problems in how systems are structured.
- Managerial anti-patterns that address common problems in software processes and development organizations.

As we are concentrating on automated design knowledge identification, we will focus only on developing anti-patterns. The number of catalogued anti-patterns is still small, if compared with the amount of available design patterns in the literature. [Koenig, 1995] claims that it is more difficult to classify anti-patterns than their counterparts because people are more likely to expose their successes than their failures.

Analog to design patterns, anti-patterns are also described in a standardized structure. This structure is a little different from the structure of the design patterns due to the nature of the anti-pattern. Instead of a problem and a solution, an anti-pattern possesses two solutions: the first one generates negative consequences, whereas the second is a migration or refactoring of the first one, aiming to eliminate or at least to reduce its negative impacts.

[Brown et al., 1998] considers a structure for the description of anti-patterns, composed of the following sections:

- Name: analogous to the form of the design patterns. The philosophy of giving names to the anti-patterns aims at creating a common terminology that

facilitates the communication among the members of development teams. An anti-pattern can also be known by other names (synonymous).

- General form: this section identifies the main characteristics of the anti-pattern, being able to include diagrams. The refactored solution resolves the problem described in this section.
- Symptoms and consequences: this section lists symptoms and resultant consequences of this anti-pattern. The symptoms supply indications of where the anti-pattern can be detected. The consequences mention the problems if this bad solution is applied to a real problem.
- Typical causes: they identify the main reasons that lead to the appearance of this type of solution.
- Known exceptions: anti-pattern behaviour and processes are not always wrong; often there are specific occasions when this is the case.
- Refactored solution: this section explains a refactored solution that resolves the forces in the anti-pattern identified in the "General form" section. The new solution is structured in terms of solution steps.
- Variations: this section lists the possible major variations of this anti-pattern. If there are alternative solutions, they are described here as well.
- Related solutions: any closely-related anti-patterns are listed and the differences are explained.

By documenting anti-patterns, we help others to recognize bad solutions before they implement them.

Figure 4 provides an example of anti-patterns named Blob, which is listed in [Brown et al., 1998]. It is presented in the form described above as shown in Figure 4.

Name: The Blob

Also Known As: The God Class

General Form: The key problem is that the majority of the responsibilities are allocated to a single class. One class monopolizes the processing; other classes primarily encapsulate data. The Blob is a procedural design even though it may be represented using object notations and implemented in object-oriented languages. That is why this anti-pattern frequently is found in designs or implementations made by former C programmers. The Blob is also frequently a result of iterative development where proof-of-concept code evolves over time into a prototype, and, eventually, a

production system. This is often caused by GUI-centric programming languages, such as Visual Basic. This kind of language is often used for rapid prototyping. The Blob is often accompanied by unnecessary code, making it hard to differentiate between the useful functionality of the Blob Class and no-longer-used code.

Symptoms and Consequences:

- Single class with a large number of attributes, operations, or both. A class with 60 or more attributes and operations usually indicates the presence of The Blob.
- A single controller class with associated simple, data-object classes.
- The Blob Class is typically too complex for reuse and testing.

Typical Causes:

- Lack of an object-oriented architecture. The designers may not have an adequate understanding of object-oriented principles or the team may lack appropriate abstraction skills.
- Lack of any architecture. The absence of definition of the system components, their interactions, and the specific use of the selected programming languages. The programming languages are not intended for use in this kind of task.
- Too limited intervention. In iterative projects, developers tend to add little pieces of functionality to existing working classes, rather than add new classes, or revise the class hierarchy for more effective allocation of responsibilities.

Known Exceptions: The Blob AntiPattern is acceptable when wrapping a legacy system. A final layer of code makes the legacy system more accessible.

Refactored Solution: A refactored solution means that we must find a way to rebuild our program. We must move behavior away from the Blob class in a way that makes the Blob less complex and it is supporting classes more capable. The method for refactoring responsibilities is described below.

1. Identify or categorize related attributes and operations according to contracts. For example: everything in The Blob Class that deals with sorting (Sort_Catalog, Search_Catalog) is grouped together. So is everything that deals with printing, etc.

2. Now we look for "natural homes" for these contract-based collections and migrate them there. In this example, we can move everything that involves sorting operations on a catalog to the Catalog Class. We do the same thing with the other groups of operations that can be migrated.

3. The third step is to remove all "far-coupled", or redundant, indirect associations.

4. Next, where appropriate, we migrate associates to derived classes to a common base class.

5. Finally, we remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments.

Variations: Sometimes too much hard work is done to refactor The Blob Class. There is another way to do it, but it provides an "80%" solution. Instead of a bottom-up refactoring of the entire class hierarchy, it may be possible to reduce the Blob class from a controller to a coordinator.

Figure 4 Blob Anti-pattern

Knowing the Blob anti-pattern, you can get the following benefits:

- An anti-pattern tells you why a bad solution is attractive: no one would choose a bad solution if there was not something attracting people. One of the biggest jobs of the anti-pattern is to let you be aware of the attractive aspect of the solution.

- An anti-pattern tells you why, in the long term, that solution is bad: in order to understand why it is an anti-pattern, you must understand how it is going to have negative effects down the road. The anti-pattern describes where you will get into trouble by using the solution.
- An anti-pattern suggests other applicable patterns that may provide good solutions: to be truly helpful an anti-pattern needs to point in the right direction; it should suggest other possibilities that may lead to good solutions.

Anti-patterns can be seen as an extension of patterns, since they represent traps and pitfalls concerning the patterns. They can also be seen as a learning tool that helps people to learn from other people's mistakes and to recognize early on where one starts to go wrong.

3.4 Relation between Design Heuristic, Pattern and Anti-Pattern

As Riel said, "Design heuristics can highlight a problem in one facet of a design while design patterns can provide the solution" [Riel, 1996]. Anti-patterns are complementary to design patterns; moreover, design heuristics and design patterns are the sources of new anti-patterns.

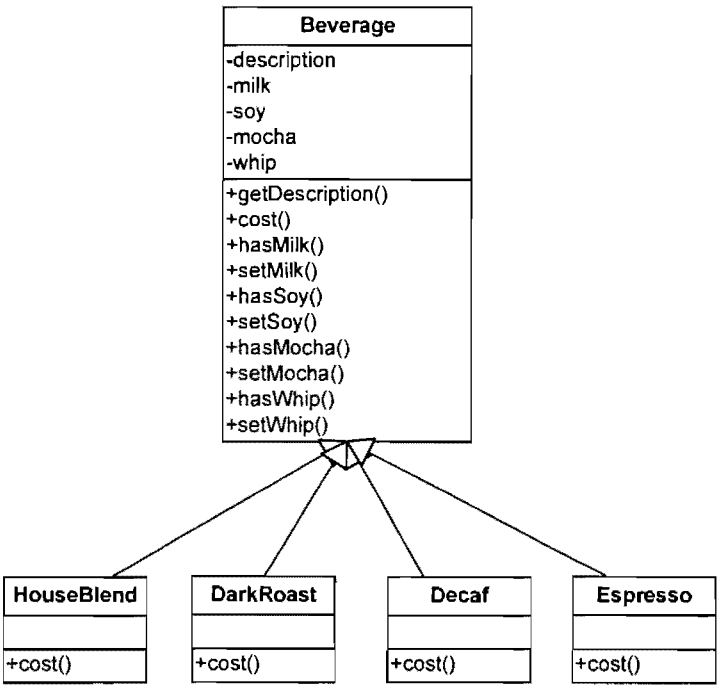


Figure 5 Beverage Class Diagram

We use an example, a beverage ordering system, to show how this relationship works. In addition to ordering different types of coffee such as HouseBlend, DarkRoast, Decaf, and Espresso, etc., consumers can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. As depicted in Figure 5, the Beverage class diagram has a beverage base class with instance variables to represent whether or not each beverage has milk, soy milk, mocha and whip. Different kinds of beverages are created by inheriting from base beverage class. The cost() function in Beverage can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will override cost(), but they will also invoke the super version so that they can calculate the total cost of the basic beverage plus the costs of the added condiments. Using this design, the system seems to produce different coffees with different topping without any problem.

Before we examine the design further, we are going to introduce two fundamental design heuristics:

- Open-Closed Principle: Classes should be open for extension, but closed for modification.
- Prefer Composition to Inheritance: favour object composition over class inheritance on reuse.

"Open" means that one should feel free to extend the classes with any new desired behaviour. If needs or requirements change (and they will), just go ahead and make your own extensions. "Close" tells that we spent a lot of time getting this code correct and bug free, so we can not let you alter the existing code. It must remain closed to modification. The goal is to allow classes to be easily extended to incorporate new behaviour without modifying existing code. Designs that comply with this heuristic are resilient to change and flexible enough to take on new functionality to meet changing requirements. Even "open-close" heuristic sounds very contradictory. As it turns out, though, many of the design patterns give us time-tested designs that protect source code from being modified by supplying a means of extension. Thinking about the Observer pattern whose class diagram is shown in Figure 3, we can extend the

Subject by adding new Observers at any time, without adding code to the Subject. There are quite a few more ways of extending behaviour with other OO design techniques.

The rationality of the second heuristic is that when inheriting behaviour by subclassing, that behaviour is set statically at compile time. In addition, all subclasses must inherit the same behaviour. If, however, we extend an object's behaviour through composition, then we can change the object's behaviour dynamically at runtime. By dynamically composing objects, we can add new functionality by writing new code rather than by altering existing code. In other word, because we are not changing existing code, the chances of introducing bugs or causing unintended side effects in pre-existing code are reduced.

Bearing the aforementioned two heuristics in mind, if we think about how the design might need to change in the future, we will find some potential problems deriving from the design of the beverage ordering system:

1. Price change for condiments will force us to alter the base class code.
2. New condiments will force us to add new methods and alter the cost method in the base class.
3. We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().
4. What if a customer wants a double mocha?

We have seen that representing our beverage plus condiment pricing scheme with inheritance has not worked out very well – we get rigid designs because it violates the "open-close" heuristic, or we add functionality to the base class that is not appropriate for some of the subclasses. That is why we "prefer composition to inheritance".

To follow the Open-Closed principle and the Composition-over-Inheritance" principle, we will apply the Decorator design pattern in the design of the beverage ordering system. We will start with a beverage and "decorate" it with the condiments

at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we will: Take a DarkRoast object; Decorate it with a Mocha object; Decorate it with a Whip object; Call the cost() method and rely on delegation to add on the condiment costs. The new design is shown in Figure 6.

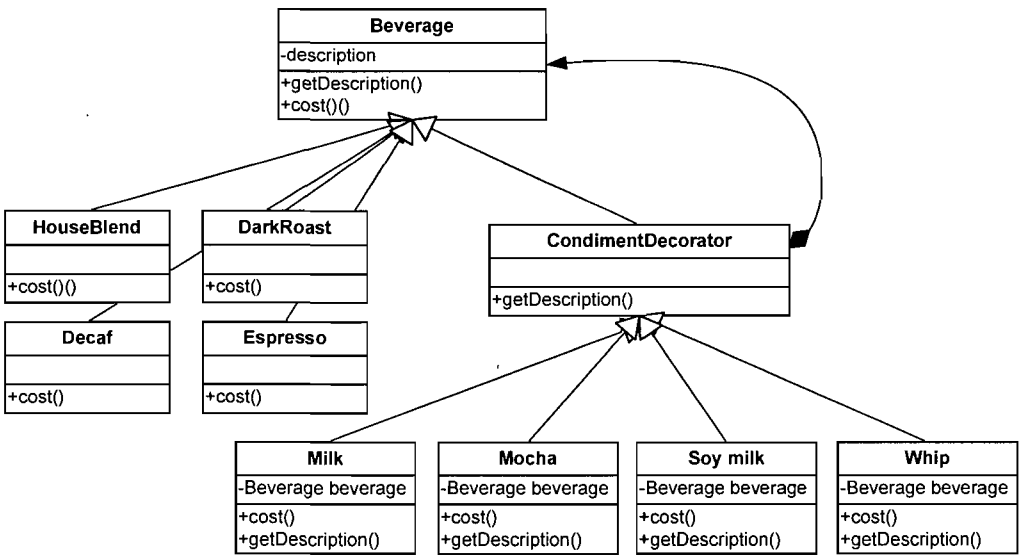


Figure 6 Improved Beverage Class Diagram

The Decorator pattern dynamically attaches additional responsibilities to an object. Decorators provide a flexible alternative to subclassing for extending functionality. We have got the following benefits from applying the decorator design pattern:

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- To do the rest of the job, the decorator adds its own behaviour either before and/or after delegating to the object it decorates.

An anti-pattern describes a solution to a recurrent problem that generates negative consequences for to a project that, normally, violates one or more design heuristics. An anti-pattern can be the result of either not knowing a better solution, or using a design pattern (theoretically, a good solution) in the wrong context. One possible way

to search for problematic solutions in OO software design is by looking at design pattern catalogues. These catalogues not only describe good solutions applicable in a particular context, but also informally discuss bad solutions that could have been used instead. Those bad solutions can be formalized and catalogued, composing a database of OO design problems.

The Singleton pattern [Gamma et al., 1995] can be taken as an example. It ensures that the designated class has only one instance at runtime and provides a global access point for the instance. It can be used when multiple instances of a class are prohibited in a system, or it can be used preclude the unnecessary object instantiations of a class. The unfamiliarity of this design pattern can be implemented with a bad solution. For example, a global variable of the system or a static attribute with public visibility is used instead of using singleton pattern. Because all the clients have direct access to this global instance, maintenance problems can occur, especially if the system is large.

Design heuristics can be another source for anti-patterns. Known forms of the breaking of a design heuristic can be interpreted as anti-pattern. As Riel said, "the heuristics are not written as hard and fast rules; they are meant to serve as warning mechanisms which allow the flexibility of ignoring the heuristic as necessary" [Riel, 1996]. A breaking of a design heuristic does not correspond necessarily to a design problem. That is perfectly compatible with the philosophy of anti-patterns. Its description allows identifying situations where this breaking would be acceptable.

As an example, the base class depends on its derived classes is a resultant anti-pattern of the breaking of the design heuristic of that "Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes" (RH 5.2 [Riel, 1996]). If base classes have knowledge of their derived classes, then it is implied that if a new derived class is added to a base class, the code of the base class will need modification. This is an undesirable dependency between the abstractions captured in the base and in the derived classes.

In such a way, we conclude that, although design heuristic, design pattern and anti-pattern appear separately in the literature, the concepts of design heuristic, design pattern, and anti-pattern are closely related. They complement each other.

4 Quality Heuristics Formalization and Identification Using Production System

We have introduced design heuristics, design patterns and anti-patterns, and quality heuristics that are derived from them. In this chapter, we propose a solution for identifying these quality heuristics automatically. In particular, we will focus on those quality heuristics introduced in Chapter 3.

First, we introduce production systems, algorithms, and applications. Next, we define the UML constructs and quality heuristic rules using production system language. Finally, we demonstrate the application of production system techniques on a design example.

4.1 Production Systems

A production system is a reasoning system that uses forward-chaining derivation techniques. It uses rules, called production rules or productions in short, to represent its general knowledge, and keeps an active memory of facts (or assertions) known as the working memory (WM).

A production rule is usually written in the following form:

```
IF conditions Then actions
```

The antecedent conditions, also known as patterns, are tests that are applied against the current state of the WM. They are partial descriptions of working memory elements. If the conditions are satisfied by some elements, the consequent actions are fired to modify the WM. The basic operation of a production system is a cyclic application in order of the following three steps, until no more rules can be applied:

1. Recognize: identify applicable rules whose antecedent conditions are satisfied by the current WM;
2. Resolve conflict: among all applicable rules (known as the conflict set), choose one to execute;

3. Act: modify the WM by applying the action given in the consequent of the executed rule.

More efficient algorithms that perform the basic operations of production systems include RETE [Jess, 2006]. RETE matches applicable rules by setting up a network that allows new working memory elements to pass incrementally for testing.

4.1.1 Working Memory

Working memory consists of a set of working memory elements (WMEs). Each has the following form:

$$(\texttt{type attribute}_1\texttt{:value}_1 \ \ldots \ \texttt{attribute}_n\texttt{:value}_n)$$

where type, attribute_i, and value_i are all atoms. Each WME can be interpreted as an existential sentence:

$$\exists x \cdot [type(x) \wedge attribute_1(x) = value_1 \wedge \cdots \wedge attribute_n(x) = value_n]$$

4.1.2 Production Rules

The antecedent of a production rule is a set of conditions. Each condition can be either positive or negative. A negative condition is of the form $\neg\,\texttt{cond}$. The body of a positive condition is composed of the following tuple:

$$(\texttt{type attribute}_1\texttt{:specification}_1\ldots\texttt{attribute}_n\texttt{:specificaiton}_n)$$

where each specification can be one of the following:

- an atom, including a string within " ", a word, a numeral;
- a variable, denoted in italic letters;
- an evaluation expression, within [], including arithmetic, string manipulation;
- a test, within {}, including $<$, $>$, $=$, $\neq$.
- the conjunction ($\wedge$), disjunction ($\vee$), or negation ($\neg$) of a specification.

A rule is applicable if all of the variables can be evaluated using the WMEs in the current WM such that the conditions are met. A positive condition is satisfied if there is a matching WME in the WM; a negative condition is satisfied if there is no matching WME in the WM. Production rules are stored in the production memory of the system.

4.1.3 Conflict Resolution

To resolve conflicts among all applicable rules, there are two general approaches. In a data-directed context, all applicable rules can be fired to obtain all consequences. In a goal-directed context, only one rule is selected to fire, allowing a single goal to be pursued.

There are a number of standard ways for selecting a rule:

1. Randomness: select a rule at random.
2. Order: choose the first rule in order of presentation. (This can be modified to use a priority scheme for the selection.)
3. Specificity: select a rule whose conditions are most specific. Rule A is said to be more specific than rule B if the conditions of B are a subset of those of A.
4. Recency: choose a rule based on how recently it has been used.
5. Hierarchical: a combination of a few of the above selection schemes in hierarchical levels because after applying a single scheme, more than one rule may still be applicable.

4.1.4 Applications and Advantages

Production systems are commonly used in practice to solve complex problems. Well-known applications include MYCIN and XCON. MYCIN was developed at Stanford with approximately 500 production rules for recognizing about 100 infections in assisting physicians in the diagnosis of such bacterial infections. XCON was developed by researchers at Carnegie-Mellon for Digital Equipment Corporation and is used in configuring computers.

Among other major advantages of production system, we wish to present the following key advantages. These advantages are the following: modularity, because each rule works independently of the others in the system; simple control structures because the controls are embedded in the productions rules, not in the algorithm; transparency because terminology used to describe the production rules are usually

derived from expert knowledge or based on observations of expert behaviour, making it easy for humans to interpret; dynamics because production rules can be added, deleted, or modified by one another on the fly, and they can be chained to achieve combinations of checks and actions. These are the main reasons why we choose to use this approach to solve inconsistency problems in software designs.

4.2 Quality Heuristics Formalization

Nearly all design heuristics and design patterns were originally intended to give hints to a human developer for creating an object-oriented design when developing a new system (forward engineering). They are given in natural language and tell software engineers what to do and how to do it. By contrast, we need the rules that can be checked on existing systems within a reengineering process. We need rules that can be checked automatically by a tool. This implies having an exact formal definition for design heuristics, design patterns, and anti-patterns.

Our approach is to express them in a more generic and more independent way which is based on a UML meta-model, that is, structural info plus some constraints. As design information recaptured from source code is represented in this way, we can definitely search occurrences of design heuristics, design patterns and anti-patterns in design information, we have to express design heuristic, design pattern and anti-pattern in meta-model. We, thereafter, use the expression "quality heuristic" to name those design heuristics, design patterns and anti-patterns that can be formalized and used in automation.

4.2.1 Meta-model

The UML meta-model defines the complete semantics for representing object models using UML [UML, 1997]. It defines class structures and their relationships. Because both design information reengineered and quality heuristics are represented in a meta-model, the meta-model must hold sufficient information to detect design heuristics, design patterns and anti-patterns in recovered software design information. That

information composes the meta-model that describes the basic elements on which our system constructs.

The meta-model used in our work is evolved from a UML semantic model [UML, 1997] and other works described in [DEMEYER et al., 1998] and [KELLER et al., 1999]. A meta-model defines the main entities of an object-oriented design, which includes static parts (package, classifier, attribute, operation, and parameter), relationships among them (dependence, accomplishment, inheritance), as well as dynamic parts such as object instantiation, object destruction, operation invocation and attribute access.

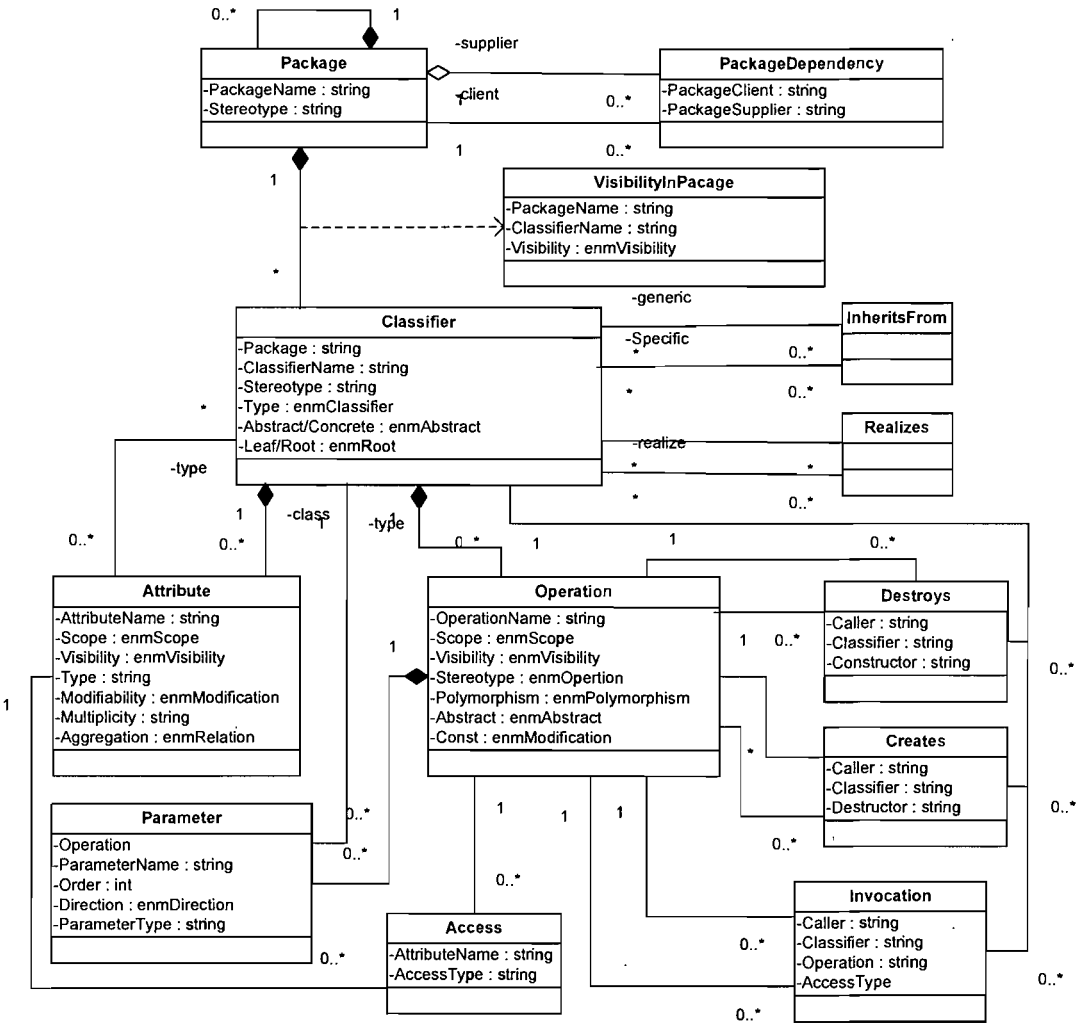


Figure 7 UML Meta-Model

Figure 7 illustrates the main entities of the meta-model in UML notation [UML, 1997] with the basic constructions.

