Design Patterns as Laws of Quality

Yann-Gaël Guéhéneuc Jean-Yves Guyomarc'h
Khashayar Khosravi Houari Sahraoui

GEODES - Group of Open and Distributed Systems, Experimental Software Engineering

Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
{guehene,guyomarj,khosravk,sahraouh}@iro.umontreal.ca

Contents

1 Introduction

This chapter is a complete coverage of our current work on software quality models and on design pattern identification. In this chapter, we explore the idea of facts in science in relation with software quality models. We show how design patterns can be used as facts to devise a quality model and we describe the processes of building and of applying such a quality model.

In science, facts are subject of observations by the scientists, who hypothesise laws to formalise recurring observations and theories to frame and to explain the laws. To the authors' best knowledge in informatics, many facts and observations have been recorded and published but only few laws and theories [Endres and Rombach, 2003]. This lack of laws and theories impedes the successful development of many software and reduces our trust in software and in software science.

The lack of laws and theories is particularly visible in software quality. In particular, there do not yet exist general software quality models that could be applied to *any* software. It is indeed difficult to build quality models without concrete laws on software and software quality. Thus, existing quality models attempt to link internal attributes of classes and external quality characteristics with little regard for actual facts on software quality and without taking into account some dimensions of the evaluated software, such as its architecture.

In the following, we use design patterns [Gamma et al., 1994] as general laws to build a software quality model. We choose design patterns because they are now well-known constructs and have been studied extensively. Design patterns provide "good" solutions to recurring design problems. They define a problem in the form of an intent and motivations and provide a solution in the form of a design motif, a prototypical micro-architecture that developers use in their design to solve the problem. Design patterns are said to promote software elegancy through flexibility, reusability, and understandability [Gamma et al., 1994, page xiii].

We assume that the design motifs provided by design patterns show flexibility, reusability, and understandability. Also, we assume that we can use design motifs as laws on software quality, as their authors intended, if not explicitly: Whenever developers use design motifs, they want to promote flexibility, reusability, and understandability in their design. Thus, whenever we find a micro-architecture similar to a design motif in a program architecture, we assume that this micro-architecture promotes (or was an attempt to promote) flexibility, reusability, and understandability.

We make the following parallels: A micro-architecture from a program architecture is a fact. That a micro-architecture displays flexibility, reusability, and understandability is an observation. A design motif defines some laws on the quality characteristics of the observed micro-architecture: It formalises recurring micro-architectures that display flexibility, reusability, and understandability. We use these laws to assess the quality characteristics of software, *i.e.*, we use design motifs to build a software quality model to assess the quality of micro-architectures, identified in a software architecture.

1.1 Definition of the Problem

We want to build a quality model that considers quality characteristics covering software elegancy. Software elegancy is highly important during software maintenance to reduce the effort (time and cost) of maintainers. We need to choose external quality characteristics related to software elegancy and to find software metrics to fill the space between characteristics and software artifacts. We use design motifs as a basis to choose quality characteristics.

Existing quality models attempt to link internal attributes of classes and external quality characteristics with little regard for the actual architectures of the programs. Thus, these quality models can hardly distinguish between well-structured programs and programs with poor architectures. We use motifs to assess programs quality characteristics using both programs internal attributes and their architectures as a mean to capture the quality of program architectures in a quality model.

If we were in art rather than in informatics, we would say that existing quality models use identical quality models to compare a cubist painting, such as “Femme Profile” by Pablo Picasso (1939), with a realist picture, as shown in Figure 1. The two faces possess two eyes, one nose, two ears, and one mouth but with very different organisations, none being more beautiful, only more beautiful according to different laws.

Figure 1: A woman’s profile: Cubist (left) and realist versions (right)

1.2 Underlying Assumptions

In building a quality model using design motifs we make the following underlying assumptions.

Human Factor. Some existing quality models can predict fault-proneness with reasonable accuracy in certain contexts. Other quality models attempt at evaluating several quality characteristics but fail at providing reasonable accuracy, from lack of data mainly.

We believe that quality models must evaluate high-level quality characteristics with great accuracy in terms well-known to software engineers to help maintainers in assessing programs and thus in predicting maintenance effort.

Such quality models can also help developers in building better quality programs by exposing the relationships between internal attributes and external quality characteristics clearly.

We take a less "quantitative" approach than quality models counting, for example, numbers of errors per classes and linking these numbers with internal attributes. We favour a more "qualitative" approach linking quality characteristics related to the maintainers' perception and work directly.

Quality Theory. Unfortunately, software engineering is well-known for its lack of laws and theories. Software engineers do not have theories to support their work on development, maintenance, and to explain quality yet.

Thus, it is important to gather as much facts, observation and laws as possible. Design motifs are an interesting bridge between internal attributes of programs, external quality characteristics, and software engineers. We use motifs as laws to link internal attributes (concrete implementation of programs) in the one hand and subjective quality characteristics (subjective perceptions on programs) on the other hand.

Program Architecture. Pairwise dependencies among classes and internal attributes of classes are not enough: The organisations of classes, the program architectures, are important because they are the first things software engineers see and deal with.

A large body of work exist on program architecture, in particular on architectural drift or decay [Tran et al., 2000], which aims at analysing, organising, and tracking the modifications that architectures must undergo to keep them easy to understand and to modify, and thus to reduce maintenance effort [Kerievsky, 2004].

However, to our best knowledge, no work attempted to develop quality models using programs internal attributes while considering their architectures explicitly. We try to build such a quality model using design motifs as laws on architectural quality. Moreover, software engineers use motifs and patterns, even unconsciously, when developing and maintaining programs [Gamma et al., 1994, page xiii].

1.3 Process of Building a Quality Model

The process of building a quality model decomposes in three main tasks:

- Choosing and organising characteristics related to software maintenance.
- Choosing internal attributes that are computable with metrics.
- Linking quality characteristics with internal attributes to produce evaluation rules.

The process of building a quality model decomposes in the following tasks when using design motifs to consider program architectures:

1. *Identifying the quality characteristics shared by a set of design motifs which make programs more maintainable.* This task consists in identifying quality characteristics and sub-characteristics related to some motifs of interest. Among all possible characteristics, we can focus on characteristics for program maintenance.

2. *Organising the quality characteristics identified from the design motifs.* This task consists in organising quality characteristics and sub-characteristics hierarchically [Fenton and Pfleeger, 1997] to build a quality model which can be linked with software artifacts using metrics.

3. *Choosing internal attributes relevant to design motifs and their quality characteristics.* This task consists in choosing internal attributes which can be measured with metrics. The internal attributes must relate to the quality model from task 2, to link software artifacts with quality characteristics.

4. *Identifying programs implementing the design motifs.* This task consists in identifying a set of programs in which developers used the design patterns of interest. We name this set of base programs $\mathcal{BP}$.

5. *Assessing the quality of design motifs using the quality model built in task 2.* This task consists in assessing the quality of design motifs in the set of base programs $\mathcal{BP}$ manually, with the characteristics and sub-characteristics of the quality model.

6. *Computing the metrics identified in task 3 on the design motifs in the programs identified in task 5.* This task consists in computing metric values for the design motifs of interest identified in $\mathcal{BP}$. If class-based metrics are used, then we can compute the metric values of the design motifs as the average or as the variance of the class-based metric values.

7. *Linking the metric values computed in task 6 and the evaluation of the quality sub-characteristics and characteristics performed in task 5.* This task consists in applying a machine learning technique to link internal attributes of programs measured with metric values computed in task 6 and the evaluation of the quality sub-characteristics and characteristics from task 5.

8. *Validating the obtained quality model on other well-known uses of design motifs.* This task consists in applying the evaluation rules from task 7 on other well-known programs to assess the evaluative power of the quality model. Using design motifs, we must apply our quality model on well-known uses of design motifs in programs with known quality characteristics.

The results of the eight previous tasks is a quality model which can evaluate the quality characteristics of programs while considering the program architectures through the assessment of the quality of design motifs. The quality model decomposes in several rules for each quality sub-characteristics and characteristics. These rules depends on different metrics to assess quality.

Figure 2 displays a simplified version of our process. First, we identify programs implementing design motifs. Second, we identify in these programs the design motifs used. Third, we evaluate the quality sub-characteristics and characteristics of the design motifs manually. Fourth, we compute metrics for each identified design motifs (by averaging class-based metrics, for example). Fifth, we use machine learning techniques to link metric values with the quality sub-characteristics and characteristics of design motifs.