4.2.2 Automation Degree

Our preliminary research has revealed that not all heuristics have the same degree of automation. In fact, some heuristics cannot be automated at all. A number of heuristics that fall into this category require specific knowledge and understanding of the domain modeled. Examples include RH3.6, "Model the real world whenever possible" and RH2.11, "Be sure that abstractions that you model are classes and not simply the roles objects play." Other heuristics are relatively straight forward, such as RH5.6, "All abstract classes must be base classes". However, the degrees of automation for many of the heuristics occur between these extremes. Where the heuristic itself may not be directly measurable, it is still possible to measure aspects of the software that might indicate whether the heuristic is being followed. RH2.8, "A class should capture one and only one key abstraction", for example, is hard to measure directly, as it is difficult to identify key abstractions. We can, however, relate this to other heuristics that can be quantified, for example RH4.6 "Most of the methods defined in a class should be using most of the data members most of the time". Additionally, we can measure indirect quantities such as LCOM [Chidamber and Kemerer, 1991], which, like RH4.6, might suggest whether the initial heuristic is being followed or not.

In general, design patterns and anti-patterns have a higher degree of automation than do design heuristics because patterns are described in explicit and organized form; they both have a graphical representation of the classes in the pattern using a notation based on UML as well as collaborations among them. Although design patterns gain a higher degree of automation than do design heuristics in general, the rest of design heuristics are easier to recognize than design patterns if we separate those design heuristics that can not be automated from all the other design heuristics. That is because design patterns are more like micro architectures and have more complex elements and collaborations than design heuristics. Therefore, the ways of

formalizing design heuristics, design patterns, and anti-patterns are the same no matter what kind of structures they are and how complex they are.

4.2.3 Formalization

In order to formalize quality heuristics, we describe and express them in terms of a meta-model, which are independent from any specific quality heuristics. As explained in a previous section, design heuristics are simpler than design patterns. We will mainly explain how to formalize design patterns, and we will explain design heuristics if there are exceptions.

Detached from the pattern type, the solution part of a structural pattern defines an arrangement of software elements to solve a particular problem. Since the problem itself is not of interest here, the arrangement of software elements is formalized for the search in a neutral manner regarding the pattern type.

As this arrangement is in fact rather a template than a combination of concrete software elements, these template elements are called roles. Roles are placeholders that can be taken from concrete elements in the instance of the pattern. Each role has a type (e.g., classifier or association meta-model elements) to determine the kind of software elements that can act as the role. Since software elements allow the nesting of other elements, each role may contain several subroles representing nested elements.

However, the existence of roles and their nesting relations in between is not sufficient to express complex arrangements of software elements, so roles are enhanced by constraints. These constraints enforce certain properties of the concrete elements acting as the role. They define, for example, visibility or stereotype properties of meta-model's elements.

Furthermore they may refer to other roles to express particular relations like inheritance or parameter types. By default every role must be played exactly once in a

pattern, but it is possible to define multiplicities to give limitations for the amount of elements acting as a role in a pattern. The multiplicity provides a lower and an upper range as it is done for association ends in UML. A lower range of zero makes a role optional and an infinite upper range allows as many elements to act the role as possible.

In one word, a role has a base meta-class in the UML meta-model, and is played by instances of the meta-class that satisfy the properties specified in the role

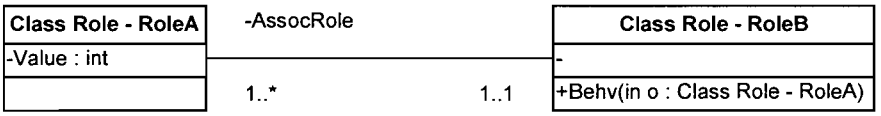


Figure 8 Independent Rule Representation

In Figure 8, there are two class roles RoleA and RoleB whose base is the Class meta-class (as denoted above their name), which constrains that only instances of the Class meta-class can play the roles.

RoleA has a structural feature role Str whose data type is integer. This further restricts the instances that can play RoleA in that they must possess a structural feature with integer data type. RoleB has a behavioural feature role Behv with a parameter role o whose type is RoleA. The class roles are connected by association role AssocRole that has two association end roles EndA and EndB. Each role defines a role multiplicity constraining the number of elements that can play the role. For example, RoleA has 1..* role multiplicity constraining that there can be one or more elements playing the role.

A role is associated with a set of meta-model level constraints. Meta-model level constraints specialize the UML meta-model by restricting the type of model elements that can play the role. They are represented graphically in diagram or textually in the Object Constraint Language (OCL). For example, in Figure 9 RoleA has three meta-model level constraints represented graphically: 1) the base meta-class constraint Class requires that a model element playing the RoleA role must be a class (an

instance of the Class meta-class), 2) the structural feature constraint Str demands that a model element playing the RoleA role must have one or more structural features playing the Str role, 3) the role multiplicity constraint 1 postulates that there must be exactly one class playing RoleA. It also has the following OCL metamodel-level constraints:

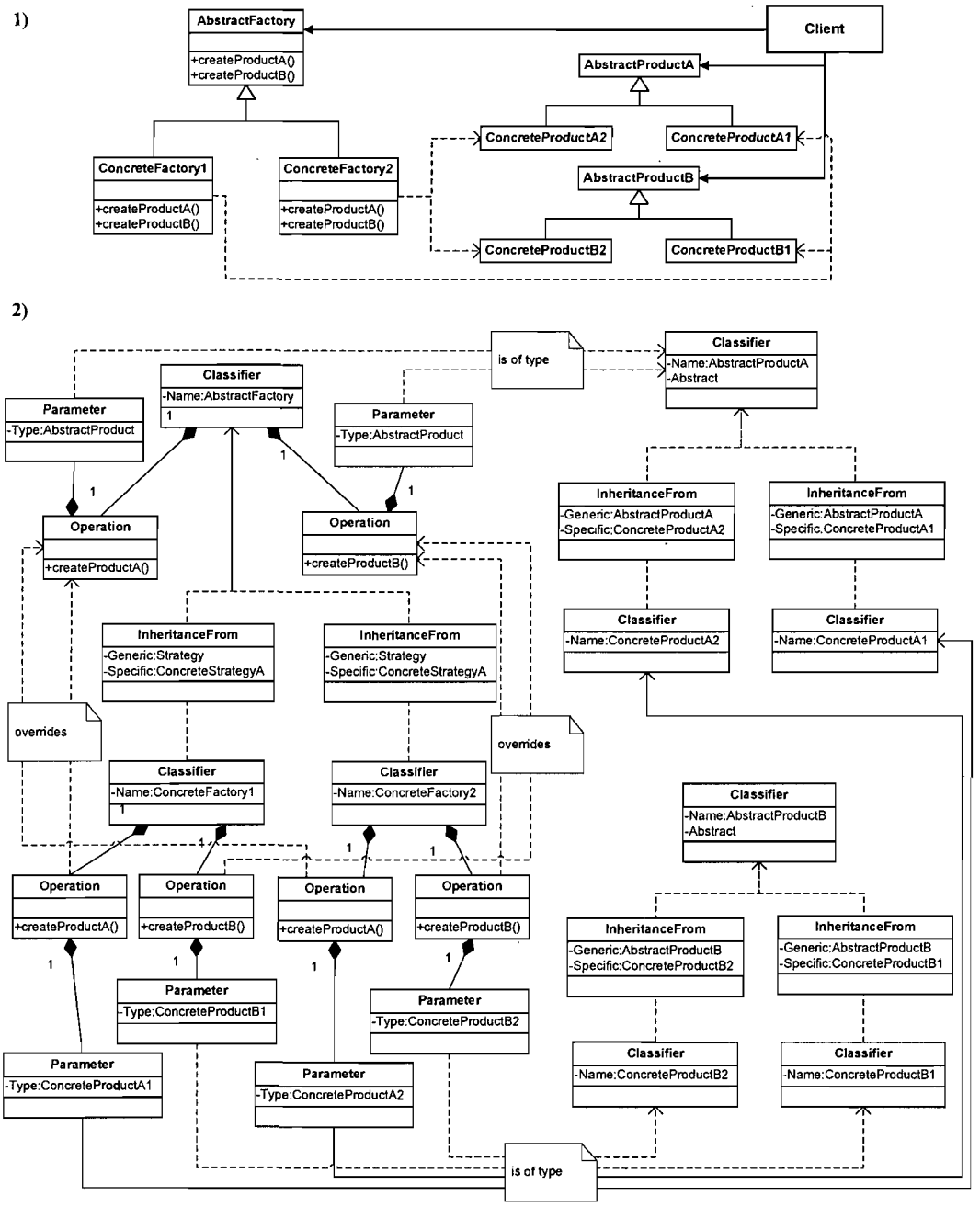


Figure 9 the design pattern Abstract Factor pictured as (1) a UML class diagram and (2) as an independent pattern definition.

- Classes playing *RoleA* must be concrete:
  context | RoleA **inv**: self.isAbstract = false
- Association ends playing *EndA* must have a multiplicity of 1:
  context | EndA **inv**: self.lowerBound() = 1 and self.upperBound() = 1
- Association ends playing *EndB* must have a multiplicity in the range of 1..*:
  context | EndA **inv**: self.lowerBound() = 1 and self.upperBound() = *

A more complicated example for the neutral representation of a design pattern [Gamma et al., 1995] is shown in Figure 9. Each element of the UML class diagram is translated to a role. The type of the UML element determines the type of the role. Child elements (e.g., parameters of operations) become subroles and properties of elements (e.g., abstraction or inheritance) are replaced by constraints for the corresponding role.

The graphical notation for the pattern definitions used in this article is a UML object diagram extended by some features of UML class diagrams. Object nodes represent roles – labelled with the name and the type of the role, separated by a colon. Aggregations express containments of subroles and constraints are represented by notation elements. For dependencies between roles dashed arrows are used.

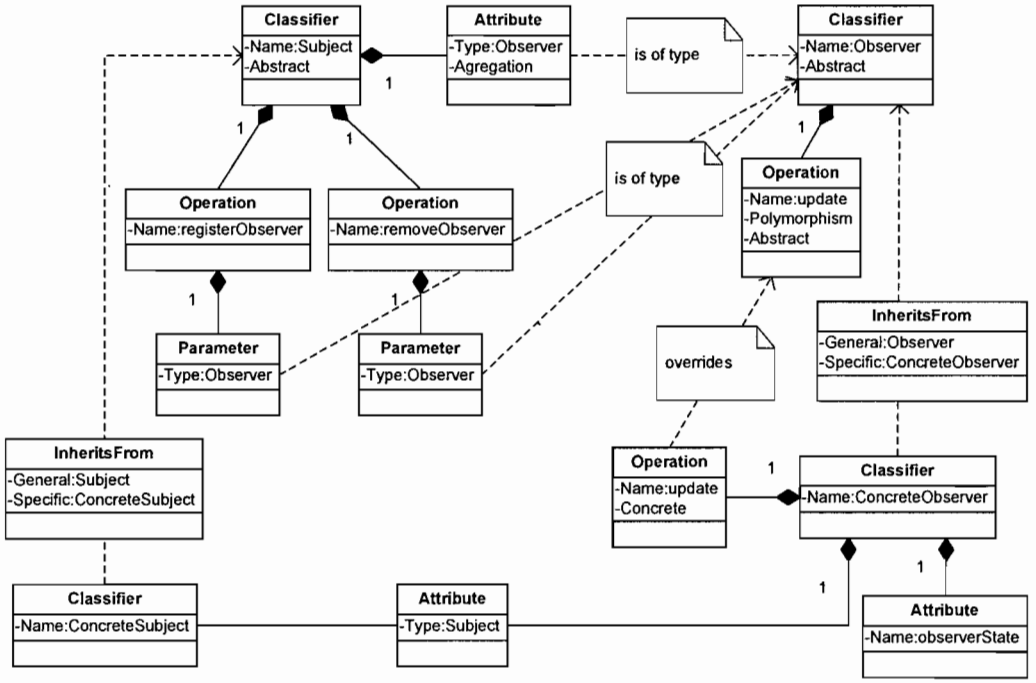


Figure 10 Observer pattern in meta-model

Figure 10 shows the neutral representation of another example, the design pattern Observer [Gamma et al., 1995], whose original structure diagram is depicted in Figure 4 at previous section.

Likewise we can formalize anti-patterns; for example, AntiSingleton anti-pattern is formalized from the Singleton design pattern [Gamma et al., 1995]. The Singleton design pattern is an object creational pattern as in Figure 11. It ensures that the designated class has only one instance and is able to provide a global access point to the instance. It can be used when multiple instances of a class are prohibited in a system, or to preclude the unnecessary object instantiations of a class.

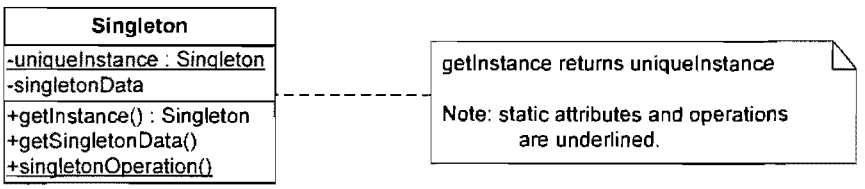


Figure 11 the Singleton design pattern

This Singleton pattern is violated if a Singleton class is used in the design, and other classes hold reference of Singleton's uniqueInstance object. Such a violation is an AntiSingleton anti-pattern. For example, a software engineer wanted, who was not familiar with the Singleton pattern, wanted to use an instance of a Singleton class inside a function; he passed the instance of the Singleton class to the function as one of this function's parameters. He did not know that he could use the instance of the Singleton inside the function directly, for instance, SingletonName.Instance(). Another kind of Anti-Singleton will be found when multiple instances of the class are instantiated by other classes or passed into several functions as parameters. If the software engineer who made this bad implementation is aware of the Singleton pattern, he will implement the class of "multiple instances" as a Singleton. Such a violation of the Singleton pattern is a ManyPointsofInstantiation anti-pattern.

4.3 Quality Heuristics Identification

In this section, we present the details of the application of the production system on automating quality heuristics identification from UML design models. First, we introduce the general mechanism of the method. Next, we define working memory elements for UML constructs and quality heuristics elements.

4.3.1 General Mechanism of the Method

First, when legacy source codes undergo a reverse engineering process, their design information is recovered according to a meta-model that defines the concepts needed according to the facts in the deductive database generation, then the artefacts, which will be presented in the next subsection, are generated and inserted into the WM. When changes are made in the working memory, the antecedent conditions of all of the rules are tested for applicability. Among all the applicable rules in the conflict set, one is selected each time for execution. According to the consequent action of the selected rule, matched quality heuristics are shown to the end user.

4.3.2 Definitions of Working Memory Elements

We saw previously that a general WME is represented as the following:

$$(\texttt{type attribute}_1\texttt{:value}_1 \ \ldots \ \texttt{attribute}_n\texttt{:value}_n)$$

In order to represent specific knowledge of the UML meta-model, we have added the following additional notations. A pair of < > brackets enclose the acceptable values or types of values. If the text in the < > brackets is in *italic*, it simply describes the requirement of the value for the given attribute. If the text in the < > brackets is regular, it provides the actual values that are available for the attribute, and choices among different values are separated by /.

Moreover, the notation of the UML meta-model elements is represented in terms of object-oriented design primitives in a predicate-like format. Each design element consists of two parts: Type and Argument. The type part contains the name of an

entity or a relation in object-oriented design, such as class, inheritance, etc. The argument part contains general information about an entity or a relation such as the information on the participants of an inheritance relation. In the following, we present the syntax and the meaning of the design primitives used in this paper:

- Package: Represents a package definition. A package is a general mechanism used for organizing model elements in groups. These packages group elements show strong cohesion with each other and loose coupling with elements in other packages.

```
(package
      name:<string> stereotype:<string> parentName:<string>)
```

- PackageDependency: represents a dependency relationship between two packages; one is a client and the other is a supplier.

```
(packageDependency
      packageClient:<string> packageSupplier:<string>)
```

- Classifier: indicates a class, interface or basic data type definition.

```
(classifier
      package:<string> classifierName:<string>
      stereotype:<string>
      type:<class/interface/primitive>
      abstractOrConcrete:<abstract/concrete>
      isLeaf:<leaf/notleaf> isRoot:<root/notroot>)
```

- VisibilityInPackage: represents the classifier visibility (public, protected, private) in a package.

```
(visibilityInPackage
      package:<string> classifier:<string>
      visibility:<public/protected/private>)
```

- Realizes: Represents the existence of a realization relationship between two classifiers as, for example, a class implementing the operations defined by an interface.

```
(realizes
      classifier:<string> realizedClassifier:<string>)
```

- InheritsFrom: Represents an inheritance relationship between two classifiers.

```
(inheritsFrom
      specificClassifier:<string> genericClassifier:<string>)
```

- Attribute: indicates the presences of an attribute or a pseudo-attribute (an association with a classifier) in a class definition; it carries association and dependency relationships. It also captures the attribute scope (class or instance), its visibility (public, protected or private), if its value can be modified or not, its type, multiplicity (1 or many), and the semantic of the association with the attribute type (association, aggregation or composition).

```
(attribute
      classifier:<string> attributeName:<string>
      scope:<class/instance>
      visibility:<public/protected/private>
      type:<string> modifiability:<const/nonconst>
      multiplicity:<1/many>
      aggregation:<association/aggregation/composition>)
```

- Operation: represents an operation defined in a class, indicating its scope (class or instance), its visibility (public, protected, private), stereotype (constructor, destructor, read accessor, write accessor, other), if the operation can be redefined by subclasses, if it modifies the object state, and whether it is only a declaration (Abstract) or a method implementation (Concrete).

```
(operation
      Classifier:<string> operationName:<string>
      Scope:<class/instance>
      visibility:<public/protected/private>
      stereotype:<string>
      polymorphism:<string>
      abstractOrConcrete:<abstract/Concrete>
      modifiability:<Const/NonConst>)
```

- Parameter: represents a parameter expected by an operation. The direction indicates whether it is an input, output, input/output or a return value.

```
(parameter
      operationName:<string> parameterName:<string>
      order:<string> parameterType:<string>
      direction:<input/output/inputoroutput/return>)
```

- Creates: represents the invocation of a class constructor resulting in an object instantiation. This predicate indicates which operation is responsible for the invocation (caller).

```
(creates
      classifierCaller:<string> caller:<string>
```

```
      classifierCallee:<string> constructor:<string>)
```

- Destroys: represents the invocation of an object destructor. This predicate indicates which operation is responsible for this invocation.

```
(destroys
      classifierCaller:<string> caller:<string>
      classifierCallee:<string> destructor:<string>)
```

- Invokes: represents the invocation of an operation. Caller corresponds to the method where this invocation occurs; Classifier is the type of the called object. Operation is the name of the called operation and AccessType tells how the called object is known in the caller method. AccessType can be the object itself (self), a parameter, an object created in the caller method (local object), a global scope object or an attribute of the caller object.

```
(invokes
      classifierCaller:<string> operationCaller:<string>
      classifierCallee:<string> operationCallee:<string>
      accessType:<self/parameter/local/remote/attribute>)
```

- Access: indicates that an operation accesses a particular attribute. This access can be value retrieval or modification, an operation call or even passing this attribute as an argument in some operation call.

```
(access
      Operation:<string> attribute:<string>
      accessType:<valueAccess/operationCall/Argument>)
```

4.3.3 Quality Heuristic Rules

Now that we have shown how to populate the working memory, we will develop a knowledge base. The knowledge base is the collection of rules that make up a rule-based system. We are going to define production rules for the quality heuristics identified in Chapter 3. Each rule has a description in text and formalization in a production system language as defined above. The description of each rule characterizes the intension of a quality heuristic, but the formalization of the antecedent condition of a rule captures the distinct structures and relationships of the quality heuristic. The consequent action of a rule usually adds elements that include a

message about the recognized quality heuristic and each of the modeling elements involved.

4.3.3.1 Design Heuristics Rules

A) All data should be hidden within its class

1. Description: The violation of this heuristic effectively throws maintenance out the window. The consistent enforcement of information hiding at the design and implementation level is responsible for a large part of the benefits of the object-oriented paradigm. If data is made public, it becomes difficult to determine which portion of the system's functionality is dependent on that data.
2. Rule: all data should be hidden within its class.

```
IF (classifier classifierName:?cn)
     (attribute
          classifierName:?cn
          attributeName:?attrName
          visibility:Public)
THEN
     (assert: ClassDataShouldHidden(?cn))
```

B) Classes should not contain more objects than a developer can fit in his or her short-term memory. A favorite value for this number is six.

1. Description: The rationale behind this heuristic is that most of the methods defined on a class should use most of the data members most of the time. Assuming this is true, the implementors of a method will need to think about all of the data members while writing the method. If the developer cannot keep all of the data in his or her short-term memory, then items will be omitted and bugs will creep into the code. The standard number of seven plus or minus two is widely accepted in the world of psychology as the number of items most people can keep in their short-term memory. We choose six to take into consideration people with poor short-term memories and the fact that most methods take an argument or two, which must be considered in addition to the data members.