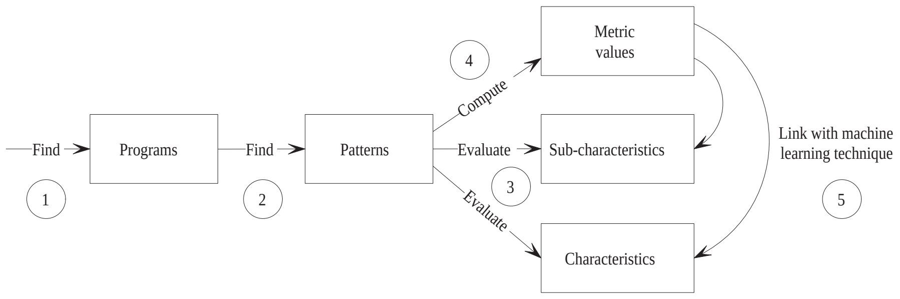

Figure 2: Simplified process of building a quality model considering program architectures, through design motifs

1.4 Process of Applying a Quality Model

Once we built a quality model using design motifs to consider the quality of program architectures in addition to internal attributes of classes or couple thereof, applying such a quality model requires to consider micro-architectures in a program architecture and to apply the quality model on these micro-architectures to assess their quality.

Again, if we were in art, we could say that existing quality models assess the quality characteristics of paintings by looking at many tiny parts of the painting (for example, classes in program architectures) rather than by looking at larger pieces of the painting (*i.e.*, design motifs in program architectures), such as sketched in Figure 3.

Thus, applying our quality model requires the four following tasks:

1. *Identifying micro-architectures similar to design motifs in the architecture of the program $\mathcal{P}$ under evaluation.* There are many techniques existing to identify design motifs in programs, for examples logic programming [Wuyts, 1998] or constraint programming [Guéhéneuc et al., 2004]. In the following, we present our technique using explanation-based constraint programming (see Section 4 page 14).

Figure 3: Level of details considered in existing software quality model (left) versus in quality models based on patterns (right)

2. *Measuring the internal attributes of each classes of the micro-architectures with metrics and averaging class-based metric values, if needed.* This task is straightforward, many tools existing to apply metrics on programs.

3. *Adapting the rules built from $\mathcal{BP}$ to $\mathcal{P}$ by computing the ratio between the metric values from $\mathcal{BP}$ and the metric values from $\mathcal{P}$.* This task consists in adapting the rules associated with the quality model built from $\mathcal{BP}$. Indeed, the rules are built from metric values with a certain minimum and maximum values depending on $\mathcal{BP}$, these values differ from the minimum and maximum values for $\mathcal{P}$. We compute the ratio between $min_{\mathcal{BP}}$ and $max_{\mathcal{BP}}$, on the one hand, and $min_{\mathcal{P}}$ and $max_{\mathcal{P}}$, on the other hand. Figure 4 illustrates rule adaptation.

   Yet again, if we were in art and we would like to compare the eyes in two different paintings, we would adapt the scales of the eyes before making any comparison.

4. *Applying our quality model on the identified micro-architectures.* This task consists in applying the rules adapted from the quality model on the metric values computed for the micro-architectures found in program $\mathcal{P}$.

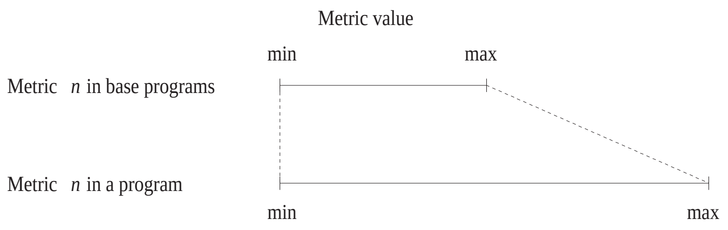

Figure 4: Adapting the rules of the quality model, ratio between minimum and maximum metric values of $\mathcal{BP}$ and $\mathcal{P}$

1.5 Discussion

The use of design motifs as laws on software quality brings an extra level of abstraction to the building of our quality model with respect to existing quality models. Indeed, we use design motifs for three purposes: First, we survey quality characteristics of design motifs theoretically to define and to organise the quality characteristics of our quality model; Second, we validate our quality model on well-known uses of design motifs; Third, we apply our quality model on micro-architectures similar to design motifs.

We can use our quality model on any micro-architecture, independently of the micro-architectures sizes and organisations, and thus potentially on complete program architectures. This use is similar to the use of existing quality models which are built using internal attributes and quality characteristics of given programs but applied on similar yet sometimes very differing programs.

The use of design motifs as a basis to build a quality model results in our choice to study qualitative quality characteristics over quantitative characteristics, such as fault-proneness. Thus, we want to build a quality model tailored for maintainers, evaluating "qualitative" characteristics with which maintainers can predict maintenance effort.

We choose design motifs and design patterns because developers always use patterns. Indeed, developers always use recurring solutions to solve design problems. Thus, design patterns are an integral part of any reasonably well-developed programs [Gamma et al., 1994].

However, the use of design motif is but a step towards quality models that can evaluate software quality while considering the architectures of programs. Indeed, a quality model built using design motifs assesses the quality of programs through larger *parts* than existing quality models because it uses micro-architectures instead of classes. Yet, it does not consider the overall architectures of the programs: It is similar in art to assessing the quality of a painting using parts rather than looking at the whole picture, such as in the right-hand side of Figure 3 page 7.

2 Background

Our work is at the conjunction of two fields of study: Quality models on the one hand, design motif identification on the other hand. We present some major work in both fields of study. We show that none of existing work attempts to build a quality model based on micro-architectures of the program architectures, using design motif identification.

2.1 Quality Models

Briand and Wüst [Briand and Wüst, 2002] present a detailed and extensive survey of quality models. They classify quality models in two categories: Correlational studies and experiments. Correlational studies use univariate and multivariate analyses, while experiments use, for examples, analysis of variance between groups (ANOVA). To our best knowledge, none of the presented quality models attempts to assess the architectural quality of programs directly. They all use class-based metrics or metrics on class couples.

Wood *et al.* [Wood et al., 1999] study the structure of object-oriented C++ programs to assess the relation between program architectures and software maintenance. The authors use three different methods (structured interviews, survey, and controlled experiments) to conclude that the use of inheritance in object-oriented programs may inhibit software maintenance.

Harrison *et al.* [Harrison et al., 2000] investigate the structure of object-oriented programs to relate modifiability and understandability with levels of inheritance. Modifiability and understandability cover only partially the quality characteristics that we are interested in. Levels of inheritance are but one architectural characteristic of programs related to software maintenance.

Wydaeghe *et al.* [Wydaeghe et al., 1998] assess the quality characteristics of the architecture of an OMT editor through the study of 7 design patterns (Bridge, Chain of Responsibility, Facade, Iterator, MVC, Observer, and Visitor). They conclude on flexibility, modularity, reusability, and understandability of the architecture and of the patterns. However, they do not link their assessment with any evaluative or predictive quality model.

Peter Wendorff [Wendorff, 2001] evaluates the use of design patterns in a large commercial software product. The author concludes that design patterns do not improve a program architecture necessarily. Indeed, architecture can be over-engineered [Kerievsky, 2004] and the cost of removing design patterns is high. The author does not link is study with any quality model.

2.2 Design Motif Identification

Most approach to the identification of occurrences of design motifs are structural. They require a structural matching between a design motif and candidate micro-architectures. Different techniques have been used to perform the structural matching: Rule inference and unification [Krämer and Prechelt, 1996 ; Wuyts, 1998], queries [Ciupke, 1999 ; Keller et al., 1999], constraints resolution [Guéhéneuc and Jussien, 2001 ; Quilici et al., 1997], and fuzzy reasoning [Jahnke and Zündorf, 1997].

Unification. Wuyts developed the SOUL environment in which design motifs are described as Prolog predicates and programs constituents as facts (classes, methods, fields...) [Wuyts, 1998]. A Prolog inference algorithm unifies predicates and facts to identify classes playing roles in design motifs. The main limitation of such structural approaches is the inherent combinatorial complexity of identifying subsets of all classes matching design motifs, which corresponds to a problem of subgraph isomorphism [Eppstein, 1995].

Constraint Resolution. Quilici *et al.* used constraint programming to identify design motifs [Quilici et al., 1997]. Their approach consists in translating the problem of design motif identification in a problem of constraint satisfaction. Design motifs are described as constraint systems, which variables have for domain the entities (classes and interfaces) of a program. The resolution of the constraint systems provides micro-architectures composed of entities respecting the constraints among the roles of a design motif. As with the unification approach, the combinatorial complexity of the resolution proves to be prohibitive.

Quantitative Evaluation. Antoniol *et al.* used constraint programming extended with metrics to reduce the search space of design motif identification [Antoniol et al., 1998]. They designed a multi-stage filtering process to identify micro-architectures *identical* to design motifs. For each class of a program, they compute some metrics (for example, numbers of inheritance, of association, and of aggregation relationships) and they compare the metric values with expected values for a design motif to reduce search space. Then, they apply a constraint-based approach to identify micro-architectures. The expected values of metrics are derived from the theoretical descriptions of design motifs. The main limitation of their work lies in the assumption that implementation (micro-architectures) accurately reflects theory (design motifs), which is often not the case. Moreover, the theoretical characterisation of roles, when possible, does not reduce the search space significantly.

Fuzzy Reasoning. In an original work, Jahnke *et al.* introduced fuzzy reasoning nets to identify design motifs [Jahnke and Zündorf, 1997]. Design motifs are described as fuzzy reasoning nets, expressing rules of identification of micro-architecture similar but not identical to design motifs. They exemplify their approach with the identification of poor implementations of the Singleton design motif in legacy C++ code. They express identification rules with the formalism of fuzzy reasoning nets and then compute the certainty of a variable being a Singleton starting from a user's assumption. The main advantage of their approach is that fuzzy reasoning nets deal with inconsistent and incomplete knowledge. However, their approach requires to describe all possible rules of approximation for a design motif and user's assumption.

3 Building a Quality Model

We use design design patterns as a basis to build a quality model. Design patterns provide *good* solutions to architectural design problems, which maintainers can use in the assessment of the quality characteristics of program architectures naturally. Indeed, "[a]ll well-structured object-oriented architectures are full of patterns" [Gamma et al., 1994, page xiii]. Also, design patterns provide a basis for choosing and for organising external quality characteristics related to the maintenance effort.

3.1 Overview

The following general information offer a synthetic view on our quality model.

Dependent Variables. The dependent variables in our quality model are quality characteristics. We choose these quality characteristics by studying the quality characteristics of the 23 design patterns in Gamma *et al.*'s book [Gamma et al., 1994]. We study the literature on design patterns and identify 5 quality characteristics which decompose in 7 quality sub-characteristics which we consider as external attributes.

Independent Variables. The independent variables in our quality model are the internal attributes which we measure on programs. These internal attributes are similar to those in other quality models from the literature: Size, filiation, cohesion, coupling, and complexity.

Analysis Technique. We use a propositional rule learner algorithm, JRip. JRip is Weka—an open-source program collecting machine learning algorithms for data mining tasks [Witten and Frank, 1999]—implementation of the Ripper rule learner. It is a fast algorithm for learning "If–Then" rules. Like decision trees, rule learning algorithms are popular because the knowledge representation is easy to interpret.

3.2 Building a Quality Model

We perform the eight tasks identified in Subsection 1.3 page 4 to build a quality model considering program architectures based on design patterns.

1. Identifying the Quality Characteristics. We consider a hierarchical model, because such model is more understandable [Fenton and Pfleeger, 1997] and because most of standard models are hierarchical, for examples [ISO, 1991] and [McCall, 2001].

Design patterns claim to bring reusability, understandability, flexibility, and modularity [Gamma et al., 1994]. So, we add these quality characteristics to our quality model. Also, through our past experience, we add robustness and scalability (which define together software elegancy [for Software Engineering, 2002]) to our quality model.

2. Organising the Quality Characteristics. We organise the quality characteristics and decompose these in sub-characteristics using definitions from IEEE, ISO/IEC, and several other models, such as McCall's, Boehm's, Firesmith's [Online, 2002 ; Smith and Williams, 2001 ; Firesmith, 2003 ; Khosravi and Guéhéneuc, 2004].

Figure 5 presents our quality model to evaluate software quality related to software maintenance based on design patterns.

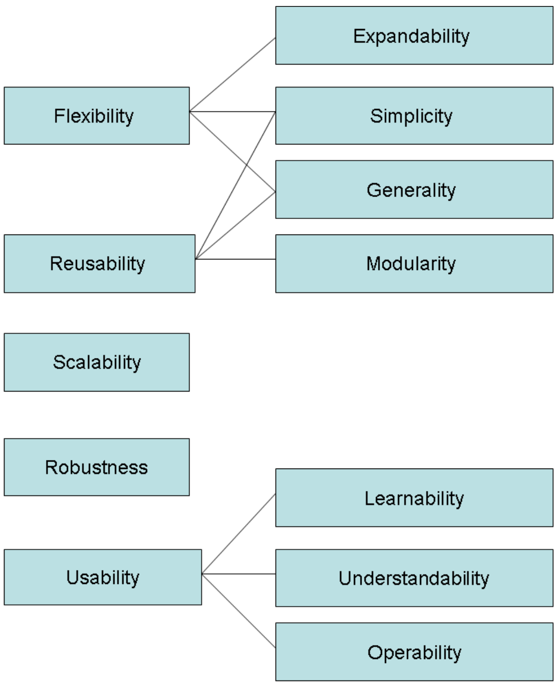

Figure 5: A quality model based on design patterns quality characteristics

3. Choosing Internal Attributes. We choose size, filiation, coupling, cohesion, and complexity as internal attributes. We use the metrics from Chidamber and Kemerer's study [Chidamber and Kemerer, 1993] mainly to measure these internal attributes, with additions from other metrics by Briand *et al.* [Briand et al., 1997a], by Hitz and Montazeri [Hitz and Montazeri, 1995], by Lorenz and Kidd [Lorenz and Kidd, 1994], and by Tegarden *et al.* [Tegarden et al., 1995].

The complete list of metrics used to measure internal attributes is: ACAIC, ACMIC, AID, CBO, CLD, cohesionAttributes, connectivity, DCAEC, DCMEC, DIT, ICHClass, LCOM1, LCOM2, LCOM5, NCM, NMA, NMI, NMO, NOA, NOC, NOD, NOP, SIX, and WMC.

4. Identifying Programs with Patterns. We use the set of programs implementing design patterns from Kuchana's book [Kuchana, 2004]. Each program of this set implements design patterns from Gamma *et al.*'s book [Gamma et al., 1994]. This set of programs forms our base programs $\mathcal{BP}$.

5. Assessing the Quality of Patterns. We assess the quality characteristics of design patterns manually, using our quality model and the set $\mathcal{BP}$. Table 1 page 13 summaries our evaluation of the quality characteristics of the twenty-three design patterns.

	Quality Sub-characteristics and Characteristics								
Design Patterns	Expendability	Simplicity	Generality	Modularity	Learnability	Understandability	Operability	Scalability	Robustness
Abs. Fact.	E	E	G	G	G	G	G	G	G
Builder	G	G	F	F	F	G	F	G	G
Fact. Met.	P	P	F	G	G	G	G	G	G
Prototype	E	G	F	G	F	G	F	E	G
Singleton	P	B	F	E	F	F	F	G	G
Adapter	F	F	P	G	G	F	F	G	F
Bridge	G	F	G	G	F	F	G	G	G
Composite	F	F	F	F	F	G	F	F	G
Decotator	E	E	G	F	G	G	G	G	F
Façade	G	G	G	G	F	G	F	F	F
Flyweight	P	P	F	G	G	P	F	G	G
Proxy	G	P	F	G	F	P	G	G	F
Chain of Res.	G	G	G	P	F	F	G	P	F
Command	G	P	F	F	P	B	G	G	G
Interpreter	G	F	G	F	F	F	G	G	F
Iterator	E	E	G	F	G	F	F	G	G
Mediator	G	F	G	G	F	F	G	G	F
Memento	G	F	F	B	P	F	G	F	P
Observer	E	G	E	F	F	G	G	G	G
State	G	G	F	P	F	B	G	G	F
Strategy	G	F	P	F	P	P	F	P	F
Tem. Met.	E	G	F	F	G	G	G	G	G
Visitor	E	G	G	F	G	P	F	G	F

Table 1: Design patterns quality characteristics in $\mathcal{BP}$ (E = Excellent, G = Good, F = Fair, P = Poor, and B = Bad)

6. Computing Metrics. The metrics we chose in task 3 to measure the internal attributes of programs are all class-based metrics. Thus, we need first to compute the metric values and second to adapt the metric values to the micro-architectures.

We analyse the programs and their micro-architectures using PADL, a meta-model to represent programs. Then, we apply POM, a framework for metrics definition and computation based on PADL [Guéhéneuc et al., 2004], on the program models to compute the metric values.