2. Rule: A class should not contain more than six objects

```
IF (classifier classifierName:?cn)
```

```
     (attribute classifierName:?cn attributeName:?attrName)
     (attribute listSize()>= 6)
THEN
     (assert:ClassContainMT6Objects(?cn, listAttributeNames))
```

C) Law of Demeters

1. Description: A well known object-oriented design standard is the Law of Demeter, which states that, "The methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only."
2. Rule: A design model should obey the Law of Demeters

```
IF (sequenceMessage id:m1 return:b pid:p)
     (sequenceObject name:b type:L pid:p)
     (sequenceMessage id:m2 ∧ { ≠ m1} to:L return:c pid:p)
     (sequenceObject name:c type:K pid:p)
     (sequenceMessage id:m3 ∧ {≠ m1 } ∧ { ≠ m2 } to:K pid:p)
 THEN
     (Msg:"Violation of the Law of Demeter."
           (pid:s location:m1 type:sequenceMessage)
           (pid:s location:m2 type:sequenceMessage)
           (pid:s location:m3 type:sequenceMessage)
           (pid:s location:b type:sequenceObject)
           (pid:s location:c type:sequenceObject))
```

4.3.3.2 Design Patterns Rules

A) Decorator Pattern

3. Description: A Decorator consists of four classes: the component top class with a concrete component subclass and a decorator subclass; the latter has one or several further subclasses called concrete decorators.
4. Rule:

```
IF (classifer
     classifierName:?clsnmCpnt type:class)
     abstractOrConcrete:abstract)
  (classifer
      classifierName:?clsnmConcrCpnt
      type:class abstractOrConcrete:concrete)
  (classifer
      classifierName:?clsnmDeco ∧ {≠ ?clsnmCpnt }
      type:class abstractOrConcrete:abstract)
  (classifer
```

```
        classifierName:?clsnmConcrDeco∧ {≠ ?clsnmConcrCpnt }
        type:class abstractOrConcrete:concrete)
    (inheritanceFrom
        specificClassifier:?clsnmConcrCpnt
        genericClassifier:?clsnmCpnt)
    (inheritanceFrom
        specificClassifier:?clsnmDeco
        genericClassifier:?clsnmCpnt)
    (inheritanceFrom
        specificClassifier:?clsnmConcrDeco
        genericClassifier:?clsnmDeco)
    (operation
        classifier:?clsnmCpnt operationName:?opnmCpnt
        Scope:class abstractOrConcrete:abstract)
    (operation
        classifier:?clsnmConcrCpnt operationName:?opnmCpnt
        scope:class abstractOrConcrete:concrete)
    (operation
        classifier:?clsnmDeco operationName:?opnmCpnt
        scope:class abstractOrConcrete:abstract)
    (operation
        Classifier:?clsnmConcrDeco operationName:?opnmCpnt)
        scope:class abstractOrConcrete:concrete)
    (attribute
        classifier:?clsnmDeco attributeName:?attrnm
        Scope:class visibility:public type:?clsnmCpnt
        multiplicity:1 aggregation:aggregation)
    (invokes
       classifierCaller:?clsnmDeco operationCaller:?opnmCpnt
       classifierCallee:?clsnmCpnt operationCallee:?opnmCpnt)
THEN
    (assert: Decrator(?clsnmCpnt, ?clsnmConcrCpnt,
             ?clsnmDeco, ?clsnmConcrDeco))
```

B) Adapter Pattern

1. Description: A Decorator consists of four classes: the component top class with a concrete component subclass and a decorator subclass; the latter has one or several further subclasses called concrete decorators.
2. Rule:

```
IF (classifer
        classifierName:?clsnmTarget type:class)
        abstractOrConcrete:abstract)
    (classifer
        classifierName:?clsnmAdapter
        type:class abstractOrConcrete:concrete)
    (classifer
        classifierName:?clsnmAdaptee∧ {≠ ?clsnmAdapter}
        type:class abstractOrConcrete:concrete)
```

```
   (inheritanceFrom
       specificClassifier:?clsnmAdapter
       genericClassifier:?clsnmTarget)
   (operation
       classifier:?clsnmTarget operationName:?opnmTarget
       scope:class abstractOrConcrete:abstract)
   (operation
       classifier:?clsnmAdapter operationName:?opnmTarget
       scope:class abstractOrConcrete:concrete)
   (operation
       classifier:?clsnmAdaptee operationName:?opnmAdaptee
       scope:class abstractOrConcrete:concrete)
   (attribute
       classifier:?clsnmAdapter attributeName:?attrnm
       scope:class visibility:public type:?clsnmAdaptee
       multiplicity:1 aggregation:association)
   (invokes
      classifierCaller:?clsnmAdapter
      operationCaller:?opnmTarget
      classifierCallee:?clsnmAdaptee
      operationCallee:?opnmAdaptee)
 THEN
  (assert: Adapter(?clsnmTarget, ?clsnmAdapter, ?clsnmAdaptee))
```

4.3.3.3 Anti-pattern Rules

Description: The antecedent condition of a pattern recognition rule formalizes one distinctive characteristic of the pattern and describes the violation of its usage.

Rule 1: When a Singleton pattern is used in a design, no other class objects should keep a reference to the singleton object. (Note that a Singleton pattern is recognized if the class has a static method returning an instance of the class and a static attribute that stores instances of this class.)

```
IF
      ;;; defining singleton design pattern
      (classifier name:?cn1)
      (operation
            classifierName:?cnSingleton
            operationName:?optcnSingleton
            returnType:?typecnSingleton
            scope:Class)
      (attribute
            classifierName:?cnSingleton
            attributeName:?attrcnSingleton
            type:?typecnSingleton
            scope:Class)
      ;;; define violation part
      (classifier (classifierName:?cnVoilation))
      (attribute
            classifierName:?cnVoilation
            attributeName :?attrVoilationName
            type ∧ { ≠} type:?typecnSingleton)
THEN
      (assert: AntiSingleton(?cnSingleton,
                  ?cnVoilation, ?attrVoilationName))
```

Rule 2: When multiple classes in a package are accessed from outside the package, a Façade pattern can be used and a Façade class should be placed as a common interface to the package.

```
IF
      (classifier classifierName:?cn1 package:?pn1)
      (classifier
            classifierName:?cn2 ∧{ ≠ ?cn1}
            package:?pn2 ∧{ ≠ ?pn1})
      (attribute classifierName:?cn2 type:?cn1)
      (classifier classifierName:?cn3 ∧{ ≠?cn1} package:?pn1)
      (classifier
            classifierName:?cn4 ∧{ ≠?cn2}
            package:?pn3 ∧{ ≠ ?pn1})
      (attribute classifierName:?cn4 type:?cn3)
THEN
      (assert: AntiFacade(
            classesInaPackage(?cn1, ?cn3),
            classesFromOtherPackage(?cn2, ?cn4)))
```

Rule 3: Anti Common-code Private Function detects the definition of methods in the public interface of a class that are used only as auxiliary methods for the implementation of other methods of this class. This contradicts the design heuristic, "Do not put implementation details such as common-code private functions into the public interface of a class" [Riel, 1996].

```
IF
  (classifier classifierName : ?clsTarget)
  (operation classifier : ?clsTarget visibility : public
            operationName : ?optNameTarget)
  ;Internal Client
  (operation classifier : ?clsInternal
            operationName : ?optNameInternal)
  ;Same Hierarchy
  (sameHierarch classifier : ?clsInternal classifier : ?clsTarget)
  (invokes classifierCaller : ?clsInternal
        operationCaller : ?optNameInternal
        classifierCallee : ?clsTarget
        operationCallee : ?optNameTarget)
  (?optNameTarget == ?optNameInternal)

  ;External Client
  (not (and (and
           (and (operation classifier : ?clsExternal
                    operationName : ?optNameExternal)
                (operation classifier : ?clsTarget
                    operationName : ?optNameTarget))
            ;Not Same Hierarchy
            (not (sameHierarch classifier : ?clsInternal
                  classifier : ?clsTarget)
               (invokes classifierCaller : ?clsExternal
                        operationCaller : ?optNameExternal
                        classifierCallee : ?clsTarget)
                        operationCallee : ?optNameTarget)))
THEN
  (assert: (antiCommoncodePrivateFunction
            (classifier ?clsTarget)(operation ?optNameTarget))))
```

4.3.3.4 Coalesce Rules

The rules we have written so far can sometimes generate multiple facts for the same rule. We could complicate all the previous rules such that they would not generate the duplicate recommendations, or we could simply allow them to be created and then clean them up at the end. We have chosen to take the latter route. A single rule coalesce rules combine multiple facts for the same rule.

Rule 1: Coalesce Abstract Factory Facts will query on all abstract factory facts, merge them into a new fact when it is found that two facts are different and delete those two facts.

```
IF
      ?r1<-(abstractFactory-pattern
                  abstractFactory : ?clsAF
                  concreteFactories : $?clsCF1
                  abstractProducts : $?clsAP1
                  concreteProducts : $?clsCP1)

      ?r2<-(abstractFactory-pattern
                  abstractFactory : ?clsAF
                  concreteFactories : $?clsCF2
                  abstractProducts : $?clsAP2
                  concreteProducts : $?clsCP2)
      (test (neq ?r1 ?r2))
THEN
      (retract ?r1 ?r2)
      (assert (abstractFactory-pattern (abstractFactory ?clsAF)
                  (concreteFactories =(union$ $?clsCF1 $?clsCF2))
                  (abstractProducts =(union$ $?clsAP1 $?clsAP2))
                  (concreteProducts =(union$ $?clsCP1 $?clsCP2)))))
```

Appendix D – Quality Heuristic Jess Rules - Quality Heuristics Jess Rules lists all quality heuristics Jess rules that are implemented in our prototype tool.

5 Implementation

In this chapter, we describe the current prototype implementation. First, we describe the architecture, the functionalities of the system, and the implementation details. The subsequent sections contain the description of the graphical user interface.

5.1 Implementation Architecture

Our prototype is a Java implementation. It uses Jess [Jess, 2002] — an off-the-shelf Java Rule Engine that implements the RETE algorithm, to execute production rules. It also use Antlr [Antlr, 2003], which is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions, to reverse engineering source code. Moreover, its GUI is developed by using SWT with the help of WindowBuilder Pro. The architecture of our prototype system is illustrated in Figure 12.

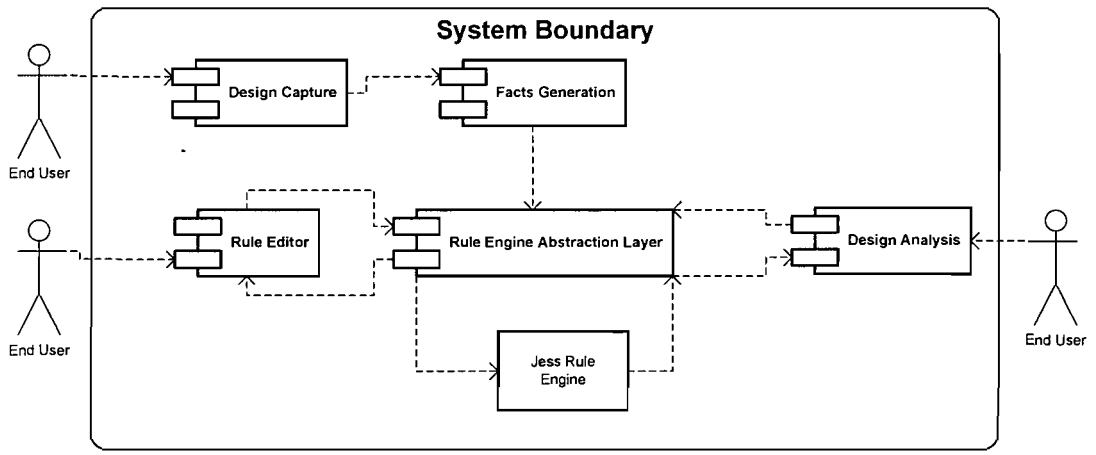


Figure 12 Implementation architecture

There are five components in our prototype system:

1. The **Design Discovery** component parses source code and generates a SDG (Semantic Directed Graph) graph which carries the design information of the source code.

2. The **Facts Generation** component traverses pre-generated the SDG graph and generates knowledge facts representing recovered design information based on predefined meta-model templates.

3. The **Rule Engine Abstraction Layer component** allows replacing Jess by another rule engine.
4. The **Rule Editor** component provides a user-interface that allows the user to manage the rule base. The user can display the status of rules and add/delete/modify rules.
5. The **Design Analysis** component applies a knowledge base to facts holding design information and reports the deduced results of good and problematic constructs.
Next we will introduce the implementation details of those components.

5.1.1 Design Discovery

The outputs of the design discovery process supply the fundamental data for other components sitting atop of it; it must provide all essential raw values, not only the structural information, but also the information related to the methods implementation. Otherwise, the deduced results from a rule engine will be inaccurate or wrong. Based on the information retrieved by this extraction, it is possible to know the following: the attributes manipulated by a particular method; how these attributes are manipulated (read, write, parameter, operation invocation); a method stereotype ('constructor", "destructor", "read accessor", "write accessor", among others), and which collaborations are necessary for the implementation of a particular method. The analysis of method invocations is done to gather information about dependencies between types and not between objects, since the latter would require a run-time analysis of the system.

In our prototype, we have built the design recovery component that accumulates complete symbol table information using a Java parser generated by the ANTLR [Antlr, 2003] parser generator. There are two main components in the design recovery component: a parser and a symbol table. There is a partial Antlr Java grammar in Appendix A.

Java programs are composed of definitions and references. You define classes, methods and variables and reference them in statements and other definitions. Each class, method and variable has a name. When used in a parser, these names are called

symbols. A symbol represents one specific entity defined in your program. Multiple symbols might appear to be the same, but are separate definitions based on their location in the source files. A Design-recovery component needs to keep track of every symbol that is defined and where those symbols are referenced. A parser tracks symbol information in a data structure called a symbol table. The symbol table contains a list of all the Java packages it has encountered, a table of all unique strings, and a Stack of scoped-definitions that represent the current parse state. The design-recovery component uses a class called SymbolTable to represent the symbol table.

Every symbol is defined within a certain scope. A scope is a section of code in which a definition is visible and usable. Scopes can be nested. For example, a class defines a scope that contains all the variables and methods defined in that class. A method defines a scope that contains all its parameters and local variable definitions. A stack is commonly used to represent this scope nesting. The innermost scope, containing the text being parsed, is always at the top of the stack. As the parse proceeds into new scopes, the new scopes are pushed onto the stack. As the parse exits scopes, the scope definition is popped from the stack. One of the key elements of name lookup is the examination of each element of the stack, starting at the top, to see if that scope contains the requested name.

The design recovery component performs its processing in three phases: parse the source code to determine definitions and references; resolve references in the contents of the symbol table; and build up SDG (System Dependency Graph). Here are the phases described in more detail:

1. **Determine Definitions and Note References**: The first phase of the design-recovery component walks through all the source code in the specified directories and their subdirectories. The parser collects information on the following constructs:
    - Package Definitions
    - Class Definitions
    - Interface Definitions

- Method Definitions
- References to other symbols

For each of the above definitions found, a new symbol is created. Some symbols, like classes, reference other symbols such as a superclass that is being extended as part of a class definition. During the first pass, these references might not yet be available; they could be defined later in the same source file or in another file that has yet to be parsed. Because of this, any references to superclasses, implemented interfaces, return types for methods and parameter types are stored as placeholders according to their names only. These placeholders are held in an instance of the DummyClass class. These will also be resolved during phase two.

2. **Resolve Definition References**: The result of the source-code parse is a symbol table that lists all constructs defined in the source files. Some of these definitions reference other definitions, for superclasses, variable types and so on. This second phase will walk through all definitions in the symbol table and resolve those references. Most of the symbol table classes implement a method called resolveTypes that is used to perform this resolution. At the conclusion of this pass, the symbol table contains all symbols defined in the parsed source files and proper resolutions to defined symbols.
3. Build SDG: This final phase looks at the data contained in the symbol table and Build up SDG graph which include call and variable reference graph. At this time the symbol table looks like what is depicted in Figure 13; it is actually a kind of graph whose edges are function calls and variable references. DM means Data Member, MF means Member Function in the diagram.

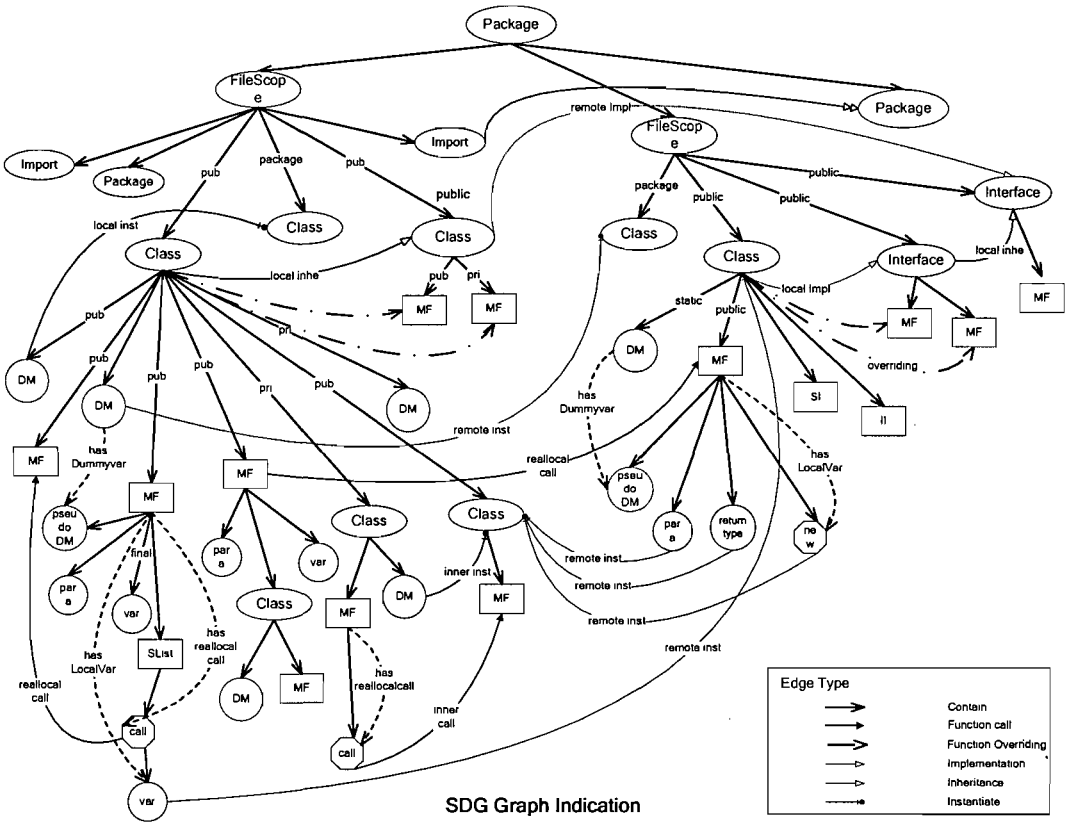


Figure 13 SDG Graph

Having covered the background information, Figure 14 shows the design-recovery component class diagram. The functionality of each class and the relationships between these classes are as follows:

- **SymbolTable**: This is the main class in the design recovery component. It provides a list of all packages that have been parsed, a stack representing the current lexical scope, and a table of unique Strings that have been read during the parse. This is the source of all symbol lookups; when resolving references, the requester asks the SymbolTable to look for a name, and it then searches the scope stack and the parsed packages to find the name. The SymbolTable also provides several methods to create instances of the other classes used in the symbol table package.

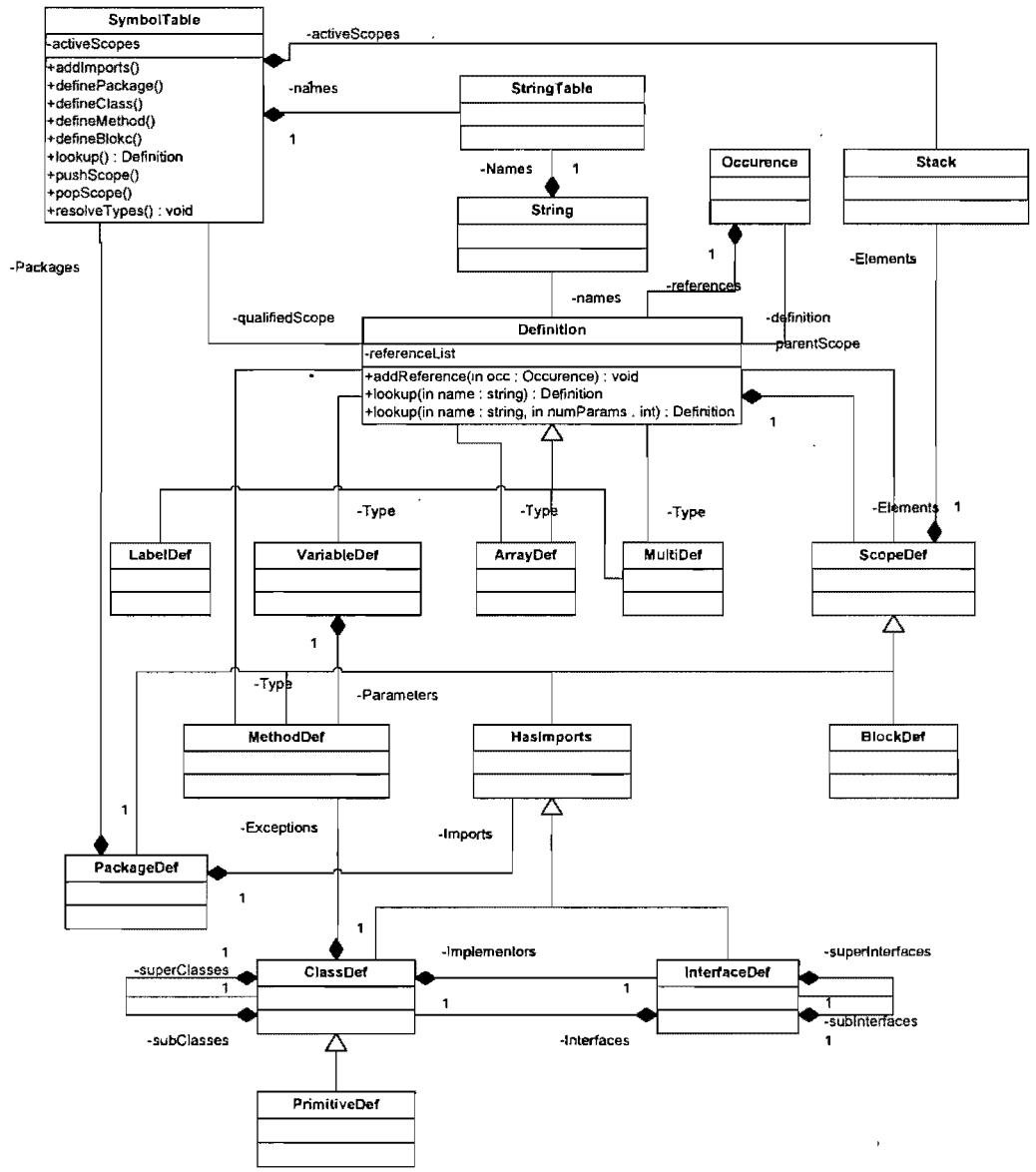


Figure 14 Design Discovery Diagram

- **StringTable**: When parsing a source file, the same strings usually occur again and again. Rather than store these Strings as separate String instances, we store each unique String in the StringTable for more efficient use of memory.
- **Stack**: This is a java.util.Stack that keeps track of the nested scopes containing the current parse position. As Java constructs such as classes, methods and packages are recognized, the parser asks the SymbolTable to push a new containing scope on the stack. This stack provides an appropriate lookup mechanism for most names. When a name is read, the SymbolTable's

lookup method checks each scope on the stack (from the most-nested scope to the outermost scope) to see if the name is found inside that scope.

- **Occurrence**: This class identifies a line in a source file. It is used to store the location of the definition for a class, interface, package, method or variable, and references to those constructs.
- **Definition**: Every Java construct stored in the symbol table is a Definition. This class provides a common base for all symbol definitions, and includes information such as an Occurrence to track the location where the symbol was defined, a list of references to the symbol, and its name.
- **ScopedDef**: Some symbols can actually contain definitions of other symbols. For example, a class can contain definitions for variables and even other classes. These types of symbols are grouped into a common base class called a ScopedDef. ScopedDef keeps a list of other constructs that were defined within it. ScopedDef objects are the objects that are stored in the scope Stack for name lookup.
- **HasImports**: A further extension of a ScopedDef is one that HasImports. A HasImports object is one that makes use of Java's import statements to access names in other packages. HasImports keeps a list of packages and classes that are names in Java import statements. Classes and interfaces are the constructs that are grouped under this base class.
- **ClassDef**: This is the definition of a Java Class. Classes can have a superclass (a reference to another ClassDef), a list of interfaces that it implements, and a list of other classes that extend it.
- **InterfaceDef**: This represents a Java interface. Interfaces can have several super-interfaces, a list of classes that implement it, and a list of other InterfaceDefs that extend it.
- **PrimitiveDef**: Java has several primitive types, such as int, long and boolean. When variables or constants of these types are passed to a method, widening conversions can be performed to make the actual parameters match the formal parameters of the method. This behaviour is very similar to the process in which objects can be widened to their superclass type to match a formal

parameter type. To take advantage of this similarity, PrimitiveDef is a subclass of ClassDef, and each primitive type is made a subclass of a primitive type to which it can widen.

- **BlockDef**: This is a wrapper for an unnamed { } delimited block of statements. It provides a scope for nested variable definitions.
- **PackageDef**: A package in Java is a collection of classes and interfaces. The contents of a package may be spread across several files, each with the same "package" statement at the top. Our SymbolTable collects PackageDef objects into a list of all packages that have been parsed or referenced. In addition, PackageDef objects can be referenced from import statements (which will be searched when a class is not found in any other context during symbol lookup.)

5.1.2 Facts Generation

The facts generation component comes into action, generating a deductive database that represents the facts captured from this design information after the design information carried by SDG graph becomes available as the result of a reverse engineering process performed by the design-recovery component.

These facts are represented in predicates corresponding to the constructions defined in the meta-model for object-oriented software; the meta-model defines the entities and relationships that are relevant to design patterns and anti-patterns identification, including not only structural elements, but also dynamic elements such as object instantiation and method calls, for instance. The details of the meta-model are explained in section 3.5.1, and its UML diagram is drawn in Figure 7.

From a structural point of view, the facts generation component generates a set of facts that describe all the classifiers found in a model (classes, interfaces, and basic data types), how those classifiers are organized into packages, their attributes and operations, including detailed information about each one (visibility, type, scope, parameters, among others). The associations, aggregations, and compositions are

captured as pseudo-attributes [UML, 1997], i.e., the pseudo-attribute may be used in the same way as an attribute of a classifier. The inheritance relationships between classifiers and the realization of a classifier are also captured by specific predicates.

From a behavioural point of view, this module generates facts about the implementation of each method. Each object instantiation and destruction, method invocation (of the same class or not), and attribute access (read or write) is captured as a predicates. The capture of these behavioural elements is essential for the identification of many design problems, such as those related to object coupling.
The information generated by the facts generation component is the source for pattern detection, representing the result of the analysis of structural and behavioural elements of an OO design.

5.1.3 Rule Engine Abstraction Layer

Although our prototype uses Jess as rule engine currently, it is better to avoid being locked into using one vendor's product. Each rule engine has its own strengths and weaknesses. Unfortunately, no standard rule language is supported by all (or even some) of the major rule-engine vendors. That is why we defined a rule-engine abstraction layer which allows us to develop our rules in a vendor-neutral language.

In the Jess language, rules are represented as *defrule* constructs. Other rule engines have their own ways of representing rules. In general, a core of common concepts can be expressed in all rule languages. Although each language represents these concepts differently, they all represent the same underlying information. If rules that only this common core of concepts are developed in a neutral, flexible representation, then they can easily be translated into the native format supported by a specific rule engine as needed.

We select XML as this neutral, flexible representation in our prototype tool. It brings tremendous benefits to dealing with rule storage, editing, and retrieval at ease. In the

prototype quality heuristic rules, facts and meta-model template are all expressed in XML format. For example, a rule like this:

```
(defrule AnimalRule2
  ?animal <- (animal (has-hair TRUE))
=>
  (modify ?animal (type mammal)))
```

can be represented as the following XML document

```
<?xml version="1.0"?>
<!DOCTYPE rulebase SYSTEM "jess.dtd">
<rulebase>
  <rule name="AnimalRule2" priority="10">
    <lhs>
      <pattern name="animal" binding="animal">
        <slot name="has-hair">
          <constant>TRUE</constant>
        </slot>
      </pattern>
    </lhs>
    <rhs>
      <function-call>
        <head>modify</head>
        <variable>animal</variable>
        <constant>(type mammal)</constant>
      </function-call>
    </rhs>
  </rule>
</rulebase>
```

The XML rule format can be transformed back to its original format using an XSLT script. XSLT programs are declarative rather than procedural—just like rules in Jess. In fact, an XSLT program is precisely a list of rules for transforming specific parts of an XML document into some desired result format. In the Appendix C, it lists an XSLT script that is used in our prototype to translate XML Jess rules into Jess rules.

5.1.4 Quality heuristics Editor

Although it is called the quality heuristics editor component, it actually is a knowledge-base management component combined with a GUI to manipulate that knowledge. This component stores quality heuristics, which are refined from design heuristics, design patterns, and anti-patterns, as Jess deductive rules. Since each quality heuristic is captured by rules, the knowledge base component allows the

definition of new rules so that new quality heuristics can be detected. Therefore, the knowledge base can evolve as a result of the organization experience in developing and maintaining OO systems. These rules are expressed using the same predicates employed in the OO design facts representation, as described earlier.

Because the editor of our prototype is just a simple text editor, we use Jess rule DTD (Data Type Definition) to check that the end users' modifications are valid. The detail of Jess rule DTD is listed in Appendix B.

5.1.5 OO Design Analysis

Our last prototype component, the design analysis module, is responsible for analyzing the facts deductive database corresponding to the OO design being verified and fro trying to find some match with the constructions captured by the quality heuristics component, using the Jess inference machine.

The user selects one or more problem categories, and one or more problems classified in the selected category. This module identifies all the design fragments that satisfy the Jess rules defined for the detection of those problems.

A report is generated indicating each problem found, the elements responsible for its occurrence, and also possible ways to overcome it. A solution section of the report will gives possible solutions corresponding to the violations of quality heuristics information captured by the expertise capture module. For example, if the tool finds the violation of ClassDataShouldBeHidden quality heuristic, it will generate a report with information of the attribute and the class where it occurs. The report will also show the solution section that a possible solution would be to move the attribute to the private area of the class, and to create accessor methods (get and set methods) for retrieving and modifying this attribute. Another result that can be achieved with this module is the identification of design patterns used in the evaluated design. By selecting the desired design patterns, the user obtains as a result, a report indicating the design patterns found and also all the elements matching each participant role in

the pattern. For example, if the tool detects an instance of the AbstractFactory design pattern, it shows not only the presence of this pattern in the design but also all classes corresponding to the Abstract Factory, Concrete Factories, Abstract Products and Concrete Product participants found in this design pattern instance.

5.2 GUI

In this section, the graphical user interface (GUI) of the tool is described. Figure 15 shows the main window of our prototype tool. There are "Design Recovery", "Facts Generation", "Knowledge Base" and "Design Analysis" menu items on the main menu, which are the main functionalities described in the previous section.

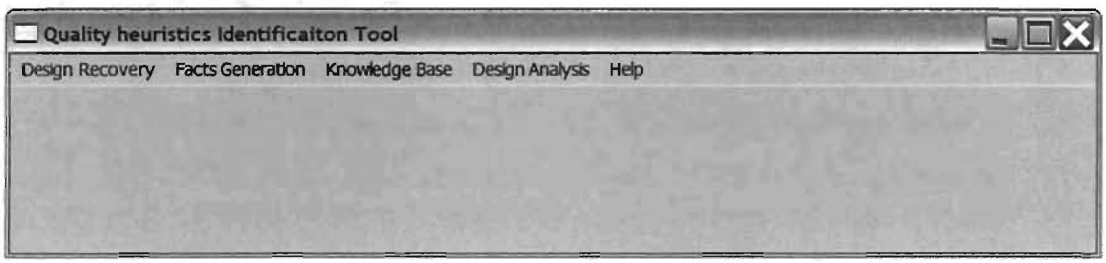


Figure 15 Main Window of the Prototype

Design Recovery's submenu is shown in Figure 16. The end user can select an Eclipse Java project with "Open project" menu. It will mainly setup a class path according to Eclipse project's settings. "Recover" will actually parse all the java files in the project and generate an SDG graph. The new generated SDG graph will be shown in the SDG graph viewer window as depicted in Figure 17.

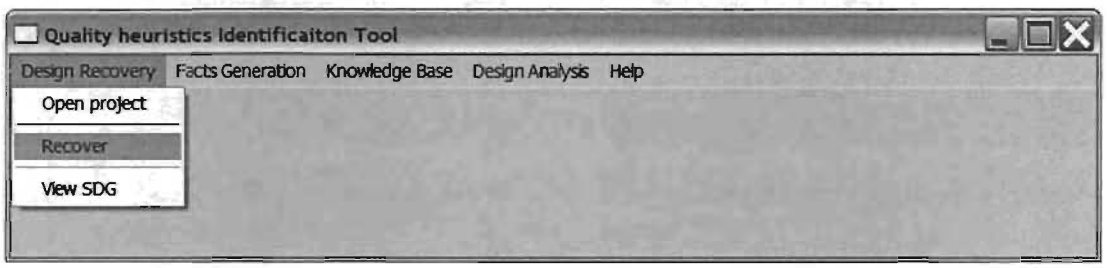


Figure 16 Submenu of Design Recovery

The SDG graph viewer window consists of two panes: the left pane shows the SDG graph in a tree structure; the end user can view the graph by clicking on each node to

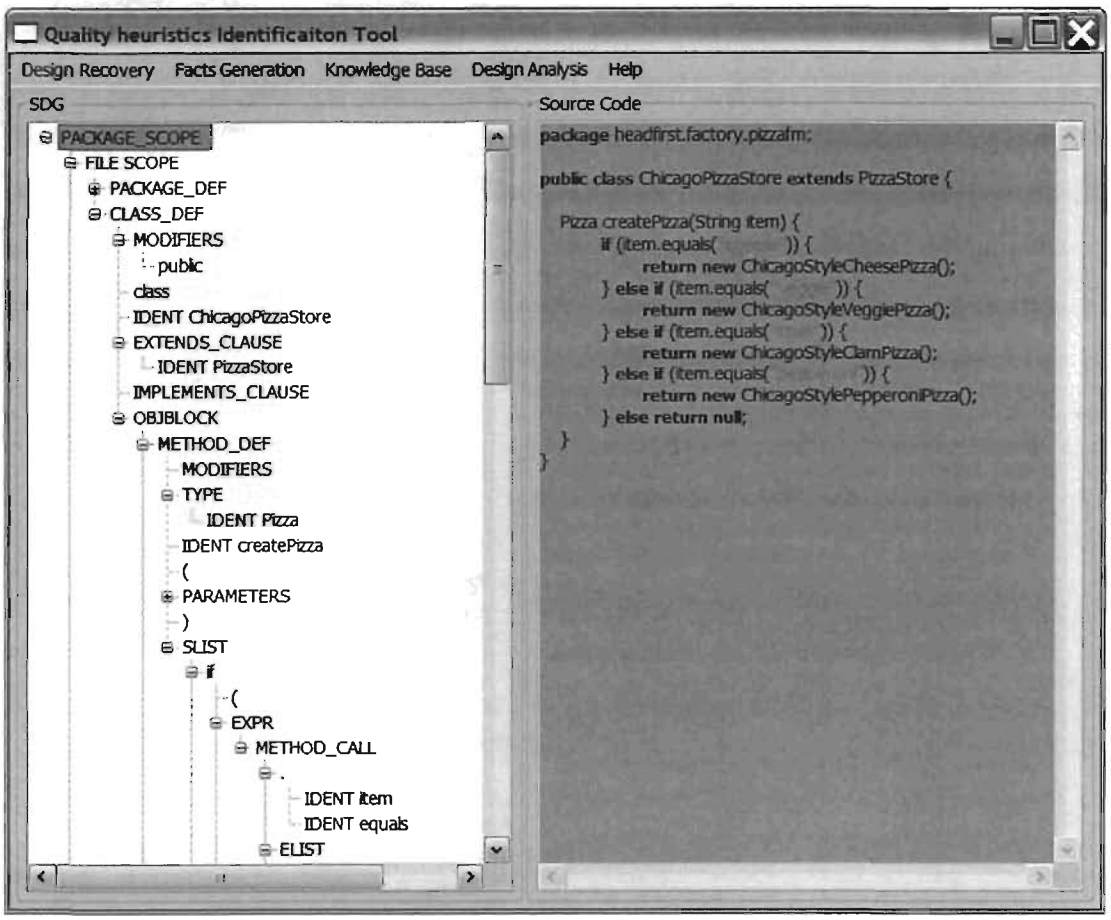


Figure 17 the SDG Graph Viewer

expand or collapse subtrees. In the right pane the corresponding source code file will be displayed; the view of the source code will be automatically synchronized to "File Scope" change in the left pane. "View SDG" allows the end user to view the previous generated SDG graph and its source code.

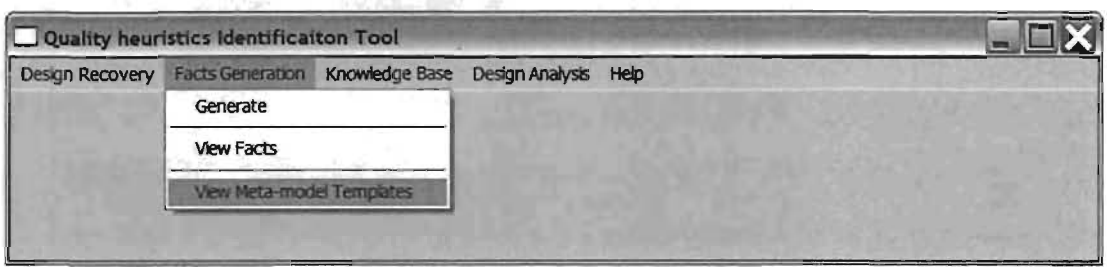


Figure 18 the Submenu of Facts Generation

The "Facts Generation" menu includes three submenus, which are: "Generate", "View Facts" and "View Meta-model Templates". The screen shot of the submenus

of "Facts Generation" is shown in Figure 18. "Generate" menu will input SDG graph in and

Figure 19 the Facts Viewer

produce facts in the form of XML. As shown in Figure 19, the facts viewer window shows the newly generated facts, the upper window shows an XML style of facts; the lower window shows the same facts but in plain text. We have explained in the previous section that types of all generated facts are defined in the meta-model; those types can be browsed in the meta-model template viewer window (in the current implementation, we do not allow the end user to modify the meta-model). As in the case of the facts viewer window, the facts viewer window has two windows which are XML and plain text forms of the meta-model templates. Figure 20 is the screen shot of the meta-model view window.

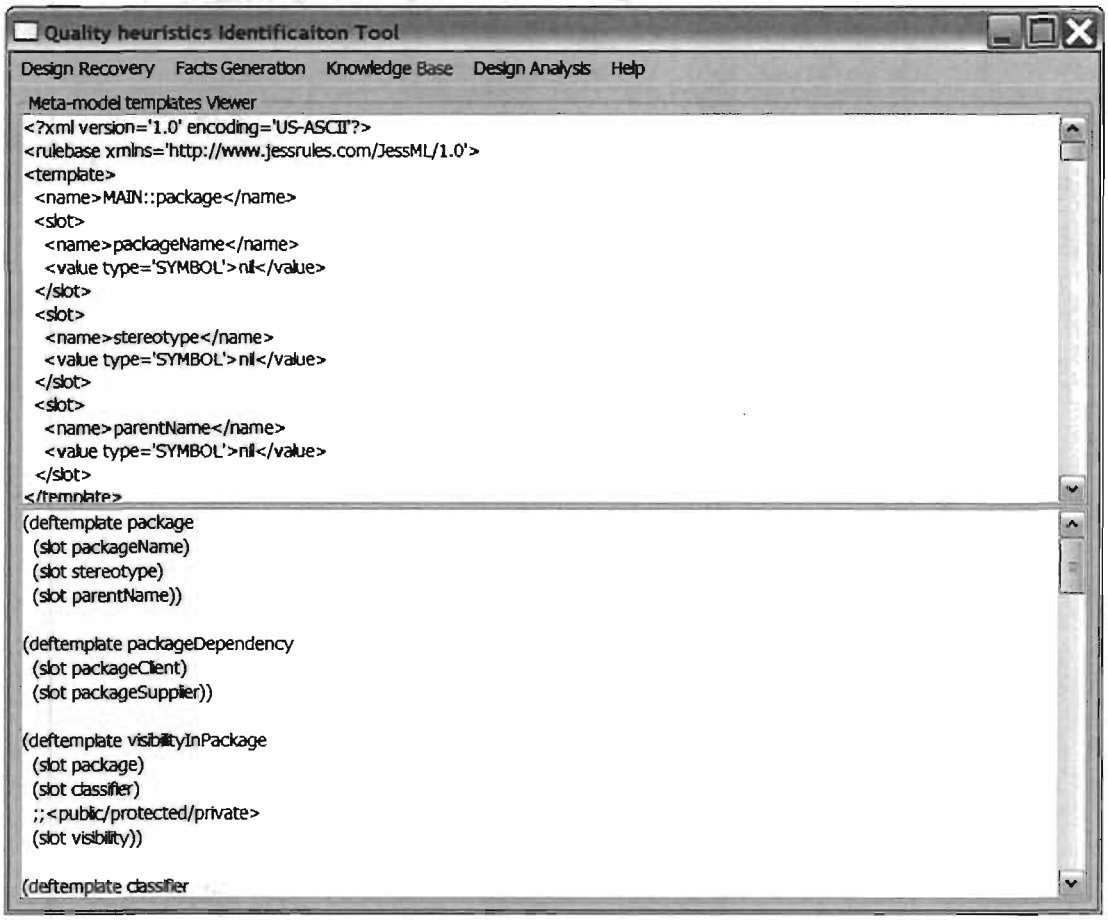


Figure 20 Meta-model templates viewer

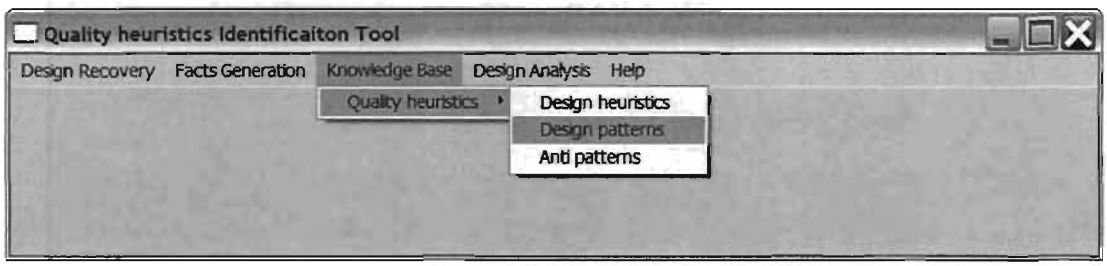


Figure 21 the submenu of Knowledge Base

Figure 20 shows the submenu of the Knowledge Base; our knowledge base consists of quality heuristics which in turn contains design heuristics, design patterns and anti-pattern rules. The end users can add, modify and delete those rules by using the Rule Editor window as shown in Figure 22 and Figure 23. The editor window is a sash container (one of the Eclipse SWT container widgets), which contains two windows: tree view and tab; the sash container makes it possible for the user to expand one window and reduce another one dynamically. The tree view at the left holds all the

quality heuristic rules in the hierarchy of Design heuristics, Design patterns, and Anti-patterns. Tabs in the right pane contain the contents of a quality heuristic and its production rule respectively; it also has several buttons to allow the end user to manipulate those quality heuristics.