Then, we adapt the class-based metric values to the micro-architectures. For a given metric, we use the average of its values on all the classes forming a micro-architecture. However, average is not a good representative of the metric

$$\begin{array}{rc} if & (LCOM5 \leq 1.1) \wedge (NOA \leq 33.25) \\ then & (Learnability = Good) \\ else & (Learnability = Fair) \end{array}$$

Table 2: Rule for learnability

values for the micro-architecture. Indeed, we should compute and study the variance of the metric values to get a better understanding of the distribution of the metric values. Variance indicates how much each of the metric values of the classes in the micro-architecture deviates from the mean. However, for the current exploratory study, we keep the average to allow a better analysis of the resulting rules.

7. Linking Internal Attributes and Quality Characteristics. We use a machine learning technique to infer rules linking the quality characteristics of the quality model and the metric values.

We use the JRip algorithm to find the rules between quality characteristics and values of the metrics. The rule in Table 2 page 14 is the rule associated with the learnability quality characteristics, when applying JRip in the metric values and the base programs from tasks 3, 4 and 5. It shows that the learnability quality characteristics is related to the NMI and NOP metrics more than to any other metric.

We do not introduce here all the rules found for the different quality sub-characteristics and characteristics in our model for lack of space. The rules are specific to the current case study but help in illustrating the advantages and limitations of our approach.

8. Validating the Quality Model. We use the leave-one-out method [Stone, 1974] for cross-validating the rules built for our quality model by JRip.

4 Applying the Quality Model

We apply the quality model built in the previous Section 3 to the JHotDraw (for clarity, we apply our quality model on a subset of the micro-architectures in JHotDraw only), JUnit, and Lexi programs. We apply the learnability rule of the quality model in particular because this rule represents a good trade-off between simplicity and expressiveness. The learnability rule has been built in task 7 in Subsection 3.2 page 11 with $min_{LCOM5} = 0.75$, $max_{LCOM5} = 1.82$, $min_{NOA} = 1.00$, and $max_{NOA} = 86.00$.

JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed originally by Erich Gamma and Thomas Eggenschwiler as a “design exercise” but is now a full-fledge framework. Its design relies heavily on some well-known design patterns. JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used to implements unit tests in Java. Lexi is a Java-based Word Processor. It has been developed by Matthew Schmidt and Brill Pappin originally. These programs are open-source and most are available on SourceForge.

Applying our quality model requires to identify in a program the micro-architectures similar to some design motifs. We consider micro-architectures as our "unit" of measurement rather than classes, as presented in Subsection 1.4 page 6. We decompose the task 1 of *Identifying micro-architectures similar to design motifs in the architecture of a program $\mathcal{P}$ under evaluation* in two subtasks to improve the performance of the identification:

1. A task of role identification, in which we identify classes that could play a role in a design motif potentially.

2. A task of design motif identification, in which we identify classes which structural organisation is similar to that of a design motif. In this subtask, we only consider classes identified in the previous subtask to reduce the search space and thus to improve performance, recall, and precision.

4.1 Identifying Roles

We associate numerical signatures with roles in design motifs to characterise classes that *could* play one of these roles and exclude classes that *obvioulsy* could not. We seek to characterise classes playing roles in design motifs using their internal attributes. The most consensual attributes for classes in object-oriented programming languages are:

- Size/complexity, *e.g.*, number of methods, of fields.

- Filiation, *e.g.*, number of parents, number of children, depth of the inheritance tree.

- Cohesion, *e.g.*, degree to which methods and attributes of a class belong together.

- Coupling, *e.g.*, strength of the association created by a link from one class to another.

Two or more classes may have identical values for a given set of internal attributes. Indeed, two or more classes may play a same role in different uses of a design motif and a same class may play two or more roles in one or more design motifs. Thus, internal attributes cannot be used to distinguish uniquely a class among classes playing roles in design motifs.

Yet, internal attributes can be used to reduce the search space of micro-architectures. We can use internal attributes to eliminate true negatives from the search space efficiently, *i.e.*, classes that *obviously* do not play a role in a design motif. Moreover, no thorough empirical studies have so far validated

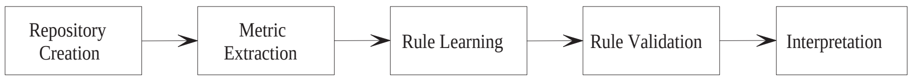

Figure 6: Process of assigning numerical signatures to design motifs roles

the impossibility to identify classes uniquely with their internal attributes, or attempted to find quantifiable *commonalities* among classes playing a given role in a design motif experimentally.

Therefore, we study the use of internal attributes of classes to quantify design motifs roles: We devise *numerical signatures* for design motifs roles using internal attributes of classes. We group these numerical signatures in rules to identify classes playing a given role. For example, a rule for the role of Singleton in the Singleton design motif could be

```
Rule for "Singleton" role:
    Filiation: Number of parents low,
               number of children low.
```

because a class playing the role of Singleton is high in the inheritance tree normally and has no (or a few) subclass usually. A rule for the role of Observer in the Observer design motif could be

```
Rule for "Observer" role:
    Coupling: Coupling with other classes low.
```

because the purpose of the Observer design motif is to reduce the coupling between the classes playing the roles of Observer and the rest of the program.

4.1.1 Building Numerical Signatures

Overview. Figure 6 presents an overview of the process of assigning numerical signature to design motifs roles. First, we build a *repository* of classes forming micro-architectures similar to design motifs in different programs. We identified the roles played by these classes in design motifs manually. Then, we *extract metrics* from the programs in which we found micro-architectures to associate a set of values for the internal attributes with each class in the repository. We feed a propositional *rule learner* algorithm with the sets of metric values. The rule learner returns a set of rules characterising design motifs roles with the metric values of the classes playing these roles. We *cross-validate* the rules using the leave-one-out method. Finally, we *interpret* the rules obtained (or the lack thereof) for roles in design motifs. The following subsections detail each step of the process.

Repository Creation. We need a repository of classes forming micro-architectures similar to design motifs to analyse these classes quantitatively. We investigate several programs manually to identify micro-architectures similar to design motifs and to build a repository of these micro-architectures, the P-MARt (Pattern-like Micro-Architecture Repository). We create this repository using different sources:

- Studies in the literature, such as the original study from James Bieman *et al.* [Bieman et al., 2003], which record classes playing roles in design motifs from several different C++, Java, and Smalltalk programs.

- Our tool suite for the identification of design motifs, Ptidej (*Pattern Trace Identification, Detection, and Enhancement in Java*) [Albin-Amiot et al., 2001 ; Guéhéneuc and Albin-Amiot, 2001], which implements JPtidejSolver, an explanation-based constraint solver to identify design motifs.

- Assignments in a undergraduate course and in a graduate course, during which students performed analyses of Java programs.

The repository of micro-architectures similar to design motifs surveys:

- For each program, motifs for which we found similar micro-architectures.

- For each motif, similar micro-architectures that we found in the program.

- For each micro-architectures, roles played by their classes in the corresponding design motif.

We validate all the micro-architectures manually before their inclusion in the repository, however we do not claim that we identified *all* micro-architectures similar to design motifs in a given program.

So far, the P-MARt contains data from 9 programs, for a total of 4,376 classes and 138 micro-architectures representing 19 different design motifs. We exclude inner classes because no inner class appears in a micro-architecture so far. Table 3 page 33 summarises the data in the P-MARt. The two first rows give the names and number of classes (and interfaces) of the surveyed programs. The following rows indicates, for a given design pattern (per row), the number of micro-architectures found similar to its design motif in each program (per column). The table summarises also the number of roles defined by a design motif and the number of classes playing a role in a design motif for all the programs (two last columns). The number of classes playing roles in design motifs shows that only a fraction of all the classes of the programs plays a role in a design motif. Moreover, some classes are counted more than once because they play different roles in different design motifs. Design motifs for which we did not identify similar micro-architectures are: Chain of Responsibility, Interpreter, and Mediator. We record this data in a XML file, which allows us to traverse the data to compute metrics and various statistics automatically.

Metric Extraction. We parse the programs surveyed in the P-MARt and calculate metrics on their classes automatically. Parsing and calculation are performed in a three-step process: First, we build a model of a program using the PADL (*Pattern and Abstract-level Description Language*) meta-model and its parsers; Second, we compute metrics using POM (*Primitives, Operators, Metrics*), an extensible framework for metric calculation based on PADL; Third, we store the results of the metric calculation, names and values, in the P-MARt, by adding specific attributes and nodes to the XML tree representation.

We use metrics from the literature to associate values with internal attributes of classes playing a role in a design motif. Table 4 page 34 presents the metrics computed on classes related to the internal attributes that we consider: Size/complexity, filiation, cohesion, and coupling. For size/complexity, we use the metrics by Lorenz and Kidd on new, inherited, and overridden methods and on the total number of methods [Lorenz and Kidd, 1994], and the count of methods weighted with their numbers of method invocations by Chidamber and Kemerer [Chidamber and Kemerer, 1993]. We do not use metrics related to fields because no design motif role is characterised by fields specifically: Only the Flyweight, Memento, Observer, and Singleton design motifs (5 out of 23) expose the internal structures of some roles to exemplify typical implementation choices. Moreover, fields should always be private to their classes with respect to the principle of encapsulation. For filiation, we use the depth in the inheritance tree and the number of children by Chidamber and Kemerer [Chidamber and Kemerer, 1993] and the number of hierarchical levels below a class, class-to-leaf depth, by Tegarden *et al.* [Tegarden et al., 1995]. For cohesion, we use the metric 'C' measuring the connectivity of a class with the rest of a program by Hitz and Montazeri [Hitz and Montazeri, 1995] and the fifth metric on lack of cohesion in methods by Briand *et al.* [Briand et al., 1997b]. Finally, for coupling, we use two metrics on class-method import and export coupling by Briand *et al.* [Briand et al., 1997a] and the metric on coupling between objects by Chidamber and Kemerer [Chidamber and Kemerer, 1993].

Rule Learning and Validation. The P-MARt contains a wealth of data to analyse. We use a machine learning algorithm to find commonalities among classes playing a same role in a design motif. We supply the data to a propositional rule learner algorithm, JRip, implemented in Weka, an open-source program collecting machine learning algorithms for data mining tasks [Witten and Frank, 1999].

We do not provide JRip with all the data in the P-MARt, this would lead to uninteresting results because of the disparities among roles, classes, and metric values. We provide JRip with subsets of the data related to each role. A subset σ of the data related to a role contains the metric values for the n classes playing the role in all the micro-architectures similar to a design motif. We add to this subset σ the metric values of $3 \times n$ classes *not* playing the role, chosen randomly in the rest of the data. We make sure the classes chosen randomly have the expected structure for the role, *i.e.*, whether the role is defined to be played by a class or by an abstract class [Gamma et al., 1994], to increase their likeliness with the classes playing the role. The rule learner infers rules related to each role from the subsets σ. We validate the rules using the leave-one-out method with each set of metric values in the subsets σ [Stone, 1974].

Interpretation. The rule learner infers rules that express the experimental relationships among metric values, on the one hand, and roles in design motifs, on the other hand. Typically, a rule inferred by the rule learner for a role `ROLE` has the form

```
Rule for "ROLE" role:
    - Numerical signature 1, confidence 1,
    - Numerical signature 2, confidence 2,
    - ...
    - Numerical signature N, confidence N.
```

where

`Numerical signature 1` $= \{metric_1 \in V_{11}, \ldots, metric_m \in V_{m1}\}$

$\ldots$

`Numerical signature N` $= \{metric_1 \in V_{1n}, \ldots, metric_m \in V_{mn}\}$

and the values of a metric $metric_i$ computed on classes playing the role `ROLE` belong to a set $V_{ij} \subset \mathbb{N}$. The degree of confidence `confidence K` is the proportion of classes concerned by a numerical signature in a subset σ, which we use to compute error and recall ratios.

We collect all the rules inferred from the rule learner and process the rules with the following criteria to remove uncharacteristic rules:

- We remove rules with a recall ratio less than 75%.
- We remove rules inferred from small subsets σ, *i.e.*, when not enough classes play a given role.

Then, we interpret the remaining rules in two ways: Qualitatively, we explain rules with respect to their corresponding roles; Quantitatively, we assess the quality of classes playing roles in design motifs. Practically, we show that numerical signatures reduce the search space for micro-architectures similar to design motifs efficiently.

4.1.2 Discussion

We decompose the data in the P-MARt in 56 subsets σ and infer as many rules with the rule learner, which decompose in 78 numerical signatures. The two first steps in the analysis process are quantitative and aim at eliminating roles that do not have a sufficient number of examples for mining numerical signatures and that do not have a high enough recall ratio. In the first step, we remove 20 over the 56 rules from all the rules inferred by the rule learner. The removed rules corresponds to:

```
Rule "Target"
    - WMC <= 2, 24/25.
```

Figure 7: Rules inferred for the role of Target in the Adapter design motif

- Design motifs roles with few corresponding micro-architectures and with a unique (or a few) classes in the micro-architectures. Some examples are the roles of Decorator in the Decorator design motif and of Prototype in Prototype.
- Design motifs roles played by "ghost" classes in many cases, *i.e.*, classes known only from import references, such classes in standard libraries. Some examples are the classes playing the roles of Command in the Command design motif and of Builder in Builder.

In the second step, we select 20 rules with a recall ratio greater than 75%, shown in Table 5 page 34, from the 36 remaining rules. All these rules have an error rate less than 10% (less than 5% for 16). Most of the rules removed because of their low recall ratio are associated with roles known to be non-key roles in design motifs and thus do not have a particular numerical signature theoretically. For example, any class may play the role of Client in the Composite design motif. Similarly, any class may play the role of Invoker in the Command design motif. (Some researchers argue that Client, Invoker... are not "real" roles and should not appear in most design motifs.)

We notice that in many cases we obtain a unique numerical signature for a given role in a design motif. Classes playing a same role have similar structures and organisations generally. For example, all the classes playing the role of Target in the Adapter design motif have a low complexity, represented by low values of WMC, as shown in Figure 7 (the degree of confidence is less than 1 because this numerical signature misclassifies one class, its error rate is 4%, as shown in Table 5 page 34). Such a low complexity is actually expected because of the architecture and of the behaviour suggested by the Adapter design motif. Likewise, many other numerical signatures confirm claims from and beliefs on design motifs. For examples, classes playing the role of Observer in the Observer design motif have a low coupling, *i.e.*, a low CBO. Classes playing the roles of Singleton in the Singleton design motif have low coupling and belong to the upper part of the inheritance tree generally.

In few cases, we obtain more that one numerical signature for a role. An example is the role of Concrete Visitor in the Visitor design motif. On the one hand, the most frequent numerical signature is characteristic of classes with a low coupling (low CBO) and a large number of methods (high NM), as expected from the problem dealt with by the Visitor design pattern. On the other hand, the second numerical signature states that the number of inherited methods is low (low NMI) for some classes playing the role of Concrete Visitor. When exploring the micro-architectures similar to the Visitor design motif in our repository, we notice that in JREFACTORY some classes play the roles of both Concrete Visitor and Visitor, which then limits the number of inherited methods. This second numerical signature is particular to the program and thus unveils design choices specific to the program or to a coding style.

4.2 Identifying Design Motif

After identifying classes which could play roles in design motifs, we perform a structural search among these classes to identify those which structures and organisations is similar to the structures and organisations advocated by some design motifs.

We use explanation-based constraint programming to identify both complete and approximate forms of design motifs, *i.e.*, groups of classes which structures and organisations are similar to the motifs, while providing explanations and allowing user-interactions.

4.2.1 Explanation-based Constraint Programming

Explanation-based constraint programming proved its interest [Jussien and Barichard, 2000] in many applications already. We recall fundamentals on explanation-based constraint programming and some of its uses.

Contradiction Explanations. We consider a constraint satisfaction problem (CSP) (V, D, C): V is the set of variables, D is the set of domains for the variables, and C is the set of constraints among variables. Decisions made during enumeration—variable assignments—are represented by unary constraints added to or removed from the current constraint system. These unary constraints are called *decision constraints* because they are not defined in the constraint satisfaction problem but are generated by the solver to represent decisions taken during the resolution.

A *contradiction explanation* (also know as *nogood* [Schiex and Verfaillie, 1994]) is a subset of the current constraint system that, left alone, leads to a contradiction—no solution. A contradiction explanation divides in two parts: A subset of the original set of constraints ($C' \subset C$ in Equation 1) and a subset of the decision constraints introduced during the search.

$$C \vdash \neg (C' \wedge v_1 = a_1 \wedge ... \wedge v_k = a_k) \tag{1}$$

A contradiction explanation without decision constraint denotes an over-constrained problem. In a contradiction explanation containing at least one decision constraint, we choose a variable v_j and rewrite Equation 1 in 2.

$$C \vdash C' \wedge \bigwedge_{i \in [1..k] \backslash j} (v_i = a_i) \rightarrow v_j \neq a_j \tag{2}$$

The left hand side of the implication is an *eliminating explanation* for the removal of value a_j from the domain of variable v_j. The eliminating explanation is noted:

$$\texttt{expl}(v_j \neq a_j)$$

Classical solvers use domain-reduction techniques to solve constraint satisfaction problems by removing values from the domains of variables. Thus, recording eliminating explanations is sufficient to compute contradiction explanations. Indeed, a contradiction is identified when the solver empties the domain of a variable v_j. A contradiction explanation can be computed with the eliminating explanations associated with each removed value, as shown in Equation 3.

$$C \vdash \neg \left(\bigwedge_{a \in d(v_j)} \texttt{expl}(v_j \neq a) \right) \tag{3}$$

Several eliminating explanations exist for the removal of a given value generally. Recording all eliminating explanations would lead to an exponential space complexity. Thus, we must *forget* (erase) eliminating explanations that are no longer relevant to the current variable assignments. An eliminating explanations is said to be relevant if all its decision constraints are valid in the current search state [Bayardo Jr. and Miranker, 1996]. We keep only *one* explanation at a time for any value removal and the space complexity remains polynomial.