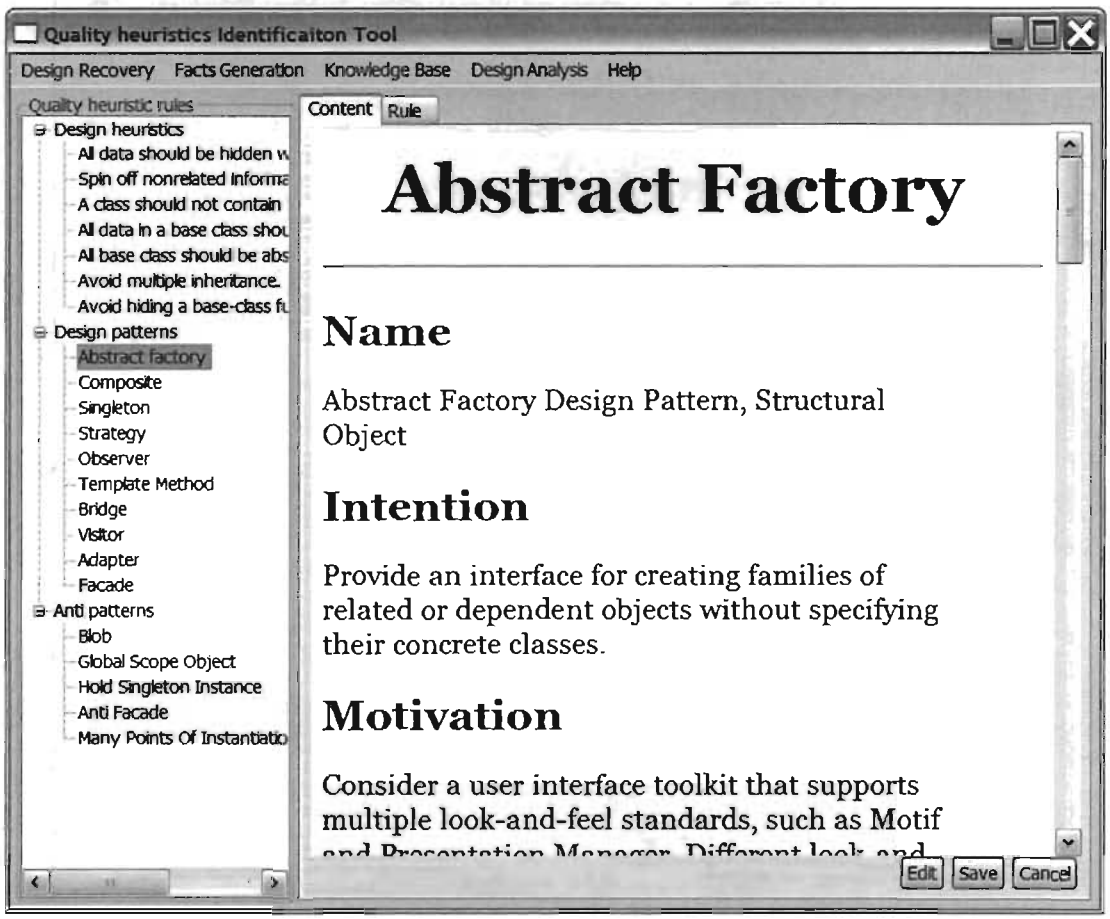


Figure 22 Quality heuristics viewer window

In the Figure 22, the **Content** tab shows the detail of a quality heuristic, such as name, description, diagrams, and implementation etc. which are explained in Chapter 3. The end user can view it like using a web browser. The **Edit** button allows the end user to modify the contents of quality heuristics, the **Save** and **Cancel** buttons allow the end user to keep their changes or not, respectively. The **Rule** Tab in Figure 23 shows the production rule associated with the selected quality heuristic.

It has the XML style and plain text style windows. The XML style window allows the end user to modify the rule, save and cancel their changes, but a plain text window

will allow the end user only to view the plain text style rule. Plain text style rule is actually

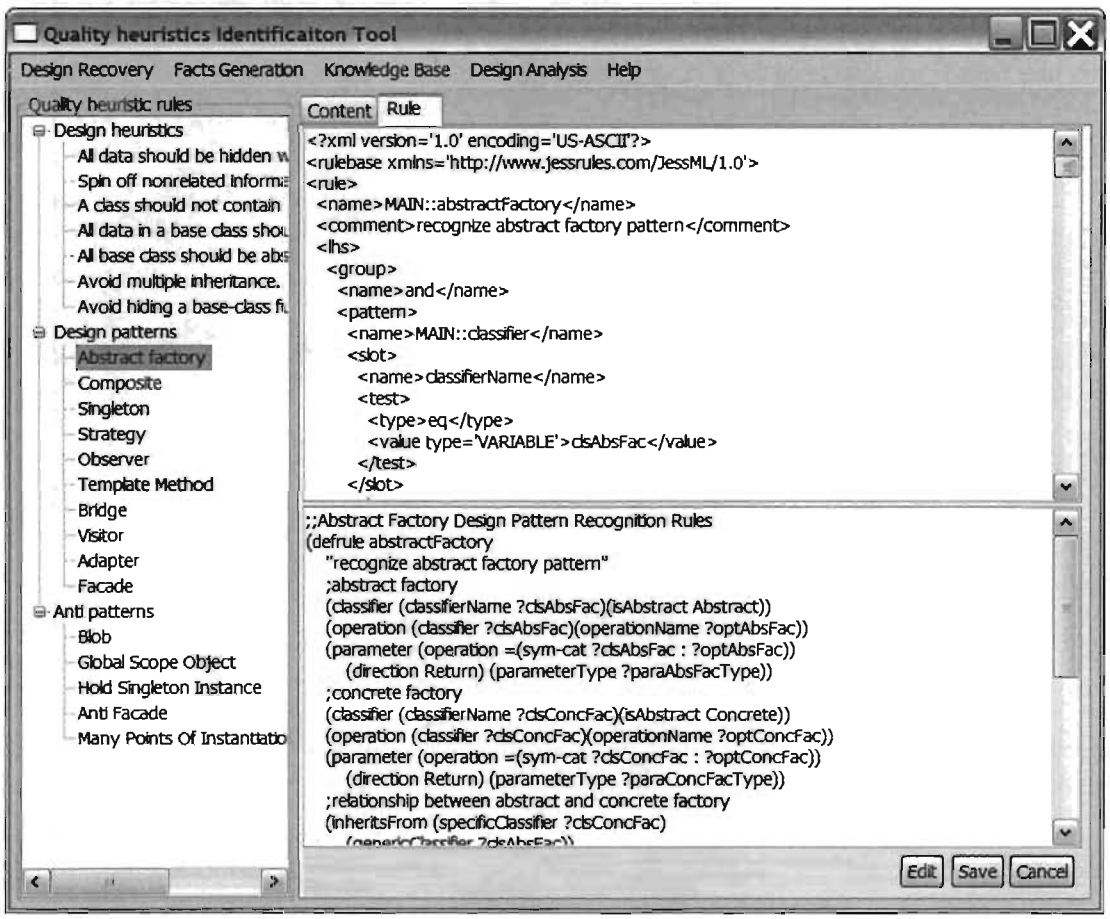


Figure 23 Rule Editor Window

in real Jess rule style; it is much simpler than the verbose XML style rule. The rationale is that we can verify XML style rule based on quality heuristic rule DTD.

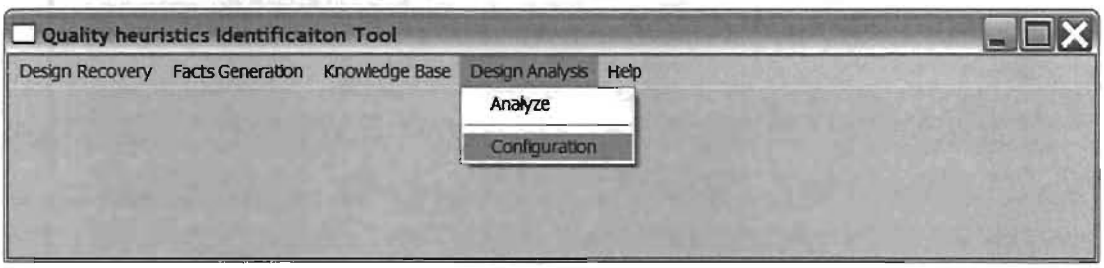


Figure 24 the submenu of Design Analysis

Finally, we introduce the submenu of design analysis; it includes Analyze and Configuration items as shown in Figure 24. "Analyze" will apply all the quality

heuristics in the knowledge base by default to the target system and report the found design patterns and problematic constructs to the end user. If the end user does not want to apply all the quality heuristics, the "Configuration" menu allows the end user to select whatever desired at a fine grain level. As shown in Figure 25, the "Analysis Configuration" window supplies a quality heuristics tree with a check box for each node, which allows the end user to freely compose the selection of the quality heuristics of interest.

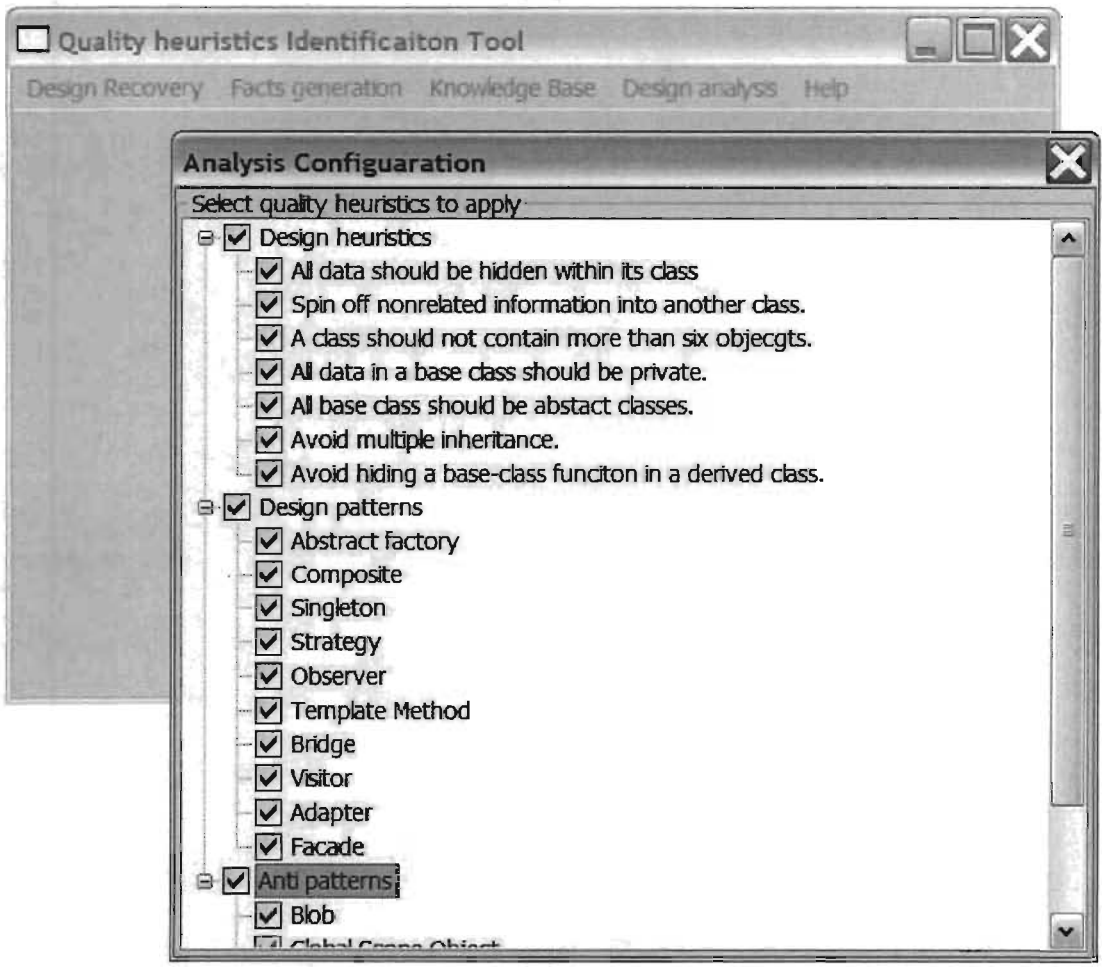


Figure 25 Analysis Configuration

Figure 26 shows an example of analysis results; that is, an abstract factory design pattern is found. Figure 26 also shows all classes that participate in the abstract factory design pattern and their file names and locations.

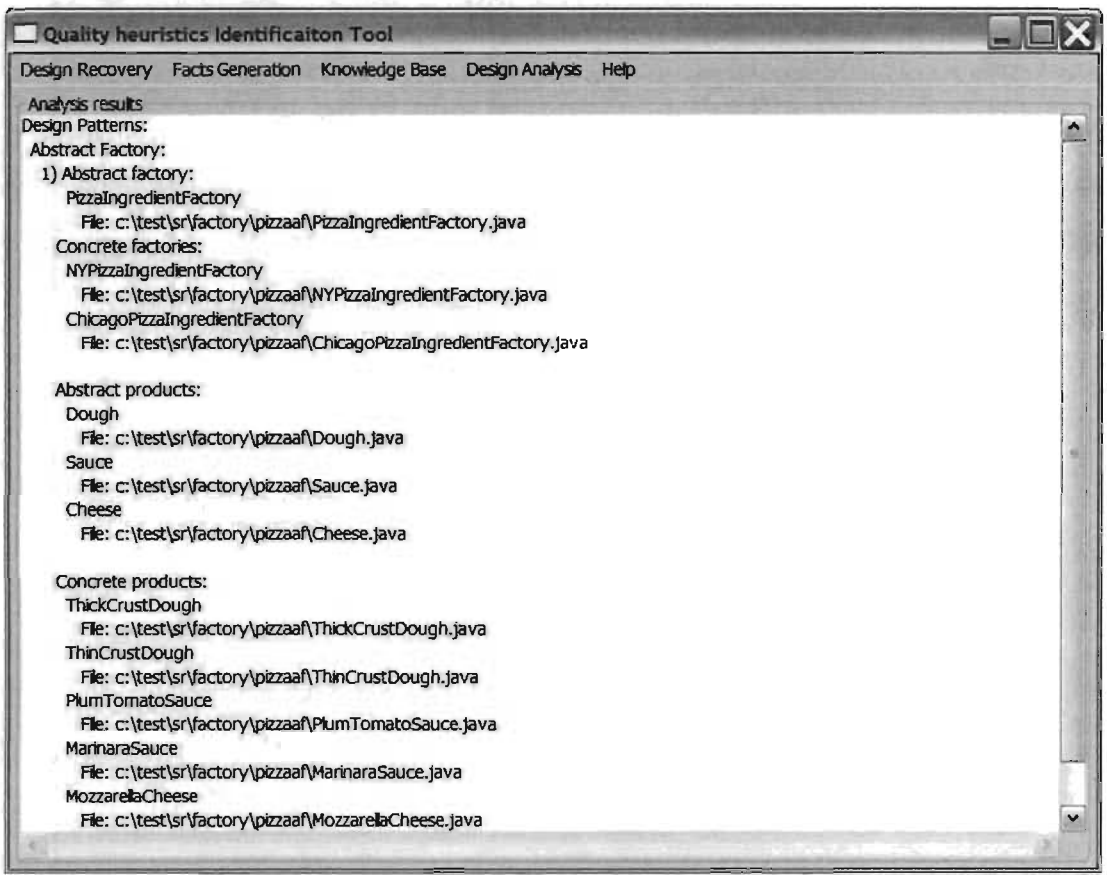


Figure 26 Analysis Results

5.3 Implementation Issues

A number of issues must be considered in the implementation of the prototype tool. They are briefly discussed below.

- Jess Performance

  Jess uses a special algorithm called Rete to match the rules to the facts. The Rete algorithm is implemented by building a network of interconnected nodes. Every node represents one or more tests found on the LHS of a rule. Each node has one or two inputs and any number of outputs. Facts that are being added to or removed from the working memory are processed by this network of nodes. Thus, a Rete network ultimately determines the Jess performance. The performance of a Rete-based system depends not so much on the number of rules and facts but on the number of partial matches generated by the rules. The classic example of writing efficient rules looks like this: this rule

```
(defrule match-1
  (item ?x)
  (item ?y)
  (item ?z)
  (item ?w)
  (find-match ?x ?y ?z ?w)
  => )
```

will consume lots memory (Jess will throw the OutofMemory exception in the worst case scenario) and run a long time to generate a result, because it must form all possible permutations of 'item' facts before finding the one permutation that matches the 'find-match' fact. If there are 10 'item' facts, this is 10x10x10x10 = 10,000 partial matches that are sent to the last join node. If the rule is rewritten like this:

```
(defrule match-2
  (find-match ?x ?y ?z ?w)
  (item ?x)
  (item ?y)
  (item ?z)
  (item ?w)
  => )
```

Then there is one and only one partial match sent to the last join node; actually only one is sent to each join node. Whereas the first rule might take several minutes to generate a result on a slow machine, the second rule generates the result instantaneously. Bearing in mind that patterns that match fewer facts should be put toward the beginning of a rule, building a rule in this way will reduce the number of partial matches and limit memory consumption.

- Improvement of the parser

  The parser provides a crucial functionality in the design extraction module. Although the current parser implementation works well, it can be extended or improved in several places. First, the current parser implements a Java 1.3 grammar compliant, not Java 1.5 grammar. That means it could not parse generic types, enumerated types, and annotation etc. Second, the parser can be modified to use Java's Class.forName method to read class information for classes whose source is unavailable but in the class path. For example, if your source code references java.awt.Color, use Class.forName("java.awt.Color") to read information about the Color class. A new Definition subclass can be

defined that adapts a java.lang.Class object to a ClassDef object, allowing the parser to reference compiled classes as easily as source.

6 Evaluation

6.1 Evaluation Procedure

It is necessary to use real-world examples of design heuristics, design patterns, and anti-patterns to test our prototype tool's ability of detecting what quality heuristics are specified. To this end, it is necessary to obtain:

- Java classes that implement design heuristics, design patterns and anti-patterns from existing external source code;
- Java classes that do not implement any design heuristics, design patterns and anti-patterns.

The most obvious source of design heuristics, design patterns and anti-patterns would be to choose some of the 'standard texts' of design heuristics and patterns, such as [Brown et al., 1998], [Gamma et al., 1995] and [Riel, 1996]. However, most of these books were written using Smalltalk and C++ as their examples, and do not show any Java examples since they were written prior to the development of Java. However, subsequent books have been written specifically for the Java platform, such as Head First Design Pattern [Eric and Elisabeth, 2004] that was published by O'Reilly and Applied Java Patterns [Stelting and Maassen, 2001] that was developed by Sun Microsystems, specifically to provide Java examples of design patterns. As a result, this book provided one of the key sets of examples for testing our prototype tool.

It is also important that our prototype tool be capable of detecting constructions according to quality heuristics from real-world examples, and be extensible to allow new quality heuristics to be defined in the future. There are many open-source or source-available projects that are written in Java and could be used. We select ArgoUML, Azureus, J2SE (Java 2 Platform, Standard Edition 6 Development Kit) [J2SE, 2005], and JHotDraw.

Evaluation was performed on the following sources:

1. **AJP** Applied Java Patterns [Stelting and Maassen, 2001].

2. **ArgoUML** [ArgoUML, 2007] is the leading open source UML modeling tool and includes support for all standard UML 1.4 diagrams.
3. **Azureus** [Azureus, 2007] is a Java-based BitTorrent client, with support for I2P and Tor anonymous communication protocols. Azureus allows users to download multiple files in a single graphical user interface (GUI).
4. **JHotDraw** [JHotDraw, 2007] is a two-dimensional graphics framework for structured drawing editors that is written in Java. It is based on Erich Gamma's JHotDraw.
5. **J2SE** Java 2 Platform [J2SE, 2005][4], Standard Edition 6 Development Kit, a development environment for building applications, applets, and components using the Java programming language.

These projects have been selected because

- They rely heavily on some well-known design patterns serving perfectly the aim of evaluating a design pattern detection algorithm.
- The authors explicitly indicate the implemented design patterns in the documentation and in this way it was possible to evaluate the results of the proposed methodology.
- They are all open-source projects with their source code publicly available.
- They vary in size enabling the scalability test of the proposed methodology.

6.1.1 Example Selection

To verify that quality heuristics were detected correctly, it was necessary to search the sources for design heuristics, design patterns and anti-patterns that were either explicitly or implicitly documented, or were considered by other developers to be a clear example of a pattern. The AJP and HFDP books provided a list of examples of Java patterns along with a description of the pattern itself, which therefore immediately provided a source that could be used to test our prototype tool.

[4] Due to the limitation of our Java parser, we have to manually modify J2SE's source code in order to parse them correctly.

To find a set of patterns from ArgoUML, JHotdraw and J2SE libraries, a manual search of the source code and its document was performed. Some patterns (such as Abstract Factory and Observer) were obviously implemented in classes such as java.awt.Toolkit and java.util.Observer; J2SE and ArgoUML, etc.; document comments and class names gave sufficient clues to be able to deduce this. However, for source files without such obviously identifying marks, it was necessary to make a judgement about whether individual classes implement a pattern or not.

In particular, design heuristics are more explicit and less documented than design patterns and anti-patterns; we have to create some examples by modifying implementations of selected examples.

6.1.2 Non-example Selection

To ensure that our prototype tool was not reporting design heuristics, design patterns and anti-patterns where none existed, it was also run against a selection of other classes. For design patterns, non-examples were created by "breaking" implementations of design patterns from [Eric and Elisabeth, 2004] and [Stelting and Maassen, 2001], by removing methods or fields that played a part in the pattern. For design heuristics and design patterns, non-examples were created by modifying implementations of selected examples.

6.1.3 Evaluation Results

The classification of the results has been performed by manually inspecting the source code and referring to the internal and external documentation of the projects. The results shown in Table 1 and Table 2 are based on the categories where quality heuristics are refined from. The results are broken down into the quality heuristics name, testing component. Table 1 shows how many instances are detected. Table 2 shows the analysis results by comparing the tool-generated results with the manual code inspection results. There are four possible outcomes in Table 2:

- **True positive** (√+): the prototype tool detected an instance of the quality heuristics, and the instance exists in the target system.

- **True negative** (√-): the prototype tool did not detect an instance of the quality heuristics, and there is no instance in the target system.
- **False positive** (×+): the prototype tool detected an instance of the quality heuristics, but no instance exists in the target system or there is a kind of ambiguous results.
- **False negative** (×-): the prototype tool did not detect any instance of the quality heuristics, but an instance exists in the target system.

A false negative (×-) result and a false positive result (×+) are not a success. A false negative result is harder to verify than a false positive result because we must have knowledge of the occurrences of the quality heuristics in the target system through manual inspection of the source code. It is a time-consuming task. There are 19 quality heuristic rules implemented in our prototype tool: 8 design heuristics rules, 8 design pattern rules from creational, structural and behavior categories and 3 anti-pattern rules.

Quality Heuristics		AJP	ArgoUML	Azureus	JHotDraw	J2SE
Parsed classes			898	2884	554	1375
Design Heuristics	BaseClassShouldBeAbstract	1	10	42	4	31
	CommonPrivateFunction[5]	1	870	883	474	542
	ClassDataShouldBeHidden	1	290	664	39	385
	ClassDataShouldBePrivate	1	23	144	8	67
	BaseClassKnowDerive	1	8	18	1	36
	LawOfDemeter[6]	1	483	4540	350	244
	MoreThanSixObjects	1	67	214	9	56
	UnusedAttribute	1	92	365	10	203
	Abstract Factory	1		124	16	18
	Singleton	1	3	16	1	14
	Adapter	1	42	642	28	57

[5] The value unit of CommonPrivateFunction in the Table 1 is function not class.
[6] The value unit of LawOfDemeter in the Table 1 is function not class. The unit of the rest values in the Table 1 is class.

Quality Heuristics		AJP	ArgoUML	Azureus	JHotDraw	J2SE
Design Pattern	Bridge	1	0	11	4	14
	Decorator	1	3	6	2	6
	Visitor	1	0	0	0	0
	Observer	1	2	65	3	22
	Strategy/State	1	42	439	39	49
Anti Pattern	Anti-Singleton	1	0	7	2	15
	God Class	1	15	63	2	26
	Global Scope Object	1	628	404	30	578

Table 1 Quality Heuristics Detection Results

To save space, some quality heuristics appearing in Table 1, Table 2, Table 3 and Table 4 are given as an abbreviated name. Their full names, detailed descriptions and rationale are listed here.

Quality Heuristics		AJP	ArgoUML	Azureus	JHotDraw	J2SE
Design Heuristics	BaseClassShouldBeAbstract	√+	√+	√+	√+	√+
	CommonPrivateFunction	√+	×+	×+	×+	×+
	ClassDataShouldBeHidden	√+	√+	√+	√+	√+
	ClassDataShouldBePrivate	√+	√+	√+	√+	√+
	BaseClassKnowsDerive	√+	√+	√+	√+	√+
	LawOfDemeter	√+	√+	√+	√+	√+
	MoreThanSixObjects	√+	√+	√+	√+	√+
	UnusedAttribute	√+	√+	√+	√+	√+
Design Pattern	Abstract Factory	√+	√+	√+	√+	√+
	Singleton	√+	√+	√+	√+	√+
	Adapter	√+	√+	×+	×+	√+
	Bridge	√+	√+	√+	√+	√+
	Decorator	√+	√+	√+	√+	√+

	Visitor	√+	√+	√-	√-	√-
	Observer	√+	×-	×-	×-	×-
	Strategy/State	√+	×+	×+	×+	×+
Anti Pattern	Anti-Singleton	√+	√+	√+	√+	√+
	God Class	√+	√+	√+	√+	√+
	Global Scope Object	√+	√+	√+	√+	√+

Table 2 Validation of the Detection Results

1. BaseClassShouldBeAbstract [Riel, 1996]

   Name: All base classes should be abstract classes.

   Rationale: Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class. Concrete things change a lot, abstract things change much less frequently. Moreover, abstractions are "hinge points"; they represent the places where the design can bend or be extended, without being modified.

2. CommonPrivateFunction [Riel, 1996]

   Name: Common-code private functions should be hidden.

   Rationale: This heuristic is designed to reduce the complexity of the class interface for its users. The basic idea is that users of a class do not want to see members of the public interface that they are not supposed to use. These items belong in the private section of the class. A common-code private function is created when two methods of a class have a sequence of code in common. It is usually convenient to encapsulate this common code in its own method. This method is not a new operation; it is simply an implementation detail of two operations of the class. Since it is an implementation detail, it should be placed in the private section of the class, not in the public section.

3. ClassDataShouldBeHidden [Riel, 1996]

   Name: All data should be hidden within its class.

   Rationale: The violation of this heuristic will make the source code harder to maintain. The consistent enforcement of information hiding at the design and implementation level is responsible for a large part of the benefits of the object-

oriented paradigm. If data is made public, it becomes difficult to determine which portion of the system's functionality is dependent upon that data.

4. ClassDataShouldBePrivate [Riel, 1996]

   Name: All data in a base class should be private. Do not use protected data.

   Rationale: It is similar to using public data which will be a weakening of data hiding, and it should be avoided.

5. BaseClassKnowsDerive [Riel, 1996]

   Name: Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

   Rationale: If base classes have knowledge of their derived classes, then it is implied that if a new derived class is added to a base class, the code of the base class will need modification. This is an undesirable dependency between the abstractions captured in the base and derived classes.

6. LawDemeter

   Name: Law of Demeter. The methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only.

   Rationale: Avoid traversing multiple links or methods. A method should have limited knowledge of an object model. A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but it should not traverse a second link from the neighbor to a third class.

7. MoreThanSixObjects [Riel, 1996]

   Name: A class should not contain more than six objects.

   Rationale: Most of the methods defined on a class should use most of the data members most of the time. Assuming this is true; implementors of a method will need to think about all of the data members while writing the method. If the developer cannot keep all of the data in his or her short-term memory, then items will be omitted and bugs will creep into the code.

8. UnusedAttribute:

   Name: avoid unused attributes of a class.

Rationale: Object-oriented applications may contain data members that can be removed from the application without affecting the program behaviour. Such "dead" data members may occur due to unused functionality in class libraries or due to the programmer losing track of member usage as the application changes over time. It will lead to the waste of memory resources and will downgrade applications' maintainability.

9. God Class [Riel, 1996]

   Name: also called Blob.

   Rationale: The behavioral form of the god class problem is caused by a common error among action-oriented developers in the process of moving to the object-oriented paradigm. These developers attempt to capture the central control mechanism so prevalent in the action-oriented paradigm within their object-oriented design. The result is the creation of a god object that performs most of the work, leaving minor details to a collection of trivial classes. Implementation code becomes complex and unstructured. Future changes are hard to accommodate.

10. Anti-Singleton

    Name: Do not use a global variable

    Rationale: A global variable can potentially be modified from anywhere, and any part of the program may depend on it. A global variable therefore has an unlimited potential for creating mutual dependencies and for adding mutual dependencies increases complexity. We should use a Singleton pattern instead of using a global variable. At the same time when a Singleton pattern is used in a design, no other class objects should keep a reference to the singleton class object. Otherwise it introduces global variable again.

Analysis of the results is in the following section.

6.2 Results Analysis

The results show that our prototype tool is capable of recognizing a number of quality heuristics implemented not only in small code examples, but also in real Java systems. The results also reveal the lack of implementations of some quality

heuristics in some systems; indeed, the reason why [Stelting and Maassen, 2001] and [Eric and Elisabeth, 2004] cover them all is due to the fact that it is a superset of design patterns defined by [Gamma et al., 1995]. Thus, not all quality heuristics turn up in certain large systems; rather, some (like Singleton) occur frequently, whereas others (like Visitor) occur infrequently.

To some extent this discrepancy can be related to the size of the quality heuristic. Smaller single class quality heuristics (such as Template Method, DataShouldBeHidden and Singleton) are relatively common, as opposed to larger multi-class quality heuristics (such as Visitor, CommonPrivateFunction and Bridge), which tend to be used for specific cases where it is necessary to link together many classes. They occur less frequently.

Also, multi-class quality heuristics are usually more specialized (the Visitor pattern is used for traversing ASTs inside compilers, for example) and so are likely to be used in fewer situations.

More examples were found in [Stelting and Maassen, 2001] and [Eric and Elisabeth, 2004] since both of them were books specifically aiming at educating the user towards using design patterns in Java. Their pattern implementations were successfully detected in our prototype tool, except State and Strategy patterns which are detected; but our prototype tool could not distinguish them because quality heuristic rules of State and Strategy are ambiguous in terms of rule definitions.

Generally speaking, there are some features of quality heuristics that our prototype tool can easily detect:

- Small quality heuristics consisting of one or a small number of classes (the problem becomes more difficult as the number of classes increases)
- Quality heuristics that are defined by their structure or relationships.

6.2.1 Positive results

In addition to AJP and HFDP test cases, we have found many design pattern instances in J2SE framework that comply with its internal and external documentations. For example, Decorator pattern was detected by our prototype tool. Here is a set of objects that use decorators to add functionality to reading data from a file: LineNumberInputStream is a concrete decorator. It adds the ability to count the line numbers as it reads data. BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, one line at a time. FileInputStream is the component that is being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes. BufferedInputStream and LineNumberInputStream both extend FilterInputStream, which acts as the abstract decorator class.

Examining the results shown in Table 1, we know that the violations of design heuristics and anti-pattern quality heuristics are detected in ArgoUML and J2SE frameworks. J2SE, especially, is a well-designed and developed framework, which has been evolving from Version 1.0 to Version 1.6 in a decade. Why does such a framework have so many flaws?

First, as explained in Section 3.1, the design heuristics are not law; they are not written as hard and fast rules; they are meant to serve as warning mechanisms that allow the flexibility of ignoring the heuristic as necessary.

Second, design heuristics include a lot of contradicting design heuristics. The contradicting design heuristics are derived from different opinions about good OO. They pursue certain design goals that can conflict with each other, e.g., with the design goals simplicity and adaptability. So they will be used in a specific situation. For example, for the correct use of inheritance, one heuristic demands inheritance hierarchies to be deep and narrow in theory while another one demands that they not

be too deep in practice. When faced with contradicting heuristics, the developer should examine the design further to determine whether or not both of them are applicable in their particular situation, and if they are, to decide which one "plays the more important role"

For those two reasons, we can not say that J2SE or ArgoUML has many flaws.

On the other hand, anti-patterns are more precise than design heuristics rules in term of working as a quality indicator. If traces of anti-patterns occurrences are found, it will indicate the system design does indeed suffer from flaws. After inspecting java.awt.Toolkit source code, we found that Toolkit class has 93 functions and 13 attributes, which are much higher than the threshold value of 60 (the sum of both functions and attributes) in our Blob quality heuristic definition. At least from a maintenance point of view, java.awt.Toolkit is not easy to maintain.

6.2.2 Negative results

There are two kind of negative results in the Table 2. One is the ambiguous results, and the other is the failed results. We first talk about the cause of ambiguous results, and afterward state the rationale for the failed results.

6.2.2.1 Ambiguous Results

A key reason for using a design pattern is that it helps describe the system, as well as implement it. Thus, when a pattern is used (and documented) in a code base, it aids other developers looking to extend the system. Many patterns have a high-level intent in the way in which they are applied; patterns such as Command, for example, have a very light structure but the intent of the pattern is clearly visible. Similarly, patterns such as Visitor have a great deal of intent; and it is this intent that sets it apart from a class hierarchy with a number of methods.

From both an implementation and a structural point of view, the State pattern depicted in Figure 28 and the Strategy patterns shown in Figure 27 can look very

similar. Both of them have a class (Context) that defines the interface of interest to clients and maintains an instance of a Concrete subclass that inherits from a class that defines the current state or algorithm.

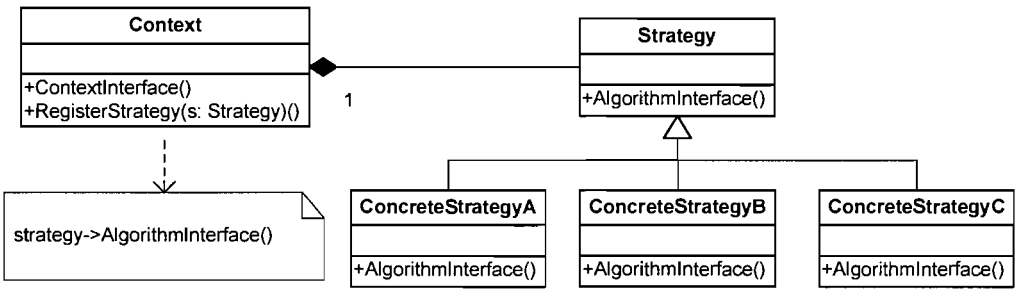


Figure 27 Strategy Design Pattern

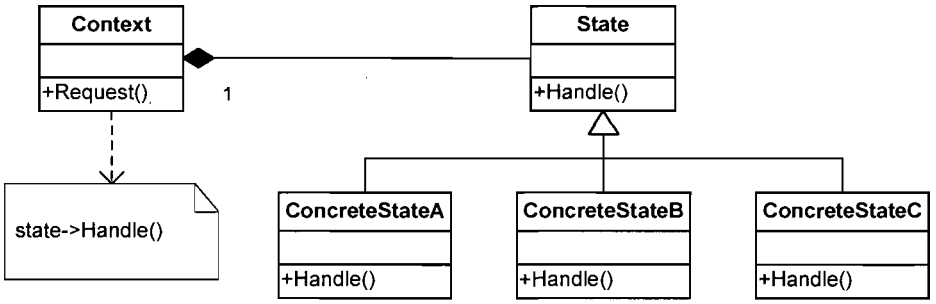


Figure 28 State Design Pattern

However, their intent is very different; the intent of the Strategy pattern is to enable several related algorithms to be encapsulated into their own respective classes, so that a client can be dynamically configured with an object of one of these classes. Whereas the intent of the State pattern is to enable an object to undergo a qualitative change in behavior when its internal state changes. Rather than an expressing this as extensive and similar case analysis in each method, this pattern defines a class that to represent each possible state the object may be in. There is no clear-cut distinction between a simple state and an algorithm; it depends on where (and how) they are used.

It would therefore be difficult to create a quality heuristic definition that would capture a Strategy but would not capture a State, and vice versa. It would also be difficult to construct the quality heuristic definition in a way that would not also

admit many ambiguous, which would devalue the benefit of such a quality heuristic definition.

We obtained another ambiguous result from J2SE's java.awt.Component test case; our prototype tool correctly recognized a Bridge pattern, but it also reported a State/Strategy pattern for the same structure. If we compare the Bridge design pattern in Figure 29 with the State in Figure 27 and the Strategy design patterns in Figure 28, we can figure out that the structures of a Bridge design pattern are the same as a State/Strategy design pattern except for the RefinedAbstraction subclass. We did not find any indication that the Bridge design pattern uses the Strategy design pattern from [Gamma et al., 1995]; thus we put it into the ambiguous category.

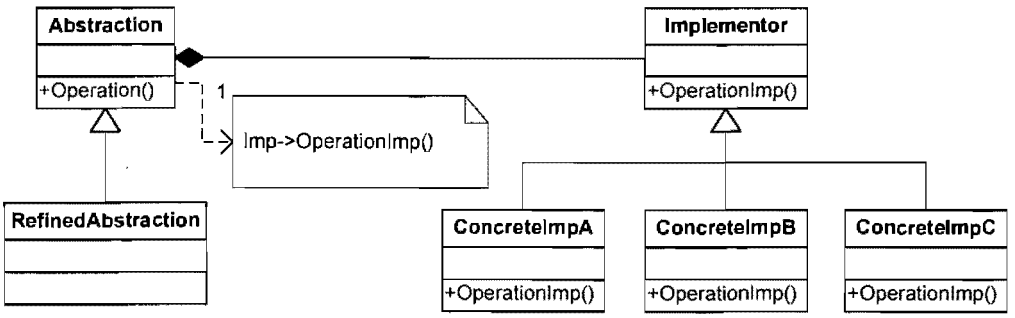


Figure 29 Bridge Design Pattern

To distinguish a State from a Strategy, the quality heuristic definition has to know whether the abstract class represents a state or an algorithm. There is a recent approach that attempts to distinguish State and Strategy employing the new syntax elements of UML 2.0 for sequence diagrams; we are going to adapt the methodology explained in [Wendehals, 2004] in our future version prototype tool.

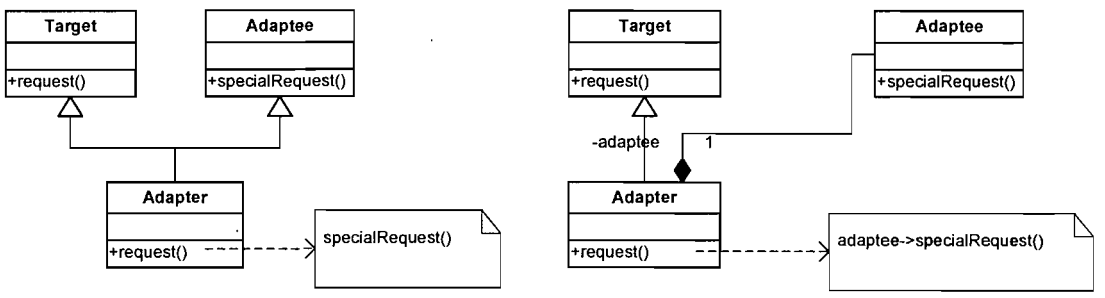
6.2.2.2 Failed Results

The quality heuristic definitions (shown in Appendix D) of the Observer pattern and Composite pattern defines a relationship between the observer/composite (container) class and the generic type of data to be added. It also adds the requirement that it must be possible to navigate from the container class to its child components; and that

there must be a way of adding (and removing) components from the container classes.

In line with standard Java practice, the methods for adding and removing items from the container should be prefixed "add" and "remove" respectively, to fit in with the JavaBeans naming conventions. Of course, this introduces a possible source for failed results, since other implementations may choose to avoid this standard naming convention. Our prototype tool did not recognize the Observer pattern implemented in our HFDP [Eric and Elisabeth, 2004] test case – Weather station, because the prefixes are "register" and "unregister" instead of "add" and "remove".

There are several possible causes of the failed results although we did not encounter them during our prototype tool evaluation. The first possible scenario is quality heuristic implementation variants. Quality heuristics are defined by static structures and their associated constraints. In the real world, static structures and constraints may have diverse implementations. If the quality heuristics do not encompass all of those diversities, those are missing will not be detected by the quality heuristic rule. Thus failed results are introduced. For example, the Adapter design pattern mainly has two different implementations; one of them is called the Class Adapter which uses multiple inheritances to adapt one interface to another; another one is called the Object Adapter, which relies on object composition. Both of Class and Object Adapters are shown in Figure 30. Likewise, the Singleton design pattern has a classic implementation named PublicSingleton along with PrivateSingleton and LazySingleton variants.



A) Class Adapter B) Object Adapter

Figure 30 Adapter implementation variants

The second cause is that the design-recovery module did not supply enough information for the rule engine to deduce the results. Losing information may be caused by parsing source code errors, having only binary code instead of source code whilst our prototype can not discover class information in bytecodes or if there is no source code and binary code at all. Losing information will break the quality heuristic rules to function as what they expected. For instance, the Anti Common-code Private Function is used to detect the definition of methods in the public interface of a class that are used only as auxiliary methods for the implementation of other methods of this class. This contradicts the design heuristic, "Do not put implementation details such as common-code private functions into the public interface of a class" [Riel, 1996]. To get a more real-world feel for common-code private functions, consider the class X to be a linked list as Figure 31 shows, `f1` and `f2` to be the functions `insert` and `remove`, and the common-code private function `fpub` to be the operation for finding the location in the linked list for an insertion or removal. As Figure 31 depicts, the fpub function appeared in public section and there is no any external function invocation; fpub is violated with Common-code Private Function rule; our Anti Common-code Private Function anti-pattern rule will pick it up immediately. If external function invocations exist in another file, and the system only has its binary code (.class or .jar) or if, even worse, those source codes are missing, Anti Common-code Private Function quality heuristic rule will report it. Thus it is a fail result.

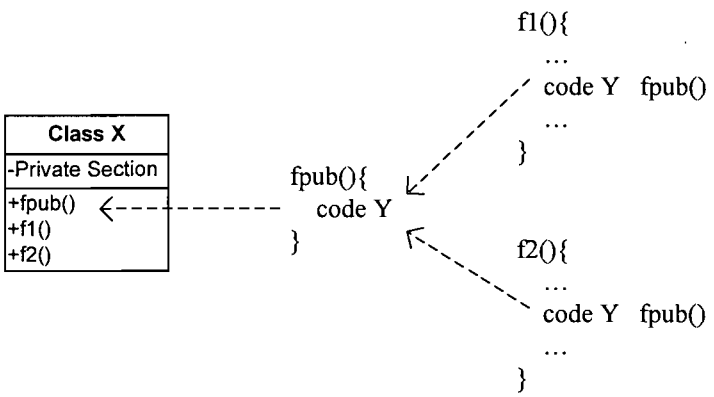


Figure 31 Anti Common-code Private Function

Actually we can improve the design-recovery module to employ a Java bytecode manipulation framework that provides detailed information concerning the static structure of the target system. Thus we can partially fix two aforementioned causes of the failed results.

In the current prototype tool, it should be noted that the applied methodology detects only quality heuristics in which all roles corresponded to classes within the system boundary. As a result, pattern instances involving classes that do not belong to the system (e.g., classes in Java or external APIs) have not been considered.

6.3 Case Study

In this section, we present a case study to which we have applied our approach and prototype tool. The aim of this case study is to demonstrate the applicability and usability of our prototype tool. We select open source project - ArgoUML [ArgoUML, 2007] as the test case. Its first version emerged in 2002, and it has evolved from Version 0.1 to current Version 0.24. ArgoUML has a total of 8 versions now. We want to test whether the quality heuristics implemented in the prototype tool are good indicators of software quality. Do software engineers bear those quality heuristics in mind when they are developing and maintaining software? What kind of benefits will software engineers get by using the prototype tool.

Table 3 shows the quality heuristic results of 8 ArgoUML versions (from 0.10 to 0.24). It also shows how many parsed classes there are in each version. In order to better understand the results, we have calculated the Pearson correlation coefficient on two data sets which are 0.1 to 0.16 and 0.18 to 0.24; the results are shown in Table 4.

Quality Heuristics		**0.10**	**0.12**	**0.14**	**0.16**	**0.18**	**0.20**	**0.22**	**0.24**
Parsed classes		898	984	1288	1269	1295	1417	1467	1524
	BaseClassShouldBeAbstract	10	11	11	12	9	8	8	8
	CommonPrivateFunction	870	941	886	908	847	893	768	748

Quality Heuristics		0.10	0.12	0.14	0.16	0.18	0.20	0.22	0.24
Design Heuristic	ClassDataShouldBeHidden	290	296	381	376	112	111	105	103
	ClassDataShouldBePrivate	23	25	28	28	14	15	14	14
	BaseClassKnowsDerive	8	8	4	3	1	1	0	0
	LawOfDemeter	481	481	535	539	544	624	731	685
	MoreThanSixObjects	67	74	86	84	68	73	70	72
	UnusedAttribute	92	111	265	297	243	250	246	252
Anti Pattern	Anti-Singleton	0	0	0	2	4	1	1	1
	God Class	15	16	21	22	17	20	15	19
	Global Scope Object	628	653	769	885	172	202	198	204
Design Pattern	Abstract Factory	0	2	3	4	3	4	5	8
	Singleton	3	27	36	41	24	21	17	17
	Adapter	42	47	41	46	21	30	38	35
	Bridge	0	0	0	0	0	1	1	0
	Decorator	3	3	2	2	2	2	2	2
	Visitor	0	0	0	0	0	0	0	0
	Observer	2	2	2	3	3	4	4	4
	Strategy/State	42	21	19	19	17	22	22	23

Table 3 Case Study Results

Table 3 shows the absolute values of the quality heuristic results. While Column 4 and 5 of Table 4 illustrate relative values of the quality heuristic results, whose value is the total number of each quality heuristic divided by the total of parsed classes for each version.