Computing Contradiction Explanations. Minimal contradiction explanations (with respect to inclusion) are the most interesting. They provide precise information on dependencies among variables and constraints identified during the search. Unfortunately, computing such explanations is time-consuming [Junker, 2001]. A compromise between size and computability consists in using the knowledge *inside* the solver. Indeed, CSP solvers always know why they remove values from the domains of variables, although not often explicitly. They can compute minimal contradiction explanations with this knowledge explicitly. We must alter the source code of a CSP solver to make such knowledge explicit. The PaLM solver [Jussien and Barichard, 2000] is a reference implementation of an explanation-based constraint solver.

Using Contradiction Explanations. We can use contradiction explanations for many different purposes [Jussien and Barichard, 2000 ; Jussien et al., 2000 ; Jussien and Lhomme, 2000]. For example, we can use explanations to debug resolution by explaining contradictions *clearly* and by explaining differences between intended and observed behaviour (answering question such as "why is x not assigned to value 4?" explicitly).

Also, we can use contradiction explanations to assess the impact of a constraint on domains of variables and to handle, for example, dynamic constraint removal. Thus, Bessière's justification system [Bessière, 1991] for solving dynamic CSP is a partial explanation-based constraint solver actually.

Finally, we can use contradiction explanations to improve standard backtracking algorithms and to improve the search: To provide intelligent backtracking [Guéret et al., 2000], to replace standard backtracking with jump-based approaches *à la* dynamic backtracking [Ginsberg, 1993 ; Jussien et al., 2000], to develop new local searches on partial instantiations [Jussien and Lhomme, 2000], to guide the search dynamically.

4.2.2 Application to Design Motif Identification

Design motif identification consists:

1. In modelling a set of design motifs as CSP. A variable is associated with each class defined by a design motif. The variables of our model are integer-valued. The domain of a variable is a set of existing classes in the source code. Each class is identified by a unique integer. Relationships among classes (inheritance, association...) are represented by constraints among variables.

2. In modelling the maintainers' source code to keep only the information needed to apply the constraints: Class names—forming the domain of the variables and the relationships among classes—verifying or not the constraints.

3. In resolving the CSP to search both approximate and complete micro-architectures: When all solutions to the CSP are found, *i.e.*,, when all micro-architectures identical to a design motif are identified, the search is guided by the maintainers to find approximate micro-architectures dynamically. Contradiction explanations provided by the constraint solver help the maintainers in guiding the search.

We build a library of specialised constraints from the relationships among classes used to describe design motifs [Gamma et al., 1994]. Specialised constraints express the inheritance, creation, association... relationships among classes. Our library offers constraints covering a broad range of design motifs. However, some design motifs are difficult to express as CSP and require additional relationships or the decomposition of existing relationships into sub-relationships. We provide the following constraints:

- *Strict inheritance* establishes a strict inheritance relationship between two classes. A strict inheritance relationship links two classes in a parent–child-like relationship, *i.e.*, superclass–subclass. When considering single inheritance, the strict inheritance relationship is a partial order, denoted $<$, on the set of classes E. For any pair of distinct classes `A` and `B` in E, if `B` inherits from `A` then: `A` $<$ `B`. The constraint associated with the strict inheritance relationship is a binary constraint between variables `A` and `B`. The operational semantics of the constraint is

$$\forall C_A \in d_A, \exists c_B \in d_B, c_A < c_B$$

$$\forall c_B \in d_B, \exists c_A \in d_A, c_A < c_B$$

where d_X represents the domains of variable X. From this definition of strict inheritance, we derive an inheritance relationship, and its associated constraint, such that the variables may represent a same class: $\mathtt{A} < \mathtt{B}$ or $\mathtt{A} = \mathtt{B}$.

- *Use* establishes a use relationship [Guéhéneuc, 2004] between classes. A class A knows about a class B if methods defined in A invoke methods of B. This relationship is binary, oriented and intransitive. We denote this relationship by $\mathtt{A} \rhd \mathtt{B}$.

- *Association*, *Aggregation*, and *Composition* enforce that two classes are associated, aggregated, or composed with one another [Guéhéneuc and Albin-Amiot, 2004], respectively. For example, a class A is composed with instances of a class B if the A class defines one or more fields of type B. We write $\mathtt{A} \supset \mathtt{B}$. This relationship is binary, oriented and intransitive.

4.2.3 Behaviour of the CSP Solver

The library of specialised constraints is not sufficient in itself to allow design motif identification. Indeed, micro-architectures that fit exactly in the modelling of a design motif as a CSP are of no use to identify area with poor quality characteristics. We need to find *approximate* micro-architectures, *i.e.*, micro-architectures that do not verify all constraints from a design motif. Explanation-based constraint programming allows to identify approximate and complete forms.

First, a specialised CSP solver computes complete forms. The resolution ends by a contradiction, there is no more micro-architectures. Explanation-based constraint programming provides a contradiction explanation for this contradiction: The set of constraints justifying that other combinations of classes do not verify the constraints describing the searched design motif. We do not need to relax other constraints than the constraints provided by the contradiction explanation: We would find no other micro-architecture. The explanation contradiction provides knowledge on which approximate forms are available. This knowledge about possible approximate forms allows maintainers to lead the search towards interesting approximate forms, from their viewpoints, by exhibiting constraints to relax. Removing a constraint suggested by a contradiction explanation does not necessarily lead to new micro-architectures but the removal is applied recursively.

Preferences are assigned to the constraints of a CSP to ease maintainers' interactions with the specialise CSP solver. They reflect a hierarchy among constraints *a priori*, but this is not mandatory in our CSP solver. We derive a metric from the preferences, with which we measure the quality of micro-architecture in terms of its distance with the search design motif, *i.e.*,, the number of constraints relaxed to obtain this micro-architecture. The metric allows the automation of the CSP solver to identify all approximate micro-architectures.

The maintainer-driven version of our CSP solver is of great interest when *a priori* preferences are hard to determine, which is often the case. Moreover, maintainers can restrict the search to a subset of interesting approximate forms interactively. Explanation-based constraint programming gives a complete control to the maintainers: This is important in an *intellectual* activity such as design motif identification.

4.2.4 Discussion

The use of explanation-based constraint programming to identify micro-architectures similar to design motifs provides two interesting properties:

- Identification of both complete and approximate forms of design motifs.
- Explanations about the identified micro-architectures and interactions with the maintainers.

We describe design motifs as constraint systems: Each role is represented as a variable and relationships among roles are represented as constraints among variables. Variables had identical domains: All the classes in a program in which to identify design motifs. For example, the identification of micro-architectures similar to the Composite design motif, shown in Figure 8, in JHotDraw translates to the constraint system

```
Variables:
    client
    component
    composite
    leaf
Constraints:
    association(client, component)
    inheritance(component, composite)
    inheritance(component, leaf)
    composition(composite, component)
```

where the four variables `client`, `component`, `composite`, and `leaf` have identical domains, which contains all the 155 classes (and interfaces) composing JHotDraw, and the four constraints represent the association, inheritance, and composition relationships suggested by the Composite design motif.

However, as other structural approaches, our approach with explanation-based constraint has limited performance and a bad recall. Indeed, the Composite design motif describe fur roles, which are express as four variables. Thus, the search of micro-architectures similar to the Composite design motif in the JHotDraw framework, which contain 155 classes, has potentially $155^4 = 577,200,625$ solutions.

We introduce numerical signatures associated with roles in design motifs to reduce the search space and improve both performance and recall.

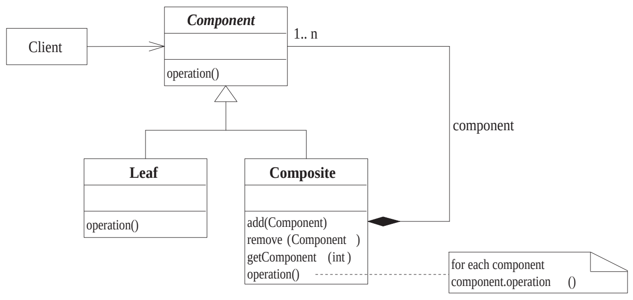

Figure 8: Composite design motif

4.3 Applying the Quality Model

We have now identified micro-architectures similar design motifs. We follow the four tasks from Subsection 1.4 page 6 to apply the quality model on these micro-architectures.

1. Identifying Micro-Architectures. JHotDraw uses 11 different design patterns in 21 micro-architectures: Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, and Template Method. JUnit contains 8 micro-architectures similar to 5 different design patterns: Composite, Decorator, Iterator, Observers, and Singletons. Lexi contains 5 micro-architectures similar to the Builder, Observer, and Singleton design patterns. Table 6 page 35 summarises the micro-architectures.

2. Measuring Internal Attributes. For each micro-architecture identified in JHotDraw or in JUnit or in Lexi, we use PADL and POM to compute the class-based metric values and Etiquette to compute the micro-architecture-based metric values (using average). Table 6 presents the data for each micro-architecture for the LCOM5 and NOA metrics.

3. Adapting the Rules. We adapt the metric values in the rule in Table 2 by computing the ratio between the minimum and maximin values of the LCOM5 and NOA metrics for the base programs on the one hand, and each micro-architecture on the other hand. Table 6 also displays the adapted rules for all the micro-architectures.

4. Applying the Rules. We compare the expected metric values in the adapted rules with the metric values computed for each micro-architecture.

Discussion. Table 6 presents the results for adapting the learnability rule in Table 2. We computed the average, the minimum, and the maximum values of the LCOM5 and NOA metrics for each program, JUnit and Lexi. We adapted the rule from the minimum and maximum values of the base programs and of JHotDraw, JUnit, and Lexi. The last column shows the adapted rules and the results of applying the rules.

The first line of the table shows an example of applying the learnability rule to a micro-architecture similar to the Command design pattern. The outcome of the rule states that this particular implementation of the Command design pattern has a Good learnability.

However, the quality model obtained is unsatisfactory for many reasons. First, the size of the base programs used to create the quality model renders the rule uninteresting in many cases. In particular, we do not have sufficient data yet but the assess the learnability of JUnit and of Lexi as Fair.

Second, adapting the rule when there is one metric value only, see for example the micro-architecture MA5 in JUnit, does not provide interesting information because the adapted threshold of the learnability rule is always inferior to the maximum (and unique) value. Adaptation requires a range or more accurate rules (based on a minimum and a maximum thresholds) to be efficient.

Third, we do not distinguish in the micro-architecture between code which plays a role in the design pattern and code which does not. Considering all the metric values, potentially for "dead" code, has an impact on the results certainly.

Moreover, learnability is a human-related quality characteristic. Thus, it is difficult to assess intrinsically because it depends on the individuals performing the assessment highly. Thus, we need to perform more evaluations to obtain an accurate rule.

5 Conclusion and Future Trends

In this chapter, we presented a global coverage of our work on software quality models and on design motif identification. We use design motif as laws on software quality and we used these laws to observe micro-architectures similar to design motifs and to assess their quality.

Using design motif as laws, we described the process of building a quality model using motifs to link metrics and quality characteristics with a learning algorithm. We also described the process of applying the quality model on software, using explanation-based contraint programming to identify micro-architectures similar to design motifs and metrics to improve the performance, recall, and precision of the identification.

This chapter highlights the need for *principles* in software engineering. These principles can be laws or theories formalising, and explaining observations realised on software. Our contribution is the use of design patterns and their solutions, design motifs, as laws on software quality to build quality models.

In the future, we believe that the software engineering community must develop its understanding of design patterns and their applications to solve problems, such as traceability, maintainability. Indeed, patterns are more and more recognised as important concepts in software engineering, in particular in conjunction with cognition during software development [Floyd, 1992 ; Miller, 1956], and they both require and deserve a more thorough and systematic study.

Also, the software engineering community must strive to identify concepts on which to build laws and theories. We strongly believe that design motifs can be considered as laws of quality. However, our belief requires further studies and analyses.

Acknowledgements

We thank James Bieman, Greg Straw, Huxia Wang, P. Willard, and Roger T. Alexander [Bieman et al., 2003] for kindly sharing their data. We are grateful to our students, Saliha Bouden, Janice Ka-Yee Ng, Nawfal Chraibi, Duc-Loc Huynh, and Taleb Ikbal, who helped in the creation of the repository. "Femme Profile" by Pablo Picasso is from `rogallery.com`, we are currently in contact with its director regarding copyrights.

All the data and programs used to perform the case study are available on the Internet at `www.iro.umontreal.ca/~labgelo/p-mart/`.

References

[ISO, 1991] (1991). *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use.* ISO/IEC. ISO/IEC 9126:1991(E).

[Albin-Amiot et al., 2001] Albin-Amiot, H., Cointe, P., Guéhéneuc, Y.-G., and Jussien, N. (2001). Instantiating and detecting design patterns: Putting bits and pieces together. In Richardson, D., Feather, M., and Goedicke, M., editors, *proceedings of the* 16th *conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press.

[Antoniol et al., 1998] Antoniol, G., Fiutem, R., and Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. In Tilley, S. and Visaggio, G., editors, *proceedings of the* 6th *International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press.

[Bayardo Jr. and Miranker, 1996] Bayardo Jr., R. J. and Miranker, D. P. (1996). A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In Weld, D. and Clancey, B., editors, *proceedings of the* 13th *national conference on artificial intelligence*, pages 298–304. AAAI Press / The MIT Press.

[Bessière, 1991] Bessière, C. (1991). Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91.*

[Bieman et al., 2003] Bieman, J., Straw, G., Wang, H., Willard, P., and Alexander, R. T. (2003). Design patterns and change proneness: An examination of five evolving systems. In Berry, M. and Harrison, W., editors, *proceedings of the* 9th *international Software Metrics Symposium*, pages 40–49. IEEE Computer Society Press.

[Briand et al., 1997a] Briand, L., Devanbu, P., and Melo, W. (1997a). An investigation into coupling measures for C++. In Adrion, W. R., editor, *proceedings of the* 19th *International Conference on Software Engineering*, pages 412–421. ACM Press.

[Briand et al., 1997b] Briand, L. C., Daly, J. W., and Wüst, J. K. (1997b). A unified framework for cohesion measurement. In Pfleeger, S. L. and Ott, L., editors, *proceedings of the* 4th *international Software Metrics Symposium*, pages 43–53. IEEE Computer Society Press.

[Briand and Wüst, 2002] Briand, L. C. and Wüst, J. (2002). Empirical studies of quality models in object-oriented systems. *in Advances in Computers*, 59.

[Chidamber and Kemerer, 1993] Chidamber, S. R. and Kemerer, C. F. (1993). A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management.

[Ciupke, 1999] Ciupke, O. (1999). Automatic detection of design problems in object-oriented reengineering. In Firesmith, D., editor, *proceeding of* 30th *conference on Technology of Object-Oriented Languages and Systems*, pages 18–32. IEEE Computer Society Press.

[Endres and Rombach, 2003] Endres, A. and Rombach, D. (2003). *A Handbook of Software and Systems Engineering.* Addison-Wesley, 1st edition.

[Eppstein, 1995] Eppstein, D. (1995). Subgraph isomorphism in planar graphs and related problems. In Clarkson, K., editor, *proceedings of the* 6th *annual Symposium On Discrete Algorithms*, pages 632–640. ACM Press.

[Fenton and Pfleeger, 1997] Fenton, N. E. and Pfleeger, S. L. (1997). *Software Metrics A Rigorous and Practical Approach.* PWS Publishing Company, 2nd edition.

[Firesmith, 2003] Firesmith, D. G. (2003). Common concepts underlying safety, security, and survivability engineering. *Carnegie Mellon Software Engineering Institute - Technical note CMU/SEI-2003-TN-033.*

[Floyd, 1992] Floyd, C. (1992). *Human Questions in Computer Science*, chapter 1, pages 15–27. Springer Verlag.

[for Software Engineering, 2002] for Software Engineering, C. (2002). *OO Analysis and Design: Modeling, Integration, Abstraction.* http://sunset.usc.edu/classes/ cs577b_2002/EC/03/EC-03.ppt.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1st edition.

[Ginsberg, 1993] Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46.

[Guéhéneuc, 2004] Guéhéneuc, Y.-G. (2004). A reverse engineering tool for precise class diagrams. In Singer, J. and Lutfiyya, H., editors, *proceedings of the* 14th *IBM Centers for Advanced Studies Conference*. ACM Press.