Quality Heuristics		0.10 - 0.16	0.18 - 0.24	v0.10 (%)	v0.24 (%)
Parsed classes		4439	5703	898	1524
	BaseClassShouldBeAbstract	0.764389	-0.89389	1.11	0.52
	CommonPrivateFunction	-0.00135	-0.67471	96.88	49.08

Quality Heuristics		0.10 - 0.16	0.18 - 0.24	v0.10 (%)	v0.24 (%)
Design Heuristic	ClassDataShouldBeHidden	0.991742	-0.89932	32.29	6.76
	ClassDataShouldBePrivate	0.98622	-0.05982	2.56	0.92
	BaseClassKnowsDerive	-0.9654	-0.82594	0.89	0
	LawOfDemeter	0.993094	0.881718	48.88	44.95
	MoreThanSixObjects	0.987326	0.665595	7.46	4.72
	UnusedAttribute	0.981237	0.802827	10.24	16.54
Anti Pattern	Anti-Singleton		-0.89389	0	0.07
	God Class	0.986065	0.135277	1.67	1.25
	Global Scope Object	0.896428	0.908503	69.93	13.39
Design Pattern	Abstract Factory	0.836784	0.894043	0	0.52
	Singleton	0.889616	-0.95441	0.33	1.12
	Adapter	-0.09714	0.918153	4.68	2.30
	Bridge			0	0
	Decorator	-0.9834		0.33	0.13
	Visitor			0	0
	Observer	0.535802	0.893892	0.22	0.26
	Strategy/State	-0.76886	0.949249	4.68	1.51

Table 4 Case Study Results Analysis

In total, 19 quality heuristics are divided into two groups. One group is of poor design indicators, which includes design heuristics and anti-patterns rules. Another group is for reusable design indicators including design patterns rules. Let us look first at the poor design indicators group. The trend is that the numbers of detected violations decline as the version numbers increase. The numbers of violations of some quality heuristics, such as GlobalScopeObject, ClassDataShouldBeHidden, ClassDataShouldBePrivate, Base Class Knows Derive, drop significantly to almost half of the initial values of the older versions. For example, BaseClassKnowsDerive is a rather bad design because, if base classes have knowledge of their derived

classes, then it is implied that, if a new derived class is added to a base class, the code of the base class will need modification. This is an undesirable dependency between the abstractions captured in the base classes and the derived classes. Looking at Table 3, the value gets lower and lower, and the violation has been eliminated in the latest version. Although the numbers of detected violations of the rest of the quality heuristics show fewer changes or even no changes, the newer version has 200 more classes than the older version on average; the latest version has almost double the number of classes as the first version. If this factor is taken into account, the detected violations are declining relatively. This conclusion can be observed easily by comparing the values of Column 4 (version 0.1) and the values of Column 5 (version 2.4) in Table 4, the values of Column 4 and Column 5 represent a percentage of violations; the latest version (2.4) is much lower than the oldest version (0.1).

By examining Table 3, we realize that the violations in Version 0.18 and the newer versions have dropped dramatically. This is accordance with what ArgoUML documents claimed, "The release 0.24 was described as a bug fix release solving the most serious problems in version 0.22, for a total of 172 bug fixes". We can also prove our observation by analyzing the Pearson correlation coefficients in Table 4's Column 0.1-0.16 and Column 0.18-0.24. In probability theory and statistics, correlation, also called correlation coefficient, indicates the strength and direction of a linear relationship between two random variables. In general statistical usage, correlation refers to the departure of two variables from independence. In this broad sense there are several coefficients, measuring the degree of correlation, adapted to the nature of the data. A number of different coefficients are used for different situations. The best known is the Pearson correlation coefficient, which is obtained by dividing the covariance of the two variables by the product of their standard deviations. The correlation is 1 in the case of an increasing linear relationship, −1 in the case of a decreasing linear relationship, and some value in between in all other cases, indicating the degree of linear dependence between the variables. The closer the coefficient is to either −1 or 1, the stronger the correlation between the variables.

We select parsed classes as one variable and the number of violations as the other one to calculate Pearson correlation, so our Pearson correlation represents the linear dependence between the classes and the violations for the two data sets (one set is Version 0.1 to Version 0.16 and results are recorded in Column 0.1-0.16; another one is version 0.18 to 0.24 and the results are listed in column 0.18-0.24). Most of the Pearson correlations of Column 0.1-0.16 are close to 1, which means the violations increase along with the increase of the parsed classes when ArgoUML evolves from Version 0.1 to Version 0.16. While looking at Column 0.18-0.24, 5 of them are close to -1, meaning that the violations are decreasing as the classes in the project are increasing. 2 of them are close to 0, meaning there is no obvious correlation between violations and the project complexity. 3 of them are close to 1, but the values are lower than their counterparts in Column 0.1-0.16. It proves that the design was improved significantly from Version 0.18 on. We can conclude that the newer versions are more reusable and maintainable than the older versions. In one word, the quality of ArgoUML is improving along with the evolution of the versions.

We can also draw a conclusion from the results that software engineers did not pay enough attention to some of the quality heuristic rules, even if those quality heuristics are good quality indicators such as God Class, Law Of Demeter and Class has More Than Six Objects etc., because the numbers of detected violations are quite high in the newer versions. For example, there are 685 detected violations of The Law of Demeter in the latest version. The Law of Demeter can be succinctly summarized as, "Only talk to your immediate friends." The fundamental notion is that a given object should assume as little as possible about the structure or properties of anything else. The advantage of following the Law of Demeter is that the resulting software tends to be more maintainable and adaptable. Since objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers. Software engineers should work on those points of design where high values of violations are generated.

On the other hand, the numbers of detected reusable designs in the different versions of ArgoUML are smaller but relatively stable. There are several reasons for causing such lower values. First, a design pattern is domain oriented, such as Visitor – it is originally designed for graph node traversing. There was no any detection of Visitor pattern in all 8 versions of ArgoUML. Second, complicated design patterns that involve more classes are appearing less and less in the system. However, the main reason is that the intrinsic complexity of design patterns preventing software engineers from apply design pattern intensively. Although design patterns are popular, developers need to be really comfortable with many patterns and gain a good understanding of the design of the target system before taking advantage of their knowledge.

With the help of our prototype, software engineers can locate and fix design problems more quickly and more easily. That is because quality heuristics specify the problem precisely and provide developers with the guidance required to solve it. The prototype tool also shows design pattern occurrences in the target systems; it will help software engineers to comprehend the source code and quickly grasp the collaborations between various parts of the program. It will help them make program implementations more flexible and reusable.

7 Conclusion and Future work

7.1 Future work

We built a prototype tool that can be used for formalizing quality heuristics and identifying them on Java applications. At the same time, more work remains to be done. In particular, the prototype tool can use a fuzzy-like evaluation mechanism to recognize not only entire patterns but also incomplete instances. The main objective is to detect design constructions with structures similar to a particular pattern but with some sort of variation that makes it undetectable on a perfect matching algorithm using the Jess inference machine. To achieve this goal, we are using a fuzzy rule engine based on works such as [Wenzel, 2006]. Second, we have to extend our quality heuristic collection; the heuristics automated in our work are a small collection of those that exist in the literature. The most challenging part of our research was reviewing heuristics to determine whether or not they could be automated effectively. Due to the subjectivity and expressive nature in the description of a heuristic, it was difficult to precisely determine what the heuristic was involved. The extension of this collection would improve the level of information that we are able to provide developers, thereby, in turn, enabling our prototype to be more useful in practice. Eventually, we anticipate that our research approach will enable developers to propose, automate, and evaluate their own heuristics along side those that already exist.

7.2 Conclusion

In this thesis, we have demonstrated not only how to formalize quality heuristics refined from the literature and software engineers experiences but also detect constructs, which are in conformance with, or in violation of, quality heuristics, from legacy source codes automatically in a rule-based system.

The whole process includes capturing knowledge about good and bad OO constructions, generating a deductive database of quality heuristics that are refined from design heuristics, design patterns and anti-patterns, parsing source code and

generating an SDG graph carrying design information of the source code, traversing pre-generated a SDG graph and generating knowledge facts representing recovered design information based on predefined meta-model templates, and applying knowledge base on facts holding design information and report deduced results of good and problematic constructs.

We implemented a prototype tool that is based on this theory and demonstrated with case studies that the theory of quality heuristics formalization and identification can be efficiently applied to a Java application. By using JESS rules to define the constructions detected by our prototype tool, it is easier to expand the scope of detection. It is possible to add new heuristics, patterns, and problematic constructions as new reports in the technical literature appear and developers gain more experience.

References

[Antlr, 2003] Antlr v2.7.5, "http://www.antlr.org," 2003

[ArgoUML, 2007] ACE v2.4, "http://argouml.tigris.org," 2007

[Azureus, 2007] ACE v3.0, "http://azureus.sourceforge.net," 2007

[Bar and Ciupke, 1998] Holger Bar, Oliver Ciupke, "Exploiting design heuristics for automatic problem detection," Germany, 1998

[Booch, 1994] G. Booch, "Object Oriented Analysis and Design with applications," 2nd ed., Addison-Wesley, 1994.

[Brown, 1996] K. Brown, "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk," Technical Report TR-96-07, Dept. of Computer Science, North Carolina State Univ., 1996

[Brown et al., 1998] W. Brown, R., Malveau, McCormick III, H., Mowbray, T., "Anti-patterns – Refactoring Software, Architectures, and Projects in Crisis," Wiley Computer Publishing, 1998

[Chidamber and Kemerer, 1991] S.R. Chidamber, C.F. Kemerer, "Towards a Metrics Suite for Object Oriented design", in A. Paepcke, (ed.) Proc. Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91), October 1991. Published in SIGPLAN Notices, 26 (11), 197-211, 1991

[Chidamber and Kemerer, 1994] S.R. Chidamber, C.F. Kemerer, "a Metrics Suite for Object Oriented design", IEEE Transaction on Software Engineering, 20(6) : 476-493, June 1991

[Correa et al., 2000] Alexandre L. Correa, Cláudia M. L. Werner and Gerson Zaverucha, "Object Oriented Design Expertise Reuse: An Approach Based on

Heuristics, Design Patterns and Anti-patterns", Springer Berlin / Heidelberg, 2000

[Coplien, 1996] James Coplien, "Patterns Software," 1 ed., SIGS Books, 1996

[DEMEYER et al., 1998] S. Demeyer, S. Tichelaar, P. Steyaert, "FAMOOS – Definition of the Common Exchange Model," http://www.iam.unibe.ch /~famous /InfoExchFormat/

[Eric and Elisabeth, 2004] Eric Freeman and Elisabeth Freeman, "Head First Design Patterns," O'Reilly, Octorber 2004.

[Ernest, 2003] Ernest Friedman-hill, "Jess in action: Rule-Based System in Java". Manning, 2003

[Fenton, 1991] Norman Fenton, *Software Metrics: A Rigorous Approach*, Chapman & Hall, London, UK, 1991

[Florijn, 1997] Gert Florijn, Marco Meijers, and Pieter van Winsen, "Tool support in design patterns," In M. Aksit and S. Matsuoka, editors, Proceedings of the European Conference on Object-Oriented Programming, pages 472–495. LNCSv ol. 1241, June 1997

[Fowler, 1999] M. Fowler, "Refactoring: Improving the Design of Existing Code," AddisonWesley Longman Publishing Co., Inc, 1999

[Gabriel, 1995] Richard Gabriel, "Pattern definitions," Available from: http://hillside.net/patterns/definition.html, 1995+.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Pattern Elements of Reusable Object Oriented Software," Addison-Wesley, 1995

[Gibbon and Higgins, 1996] C. Gibbon and C. Higgins, "Teaching object-oriented design with heuristics," SIGPLAN Not., 31(7):12–16, 1996

[Giarratano and Riley, 1998] Joseph Giarratano and Gary Riley, "Expert Systems: principles and programming," PWS Publishing Company, 1998

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

[Guéhéneuc, 2005] Yann-Gaël Guéhéneuc, "Ptidej: Promoting Patterns with Patterns," In *Proceedings of the 1^{st} ECOOP workshop on Building a System using Patterns*, Springer-Verlag, July 2005.

[Haynes, 1996] P. Haynes, "Detection an Prevention of Software Cancer in OO Systems". Presented at OOPSLA 1996 Metrics Workshop, San Jose, 1996

[J2SE, 2005] J2SE JDK 6.0, "http://java.sun.com/," 2005

[Jess, 2006] Jess v7.0, "http://herzberg.ca.sandia.gov/jess/," 2006

[JHotDraw, 2007] ACE v7, "http://sourceforge.net/projects/jhotdraw," 2007

[KELLER et al., 1999] R. Keller, R. Schauer, S. Robitaille, P. Page, "Pattern-Based Reverse-Engineering of Design Components", International Conference on Software Engineering – ICSE'99, pp. 226-235, Los Angeles, CA, 1999

[Koenig, 1995] A. Koenig, "Patterns and antipatterns," Journal of Object Oriented Programming, 8(1), Marco, 1995

[Lane, 1986] N.E. Lane, "Global Issues in Evaluation of Expert Systems" in Proceedings of the 1986 IEEE International Conference on Systems, Man, and Cybernetics, IEEE Computer Society Press, Piscataway, New Jersey, 1986

[Lieberherr, 1996] K. J. Lieberherr, "Adaptive Object Oriented Software. The Demeter method with propagation patterns," PWS Publishing Company, 1996

[Lakos, 1996] John Lakos, "Large-Scale C++ Software Design," Addison-Wesley, 1996

[Li and Henry, 1999] Wei Li and Sallie Henry, "Maintenance Metrics for the Object-Oriented Paradigm", *Proceedings of the First International Software Metrics Symposium*, May 1993b

[Lorenz and Kidd, 1994] Mark Lorenz and Jeff Kidd, "Object-Oriented Software Metrics," Prentice Hall, Englewood Cliffs, NJ, 1994

[Martin, 2000] Robert C. Martin, "Design Principles and Design Patterns," http://www.objectmentor.com, 2006

[Martin, 1996a] Robert C. Martin, "The Open-Close Principle," http://www.objectmentor.com, 2006

[Martin, 1996b] Robert C. Martin, "The Dependency Inversion Principle," http://www.objectmentor.com, 2006

[Martin, 1996c] Robert C. Martin, "The Interface Segregation Principle," http://www.objectmentor.com, 2006

[Meyer, 1988] Bertrand Meyer, "Object Oriented Software Construction," Prentice Hall, 1988

[McCabe, 1976] T. J. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, SE-2(4):308–320, 1976

[Moha and Gueheneuc, 2006] N. Moha and Y. Gueheneuc, "Automatic Generation of Detection Algorithms for Design Defects," 21st IEEE International Conference on Automated Software Engineering (ASE'06) pp. 297-300, 2006

[Moha et al. 2006] N. Moha, D. Huynh and Y. Gueheneuc, "Une taxonomie et un metamodele pour la detection des defauts de conception," In actes du 12e

colloque Langages et Modeles a Objets, pages 201–216. Hermes Science Publications, March 2006

[Prechelt and Kramer, 1998] L. Prechelt and C. Kramer, "Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns," J. Universal Computer Science, vol. 4, no. 12, pp. 866-882, Dec. 1998

[Rational, 2005] IBM Rational Software Corp. "www.ibm.com/software/rational," 2005

[UML, 1997] Rational Software Corp. "UML Semantics". Version 1.1, 1997

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

[Rumbaugh et al. 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, "Object-Oriented Modeling and Design," Prentice-Hall, 1991.

[Stelting and Maassen, 2001] Stephen Stelting and Olav Maassen, "Applied Java patterns," Java series. Prentice Hall, December 2001.

[Together, 2006] Borland Together, "http://www.borland.com/us/products/together" Together Version 2006, 2006

[Wendehals, 2004] L. Wendehals, "Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams," Proc. Sixth Workshop Software Reeng. (WSR '04), pp. 63-64, May 2004.

[Wenzel, 2006] Sven Wenzel, "Automatic Detection of Incomplete Instances of Structural Patterns in UML Class Diagrams," Tampere University of Technology, Finland, 2006

Appendix A – Partial Antlr Java Grammar

```
class JavaRecognizer extends Parser;
options {
  ASTLabelType = "RefPNode";
  k = 2;                              // two token lookahead
  exportVocab=Java;                   // Call its vocabulary "Java"
  codeGenMakeSwitchThreshold = 2;     // Some optimizations
  codeGenBitsetTestThreshold = 3;
  defaultErrorHandler = false;//Don't generate parser error handlers
  buildAST = true;
}

tokens {
 BLOCK; MODIFIERS; OBJBLOCK; SLIST; CTOR_DEF; METHOD_DEF;
 VARIABLE_DEF; INSTANCE_INIT; STATIC_INIT; TYPE; CLASS_DEF;
 INTERFACE_DEF; PACKAGE_DEF; ARRAY_DECLARATOR; EXTENDS_CLAUSE;
 IMPLEMENTS_CLAUSE; PARAMETERS; PARAMETER_DEF; LABELED_STAT;
 TYPECAST; INDEX_OP; POST_INC; POST_DEC; METHOD_CALL; EXPR;
 ARRAY_INIT; IMPORT; UNARY_MINUS; UNARY_PLUS; CASE_GROUP; ELIST;
 FOR_INIT; FOR_CONDITION; FOR_ITERATOR; EMPTY_STAT; FINAL="final";
 ABSTRACT="abstract";
}

// Compilation Unit: In Java, this is a single file.  This is the
// start rule for this parser
compilationUnit :
  // A compilation unit starts with an optional package definition
  (packageDefinition|/* nothing */)
  // Next we have a series of zero or more import statements
  (importDefinition)*
  (typeDefinition )*
   EOF!;

// Package statement: "package" followed by an identifier.
packageDefinition options {defaultErrorHandler = true;}:
  p:"package"^ {#p->setType(PACKAGE_DEF);}
  identifier SEMI;

// Import statement: import followed by a package or class name
importDefinition options {defaultErrorHandler = true;}:
  i:"import"^ {#i->setType(IMPORT);}
  identifierStar SEMI;

// A type definition in a file is either a class or interface
definition.
typeDefinition options {defaultErrorHandler = true;}:
    m:modifiers!(classDefinition[#m] | interfaceDefinition[#m])
  | SEMI!   ;

// A declaration is the creation of a reference or primitive-type
variable
declaration! :
  m:modifiers
  t:typeSpec[false]
  v:variableDefinitions[#m,#t]{#declaration = #v;};

modifiers:
  ( modifier )*
```

```
    {#modifiers = #([MODIFIERS, "MODIFIERS"], #modifiers);};

// A type specification is a type name with possible brackets
// afterwards (which would make it an array type).
typeSpec[bool addImagNode] :
     classTypeSpec[addImagNode]
   | builtInTypeSpec[addImagNode];

// A class type specification is a class type with possible brackets
afterwards
//    (which would make it an array type).
classTypeSpec[bool addImagNode] :
  identifier (lb:LBRACK^ {#lb->setType(ARRAY_DECLARATOR);} RBRACK)*
    {
       if ( addImagNode ) {
          #classTypeSpec = #(#[TYPE,"TYPE"], #classTypeSpec);
       }
    }
   ;

// A builtin type specification is a builtin type with possible
brackets afterwards (which would make it //an array type).
builtInTypeSpec[bool addImagNode] :
  builtInType (lb:LBRACK^ {#lb->setType(ARRAY_DECLARATOR);} RBRACK)*
      {
        if ( addImagNode ) {
          #builtInTypeSpec = #(#[TYPE,"TYPE"], #builtInTypeSpec);
        }
      };
```

Appendix B – Jess Rule DTD

```
<!-- XML DTD for Jess rules -->
<!ELEMENT rulebase (rule)*>
<!ELEMENT rule (lhs,rhs)>
<!ATTLIST rule name CDATA #REQUIRED priority CDATA "">
<!ELEMENT rhs (function-call)*>
<!ELEMENT lhs (group | pattern)*>
<!ELEMENT group (group | pattern)*>
<!ATTLIST group name CDATA #REQUIRED>
<!ELEMENT pattern (slot*)>
<!ATTLIST pattern name CDATA #REQUIRED binding CDATA "">
<!ELEMENT slot (variable | constant | function-call)*>
<!ATTLIST slot name CDATA #REQUIRED>
<!ELEMENT variable EMPTY>
<!ATTLIST variable name CDATA #REQUIRED>
<!ELEMENT function-call (head,(constant|variable|function-call)*)>
<!ELEMENT head (#PCDATA)>
<!ELEMENT constant (#PCDATA)>
```

Appendix C – Jess Rule XSLT

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" indent="no"/>
  <xsl:strip-space elements="*"/>
  <!-- Top-level rule template -->
  <xsl:template match="rule">
    <xsl:text>(defrule </xsl:text>
    <xsl:value-of select="@name"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:if test="@priority != ''">
      <xsl:text>  (declare (salience </xsl:text>
      <xsl:value-of select="./@priority"/>
      <xsl:text>))&#xA;</xsl:text>
    </xsl:if>
    <xsl:apply-templates select="./lhs"/>
    <xsl:text>  =&gt;</xsl:text>
    <xsl:apply-templates select="./rhs"/>
    <xsl:text>)&#xA;</xsl:text>
  </xsl:template>

  <!-- Rule left hand sides -->
  <xsl:template match="lhs">
    <xsl:for-each select="./group | ./pattern">
      <xsl:text>  </xsl:text>
      <xsl:apply-templates select="."/>
    <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
  <xsl:template match="group">
    <xsl:text>(</xsl:text>
    <xsl:value-of select="./@name"/>
    <xsl:text> </xsl:text>
    <xsl:apply-templates/>
    <xsl:text>)</xsl:text>
  </xsl:template>
  <xsl:template match="pattern">
    <xsl:if test="@binding != ''">
      <xsl:text>?</xsl:text>
      <xsl:value-of select="@binding"/>
      <xsl:text> &lt;- </xsl:text>
    </xsl:if>
    <xsl:text>(</xsl:text>
    <xsl:value-of select="./@name"/>
    <xsl:apply-templates select="./slot"/>
    <xsl:text>)</xsl:text>
  </xsl:template>
  <xsl:template match="slot">
    <xsl:text> (</xsl:text>
    <xsl:value-of select="./@name"/>
    <xsl:for-each select="./*">
      <xsl:if test="position() != 1">
        <xsl:text>&amp;</xsl:text>
      </xsl:if>
      <xsl:apply-templates select="."/>
    </xsl:for-each>
    <xsl:text>)</xsl:text>
  </xsl:template>
  <xsl:template match="slot/function-call">
    <xsl:text>:</xsl:text>
    <xsl:call-template name="funcall"/>
```

```
  </xsl:template>

  <!-- Rule right hand sides -->
  <xsl:template match="rhs/function-call">
    <xsl:text>&#xA;  </xsl:text>
    <xsl:call-template name="funcall"/>
    <xsl:text></xsl:text>
  </xsl:template>

  <!-- Function calls -->
  <xsl:template match="function-call">
    <xsl:call-template name="funcall"/>
  </xsl:template>
  <xsl:template name="funcall">
    <xsl:text>(</xsl:text>
    <xsl:apply-templates select="./*"/>
    <xsl:text>)</xsl:text>
  </xsl:template>
  <xsl:template match="function-call/function-call">
    <xsl:text> </xsl:text>
    <xsl:call-template name="funcall"/>
  </xsl:template>

  <!-- Miscellaneous -->
  <xsl:template match="variable">
    <xsl:text> ?</xsl:text>
    <xsl:value-of select="@name"/>
  </xsl:template>
  <xsl:template match="constant">
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

Appendix D – Quality Heuristic Jess Rules

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;Rules for design heuristics

(defrule ADsbHwiC
      "all data should be hidden within its class"
   (classifier (classifierName ?clsName))
   (attribute (classifier ?clsName)(visibility ?attrVisiblity))
   (test (neq private ?attrVisiblity))
   (test (neq protected ?attrVisiblity))
    =>
   (assert (dataShouldHidden (class ?clsName))))

(defrule CsnCmtSO
      "a class should not contain more than six objects"
  (classifier (classifierName ?clsName))
    ?total <- (accumulate
    (bind ?count 0)                  ; initializer
    (bind ?count (+ ?count 1))       ; action
    ?count                           ; result
    (attribute (classifier ?clsName)
    (typeName ?typeName&~int&~long&~boolean&~String)))
  (test (> ?total 6))
   =>
  (assert (moreSixObjects (class ?clsName))))

(defrule ADiBCsbP
      "all data in a base class should be private"
  (classifier (classifierName ?clsName)(isRoot root))
      (attribute (classifier ?clsName)
      (visibility ?attrVisiblity&~private))
  (inheritsFrom (specificClassifier ?clsSpec)
      (genericClassifier ?clsName))
  =>
  (assert (baseDataShouldPrivate (class ?clsName))))

(defrule ABCsbAC
      "all base class should be abstract classes"
  (classifier (classifierName ?clsName)(isRoot root)
        (isAbstract ?clsIsAbstract&~abstract))
  (inheritsFrom (specificClassifier ?clsSpec)
            (genericClassifier ?clsName))
  =>
  (assert (baseShouldAbstract (class ?clsName))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;Rules for anti patterns
;;anti singleton
(defrule anti-singleton "recognize anti-singleton anti-pattern"
  (singleton-pattern (singleton ?clsSingleton))
  (classifier (classifierName ?clsAS))
  (attribute (classifier ?clsAS)
        (attributeName ?attrName)
        (typeName ?clsSingleton))
   =>
   (assert (anti-singleton
                 (singleton ?clsSingleton)(violations ?clsAS))))

;;anti-facade
```

```
(defrule anti-facade "recognize anti-facade anti-pattern"
  (classifier (classifierName ?cls1)(package ?pkg1))
  (classifier (classifierName ?cls2&~?cls1)
        (package ?pkg2&~?pkg1))
  (attribute (classifier ?cls2)(attributeName ?attr2)
        (typeName ?cls1))
  (classifier (classifierName ?cls3&~?cls1)
        (package ?pkg1))
  (classifier (classifierName ?cls4&~?cls2)
        (package ?pkg3&~?pkg1))
  (attribute (classifier ?cls4)(attributeName ?attr4)
        (typeName ?cls3))
  =>
  (assert (anti-facade
            (package ?pkg1)(classes ?cls1 ?cls3)
            (otherPackages ?pkg2 ?pkg3)
            (classesFromOtherPackages ?cls2 ?cls4))))

;;blob
(defrule blob "recognize blob anti-pattern, a class contains more
         than 60 attributes and methods"
  (classifier (classifierName ?clsName))
  ?totalAttrbutes <- (accumulate
            (bind ?count 0)                    ; initializer
            (bind ?count (+ ?count 1))  ; action
            ?count                      ; result
            (attribute (classifier ?clsName)))
  ?totalMethods <- (accumulate
            (bind ?count 0)                    ; initializer
            (bind ?count (+ ?count 1))  ; action
            ?count                      ; result
            (operation (classifier ?clsName)))
  (test (> (+ ?totalMethods ?totalAttrbutes) 60))
 =>
   (assert (blob (class ?clsName))))

(defrule globalScopeObject "Global Scope Object anti-pattern"
  (classifier (classifierName ?clsName))
  (attribute (classifier ?clsName)(unqualifiedName ?attrUName)
        (visibility public)
  (typeName ?type&~int&~long&~boolean&~String))
  =>
  (assert (globalScopeObject(class ?clsName)
            (attribute ?attrUName))))

(defrule antiCommoncodePrivateFunction
  (classifier (classifierName ?clsTarget))
  (operation (classifier ?clsTarget)
        (operationName ?optNameTarget)(visibility public))

  ;Internal Client
  (operation (classifier ?clsInternal)
            (operationName ?optNameInternal))
  ;Same Hierarchy
  (or (or (descendant (ancestor ?clsInternal)
                (descendant ?clsTarget))
          (ancestor (ancestor ?clsTarget)
                (descendant ?clsInternal)))
      (test (eq ?clsInternal ?clsTarget)))
  (invokes (classifierCaller ?clsInternal)
        (operationCaller ?optNameInternal)
        (classifierCallee ?clsTarget)
        (operationCallee ?optNameTarget))
  (test (neq ?optNameTarget ?optNameInternal))
```

```
  ;External Client
  (not
     (and
        (and
           (and (operation (classifier ?clsExternal)
                  (operationName ?optNameExternal))
                (operation (classifier ?clsTarget)
                  (operationName ?optNameTarget)))
            ;Not Same Hierarchy
            (not
               (or (or (descendant (ancestor ?clsExternal)
                          (descendant ?clsTarget))
                       (ancestor (ancestor ?clsTarget)
                          (descendant ?clsExternal)))
                   (test (eq ?clsExternal ?clsTarget)))))
               (invokes (classifierCaller ?clsExternal)
                   (operationCaller ?optNameExternal)
                   (classifierCallee ?clsTarget)
                   (operationCallee ?optNameTarget))))
  =>
  (assert (antiCommoncodePrivateFunction
            (classifier ?clsTarget)(operation ?optNameTarget))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;Rules for design patterns
;;;;;;Creational patterns

;;Abstract Factory pattern rule
(defrule abstractFactory "recognize abstract factory pattern"
    ;AF is abstract and has a function whose return type is AP
    ;it's better to check AF has abstract method because it's
    ;possible that AF is not abstract but it has abstract method.
  (classifier (classifierName ?clsAF)(isAbstract abstract))
  (operation (classifier ?clsAF)
            (unqualifiedName ?optUNameAF)(operationName ?optAF))
  (parameter (operationName ?optAF)
            (direction return) (parameterType ?clsAP))
  ;CF is concrete, has a function whose return type is AP
  (classifier (classifierName ?clsCF)(isAbstract concrete))
  (operation (classifier ?clsCF)
            (unqualifiedName ?optUNameAF)(operationName ?optCF))
  (parameter (operationName ?optCF)
            (direction return) (parameterType ?clsAP))

  ;test optA and optB have the same signature
  (not (and
         (parameter (operationName ?optAF)
              (order ?orderAF)(parameterType ?typeParamAF))
         (parameter (operationName ?optCF)
              (order ?orderAF)(parameterType ~?typeParamAF))))

  ;CF subtype AF
  (inheritsFrom (specificClassifier ?clsCF)
            (genericClassifier ?clsAF))
  ;AP is abstract and name is AF/CF's funtion return type
      (classifier (classifierName ?clsAP)(isAbstract abstract))
  ;CP subtype AP
    (inheritsFrom (specificClassifier ?clsCP)
          (genericClassifier ?clsAP))
  =>
  (assert (abstractFactory-pattern (abstractFactory ?clsAF)
            (concreteFactories ?clsCF)(abstractProducts ?clsAP)
            (concreteProducts ?clsCP))))
```

```
;;Singleton pattern rule
(defrule singleton "recognize singleton pattern"
    ;To prevent clients instantiating singleton class
  (or
    ;abstract class
    (classifier (classifierName ?clsSingleton)(isAbstract abstract))
    ;constructor is private or protected
    (operation (classifier ?clsSingleton)
            (unqualifiedName -constructor-)(visibility private)))
  ;lazySingleton's attribute should be private static
  ;but not final. public and private singletion should be final
  (attribute (classifier ?clsSingleton)(scope class)
            (visibility private)(typeName ?clsSingleton))
  (operation (classifier ?clsSingleton)
            (unqualifiedName ?optUNameSingleton)
            (operationName ?optSingleton)
            (scope class)(visibility public))
  (parameter (operationName ?optSingleton)
            (direction return) (parameterType ?clsSingleton))

  =>
  (assert (singleton-pattern (singleton ?clsSingleton))))

;;;;;;Structural patterns
;;Adapter pattern rule
(defrule adapter "recognize adapter pattern"
    ;there is a class adapter, I didn't implement it here.
  ;Target
  (classifier (classifierName ?clsTarget)(isAbstract abstract))
  (operation (classifier ?clsTarget) (operationName ?nameOptTarget)
            (unqualifiedName ?unameOptTarget))

  ;Adapter
  (classifier (classifierName ?clsAdapter))
  (attribute (classifier ?clsAdapter)(typeName ?clsAdaptee))
  (operation (classifier ?clsAdapter)(operationName ?nameOptAdapter)
            (unqualifiedName ?unameOptAdapter))

  ;Adaptee
  (classifier (classifierName ?clsAdaptee))
  (operation (classifier ?clsclsAdaptee)
            (operationName ?nameOptAdaptee)
            (unqualifiedName ?unameOptAdaptee))

  ;Adapter's instance invokes adaptee's interface
  (invokes (classifierCaller ?clsAdapter)
            (operationCaller ?nameOptAdapter)
            (classifierCallee ?clsAdaptee)
            (operationCallee ?nameOptAdaptee))

  ;Adapter inheritance Target
  (inheritsFrom (specificClassifier ?clsAdapter)
            (genericClassifier ?clsTarget))
  (test (eq ?unameOptTarget ?unameOptAdapter))
  ;Target and Adaptee have to different ?
  (test (neq ?clsAdaptee ?clsTarget))
  =>
  (assert (adapter-pattern (target ?clsTarget)
            (adapter ?clsAdapter)(adaptee ?clsAdaptee))))

;;Bridge pattern rule
(defrule bridge "recognize bridge pattern"
  ;Abstraction may or may not be abstract class
  (classifier (classifierName ?clsAbs))
```

```
  (attribute (classifier ?clsAbs)(typeName ?clsImp))
  (operation (classifier ?clsAbs)(unqualifiedName ?unameOptAbs)
            (operationName ?nameOptAbs))

  ;Refined Abstraction
  (classifier (classifierName ?clsRA)(isAbstract concrete))
  (operation (classifier ?clsRA)(unqualifiedName ?unameOptRA)
            (operationName ?nameOptRA))

  ;RA inheritance Abs
  (inheritsFrom (specificClassifier ?clsRA)
            (genericClassifier ?clsAbs))
  (test (eq ?unameOptAbs ?unameOptRA))

  ;Implementor
  (classifier (classifierName ?clsImp)(isAbstract abstract))
  (operation (classifier ?clsImp)(unqualifiedName ?unameOptImp)
            (operationName ?nameOptImp))

  ;Abstraction's instance invokes implement's interface
  (invokes (classifierCaller ?clsAbs)(operationCaller ?nameOptAbs)
            (classifierCallee ?clsImp)(operationCallee ?nameOptImp))

  ;Concrete Implemntor
  (classifier (classifierName ?clsCI)(isAbstract concrete))
  (operation (classifier ?clsCI)(unqualifiedName ?unameOptCI)
            (operationName ?nameOptCI))

  ;CI inheritance Imp
  (inheritsFrom (specificClassifier ?clsCI)
            (genericClassifier ?clsImp))
  (test (eq ?unameOptImp ?unameOptCI))
  =>
  (assert (bridge-pattern (abstraction ?clsAbs)
            (refinedAbstractions ?clsRA)(implementor ?clsImp)
            (concreteImplementors ?clsCI))))

;;Decorator pattern rule
(defrule decorator "recognize decroator pattern"
  ;AC
  (classifier (classifierName ?clsAC)
            (isAbstract abstract)(isRoot root))
  (operation (classifier ?clsAC)(unqualifiedName ?unameOptAC)
            (operationName ?nameOptAC))
  ;CC
  (classifier (classifierName ?clsCC)(isAbstract concrete))
            (operation (classifier ?clsCC)
            (unqualifiedName ?unameOptAC)(operationName ?nameOptCC))
  ;CC inheritance AC
  (inheritsFrom (specificClassifier ?clsCC)
            (genericClassifier ?clsAC))
  ;AD
  (classifier (classifierName ?clsAD))
  (operation (classifier ?clsAD)(unqualifiedName ?unameOptAD)
            (operationName ?nameOptAD))
  ;AD inheritance AC
  (inheritsFrom (specificClassifier ?clsAD)
            (genericClassifier ?clsAC))

  (or ;AD calls method of AC, Decorator Variant1: AJP
      (invokes (classifierCaller ?clsAD)(operationCaller ?nameOptAD)
            (classifierCallee ?clsAC)(operationCallee ?nameOptAC))
      (and
            ;Decorator Variant 2: HFDP, CD calls method of AC
      (invokes (classifierCaller ?clsCD)(operationCaller ?nameOptCD)
```

```
            (classifierCallee ?clsAC)(operationCallee ?nameOptAC))
        ;CD
        (classifier (classifierName ?clsCD)(isAbstract concrete))
        (operation (classifier ?clsCD)
                   (unqualifiedName ?unameOptCD)
                   (operationName ?nameOptCD))))

  ;CD inheritance AD
  (inheritsFrom (specificClassifier ?clsCD)
        (genericClassifier ?clsAD))
  (test (neq ?clsCC ?clsAD))
  (test (neq ?clsAC ?clsAD))
  =>
  (assert (decorator-pattern (abstractComponent ?clsAC)
            (concreteComponents ?clsCC)(abstractDecorator ?clsAD)
            (concreteDecorators ?clsCD))))

;;;;;;Behavioral patterns
;;Visitor pattern rule
(defrule visitor "recognize visitor pattern"
  ;AV
  (classifier (classifierName ?clsAV)(isAbstract abstract))
  (operation (classifier ?clsAV)(unqualifiedName ?unameOptAV)
            (operationName ?nameOptAV))
  (parameter (operationName ?nameOptAV)(direction ~return)
            (order ?orderParaAV)(parameterType ?typeParaAV))
  ;CV
  (classifier (classifierName ?clsCV)(isAbstract concrete))
  (operation (classifier ?clsCV)(unqualifiedName ?unameOptAV)
            (operationName ?nameOptCV))
  (parameter (operationName ?nameOptCV)(order ?orderParaAV)
            (parameterType ?typeParaAV))

  ;CV inheritance AV
  (inheritsFrom (specificClassifier ?clsCV)
            (genericClassifier ?clsAV))

  ;AN
  (classifier (classifierName ?clsAN)(isAbstract abstract))
  (operation (classifier ?clsAN)(unqualifiedName ?unameOptAN)
            (operationName ?nameOptAN))
  (parameter (operationName ?nameOptAN)(direction ~return)
            (order ?orderParaAN)(parameterType ?typeParaAN))
  ;CN
  (classifier (classifierName ?clsCN)(isAbstract concrete))
  (operation (classifier ?clsCN)(unqualifiedName ?unameOptAN)
            (operationName ?nameOptCN))
  (parameter (operationName ?nameOptCN)(order ?orderParaAN)
            (parameterType ?typeParaAN))
  ;CN inheritance AN
  (ancestor (ancestor ?clsAN) (descendant ?clsCN))
  ;CV's visit method take AN as one of its parameter type
  ;CN's accept method take AV as one of its parameter type
  (test (eq ?typeParaAV ?clsCN))
  (test (eq ?typeParaAN ?clsAV))
  =>
  (assert (visitor-pattern (abstractVisitor ?clsAV)
            (concreteVisitors ?clsCV)(abstractNode ?clsAN)
            (concreteNodes ?clsCN))))

;;Observer pattern rule
(defrule observer "recognize observer pattern"
  ;AS
  (classifier (classifierName ?clsAS)(isRoot root))
  ;Hyphosis operation name has "add" or "register" prefix
```

```
  ;it has the parameter whose type is AO type
  (operation (classifier ?clsAS)(unqualifiedName ?unameOptAddAS&:
  (or (and (>= (str-length ?unameOptAddAS) 3)
     (= 0 (str-compare (sub-string 1 3 ?unameOptAddAS) "add")))
     (and (>= (str-length ?unameOptAddAS) 8)
   (= 0 (str-compare (sub-string 1 8 ?unameOptAddAS) "register")))))
  (operationName ?nameOptAddAS))
      (parameter (operationName ?nameOptAddAS)
      (direction nil)(parameterType ?clsAO))
  ;Hyphosis operation name has "remove" or "unregister" prefix
  ;it has the parameter whose type is AO type
  (operation (classifier ?clsAS)(unqualifiedName ?unameOptRemoveAS&:
      (or (and (>= (str-length ?unameOptRemoveAS) 6)
                (= 0 (str-compare
                     (sub-string 1 6 ?unameOptRemoveAS) "remove")))
                (and (>= (str-length ?unameOptRemoveAS) 10)
                (= 0 (str-compare
                (sub-string 1 10 ?unameOptRemoveAS) "unregister")))))
      (operationName ?nameOptRemoveAS))
      (parameter (operationName ?nameOptRemoveAS)
        (direction nil)(parameterType ?clsAO))
  ;AS's notify function
  (operation (classifier ?clsAS) (operationName ?nameOptNotifyAS)
      (unqualifiedName
           ?unameOptNotifyAS&~?unameOptAddAS&~?unameOptRemoveAS))

  ;AO
  (classifier (classifierName ?clsAO)
           (isAbstract abstract)(isRoot root))
  ;AO's update function
  (operation (classifier ?clsAO)(unqualifiedName ?unameOptUpdateAO)
           (operationName ?nameOptUpdateAO))

  ;AS's notify function invokes AO's update function
  (invokes (classifierCaller ?clsAS)
           (operationCaller ?nameOptNotifyAS)
           (classifierCallee ?clsAO)
           (operationCallee ?nameOptUpdateAO))

  ;AS can navigate to AO, that is, AS is associate with AO
  ;AS has an(vector) attribute(s) whose type is AO
  ;(navigable (classifierSubject ?clsAS)(classifierObject ?clsAO))

  ;CS
  (classifier (classifierName ?clsCS))
  (or ;CS and AS can be one class
    (test (eq ?clsCS ?clsAS))
    ;CC inheritance AC
   (inheritsFrom (specificClassifier ?clsCS)
                 (genericClassifier ?clsAS)))

  ;CO
  (classifier (classifierName ?clsCO))
  ;CO inheritance AO
  (inheritsFrom (specificClassifier ?clsCO)
           (genericClassifier ?clsAO))
=>
  (assert (observer-pattern (abstractSubject ?clsAS)
           (concreteSubject ?clsAS)(abstractObserver ?clsAO)
           (concreteObservers ?clsCO))))

;;Strategy, State and Command
(defrule Strategy "recognize strategy, state and Command pattern"
  ;Context
  (classifier (classifierName ?clsContext)(isAbstract concrete))
```

```
  (attribute (classifier ?clsContext)(typeName ?clsStrategy))
  (operation (classifier ?clsContext)(operationName ?nameContext)
            (unqualifiedName ?unameContext))

  ;Strategy
  (classifier (classifierName ?clsStrategy)(isAbstract abstract))
  (operation (classifier ?clsStrategy)
            (unqualifiedName ?unameOptStrategy)
            (operationName ?nameOptStrategy))

  ;Abstraction's instance invokes implement's interface
  (invokes (classifierCaller ?clsContext)
           (operationCaller ?nameOptContext)
           (classifierCallee ?clsStrategy)
           (operationCallee ?nameOptStrategy))

  ;Concrete strategies
  (classifier (classifierName ?clsCS)(isAbstract concrete))
  (operation (classifier ?clsCS)(unqualifiedName ?unameOptCS)
                  (operationName ?nameOptCS))

  ;Concrete Strategy inheritance Strategy
  (inheritsFrom (specificClassifier ?clsCS)
                  (genericClassifier ?clsStrategy))
  ;better to test if both have the same signature
  (test (eq ?unameOptCS ?unameOptStrategy))
  =>
  (assert (strategy-pattern (context ?clsContext)
      (strategy ?clsStrategy)(concreteStrategies ?clsCS))))

(defrule same-signature
      "two method have the same signatures"
  (classifier (classifierName ?clsA))
  (operation (classifier ?clsA)(operationName ?optA)
            (unqualifiedName ?uqnameOptA)
            (numberParameters ?paraANumber))

  (classifier (classifierName ?clsB))
  (operation (classifier ?clsB)(operationName ?optB)
            (unqualifiedName ?uqnameOptA)
            (numberParameters ?paraANumber))

  (not (and (parameter (operationName ?optA)
                  (order ?orderA)(parameterType ?typeParamA))
            (parameter (operationName ?optB)
                  (order ?orderA)(parameterType ~?typeParamA))))

  (ancestor (ancestor ?clsA)(descendant ?clsB))
  =>
  (assert (sameSignature (methodA ?optA)(methodB ?optB)))
```