[Guéhéneuc and Albin-Amiot, 2001] Guéhéneuc, Y.-G. and Albin-Amiot, H. (2001). Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Li, Q., Riehle, R., Pour, G., and Meyer, B., editors, *proceedings of the* 39th *conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press.

[Guéhéneuc and Albin-Amiot, 2004] Guéhéneuc, Y.-G. and Albin-Amiot, H. (2004). Recovering binary class relationships: Putting icing on the UML cake. In Schmidt, D. C., editor, *proceedings of the* 19th *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press.

[Guéhéneuc and Jussien, 2001] Guéhéneuc, Y.-G. and Jussien, N. (2001). Using explanations for design-patterns identification. In Bessière, C., editor, *proceedings of the* 1st *IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press.

[Guéhéneuc et al., 2004] Guéhéneuc, Y.-G., Sahraoui, H., and Zaidi, F. (2004). Fingerprinting design patterns. In Stroulia, E. and de Lucia, A., editors, *proceedings of the* 11th *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press.

[Guéret et al., 2000] Guéret, C., Jussien, N., and Prins, C. (2000). Using intelligent backtracking to improve branch and bound methods: An application to open-shop problems. *European Journal of Operational Research*, 127(2):344–354.

[Harrison et al., 2000] Harrison, R., Counsell, S. J., and Nithi, R. V. (2000). Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *journal of Systems and Software*, 52(2–3).

[Hitz and Montazeri, 1995] Hitz, M. and Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *proceedings of the* 3rd *International Symposium on Applied Corporate Computing*, pages 25–27. Texas A & M University.

[Jahnke and Zündorf, 1997] Jahnke, J. H. and Zündorf, A. (1997). Rewriting poor design patterns by good design patterns. In Demeyer, S. and Gall, H. C., editors, *proceedings the* 1st *ESEC/FSE workshop on Object-Oriented Reengineering*. Distributed Systems Group, Technical University of Vienna. TUV-1841-97-10.

[Junker, 2001] Junker, U. (2001). QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. Technical report, Ilog SA.

[Jussien and Barichard, 2000] Jussien, N. and Barichard, V. (2000). The PaLM system: Explanation-based constraint programming. In Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T., Perron, L., and Schulte, C., editors, *Proceedings of TRICS: Techniques foR Implementing Constraint Programming Systems*, pages 118–133. School of Computing, National University of Singapore, Singapore. TRA9/00.

[Jussien et al., 2000] Jussien, N., Debruyne, R., and Boizumault, P. (2000). Maintaining arc-consistency within dynamic backtracking. In Dechter, R., editor, *proceedings of the* 6^{th} *conference on principles and practice of Constraint Programming*, pages 249–261. Springer-Verlag.

[Jussien and Lhomme, 2000] Jussien, N. and Lhomme, O. (2000). Local search with constraint propagation and conflict-based heuristics. *Proceedings of AAAI*, pages 169–174.

[Keller et al., 1999] Keller, R. K., Schauer, R., Robitaille, S., and Pagé, P. (1999). Pattern-based reverse-engineering of design components. In Garlan, D. and Kramer, J., editors, *proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 226–235. ACM Press.

[Kerievsky, 2004] Kerievsky, J. (2004). *Refactoring to Patterns.* Addison Wesley Professional, 1^{st} edition.

[Khosravi and Guéhéneuc, 2004] Khosravi, K. and Guéhéneuc, Y.-G. (2004). A quality model for design patterns. Technical Report 1249, Université de Montréal.

[Krämer and Prechelt, 1996] Krämer, C. and Prechelt, L. (1996). Design recovery by automated search for structural design patterns in object-oriented software. In Wills, L. M. and Baxter, I., editors, *proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press.

[Kuchana, 2004] Kuchana, P. (2004). *Software Architecture Design Patterns in Java.* Auerbach Publications, 1^{st} edition.

[Lorenz and Kidd, 1994] Lorenz, M. and Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Approach.* Prentice-Hall, 1^{st} edition.

[McCall, 2001] McCall, J. A. (2001). Quality factors. *Encyclopedia of Software Engineering*, 1-2:958–ff.

[Miller, 1956] Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97.

[Online, 2002] Online, C. (2002). Scalability from the edge. *Computer Business review Online, CBR Online.*

[Quilici et al., 1997] Quilici, A., Yang, Q., and Woods, S. (1997). Applying plan recognition algorithms to program understanding. *journal of Automated Software Engineering*, 5(3):347–372.

[Schiex and Verfaillie, 1994] Schiex, T. and Verfaillie, G. (1994). Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207.

[Smith and Williams, 2001] Smith, C. U. and Williams, L. G. (2001). *Introduction to Software Performance Engineering*, chapter 1. Addison Wesley. http://www.awprofessional.com/articles/article.asp?p=24009.

[Stone, 1974] Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society*, 36:111–147. Series B: Statistical Methodology.

[Tegarden et al., 1995] Tegarden, D. P., Sheetz, S. D., and Monarchi, D. E. (1995). A software complexity model of object-oriented systems. *Decision Support Systems*, 13(3–4):241–262.

[Tran et al., 2000] Tran, J. B., Godfrey, M. W., Lee, E. H., and Holt, R. C. (2000). Architectural repair of open source software. In *proceedings of the* 8^{th} *International Workshop on Program Comprehension*, pages 48–57. IEEE Computer Society Press.

[Wendorff, 2001] Wendorff, P. (2001). Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Sousa, P. and Ebert, J., editors, *proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press.

[Witten and Frank, 1999] Witten, I. H. and Frank, E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1^{st} edition.

[Wood et al., 1999] Wood, M., Daly, J., Miller, J., and Roper, M. (1999). Multimethod research: An empirical investigation of object-oriented technology. *journal of Systems and Software*, 48(1).

[Wuyts, 1998] Wuyts, R. (1998). Declarative reasoning about the structure of object-oriented systems. In Gil, J., editor, *proceedings of the* 26^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press.

[Wydaeghe et al., 1998] Wydaeghe, B., Verschaeve, K., Michiels, B., Damme, B. V., Arckens, E., and Jonckers, V. (1998). Building an omt-editor using design patterns: An experience report. In *proceedings of the* 26^{th} *Technology of Object-Oriented Languages and Systems conference*, pages 20–32. IEEE Computer Society Press. citeseer.ist.psu.edu/wydaeghe98building.html.

	JHotDraw v5.1	JRefactory v2.6.24	JUnit v3.7	Lexi v0.0.1α	MapperXML v1.9	Nutch v0.4	PMD v1.8	NetBeans v1.0.x	QuickUML 2001	Total	Number of roles [Gamma et al., 1994]	Number of classes playing a role
Number of classes	155	569	79	217	172	2,558	447	24	155	4,376		
Design motifs	Number of micro-architectures											
Abstract Factory					1	12			1	14	5	242
Adapter	1	17			2	8	2	1		31	4	252
Bridge							2			2	4	25
Builder		2		1				2	1	6	4	44
Command	1					1	2		1	5	5	85
Composite	1		1		1			2	2	7	4	147
Decorator	1		1							2	4	64
Facade					1					1	2	11
Factory Method	3	1			1			3		8	4	111
Iterator			1			5	1	1		8	5	41
Memento								2		2	3	15
Observer	2		3	2	1			2	1	11	4	135
Prototype	2									2	3	32
Proxy								1		1	3	3
Singleton	2	2	2	2	3		1		1	13	1	13
State	2	2								4	3	32
Strategy	4				1		2			7	3	47
Template Method	2				4		3	1		10	2	102
Visitor		2						1		3	5	143
								Total		138	70	1552

Table 3: Overview of the data set: Programs, design motifs, micro-architectures, and roles

	Acronyms	Descriptions and references	
Size/complexity	NM	Number of methods	[Lorenz and Kidd, 1994]
	NMA	Number of new methods	[Lorenz and Kidd, 1994]
	NMI	Number of inherited methods	[Lorenz and Kidd, 1994]
	NMO	Number of overridden methods	[Lorenz and Kidd, 1994]
	WMC	Weighted methods count	[Chidamber and Kemerer, 1993]
Filiation	CLD	Class-to-leaf depth	[Tegarden et al., 1995]
	DIT	Depth in inheritance tree	[Chidamber and Kemerer, 1993]
	NOC	Number of children	[Chidamber and Kemerer, 1993]
Cohesion	C	Connectivity 'C'	[Hitz and Montazeri, 1995]
	LCOM5	Lack of cohesion in methods 5	[Briand et al., 1997b]
Coupling	ACMIC	Ancestors class-method import	[Briand et al., 1997a]
	CBO	Coupling between object	[Chidamber and Kemerer, 1993]
	DCMEC	Descendants class-method export	[Briand et al., 1997a]

Table 4: External attributes for classes and corresponding metrics

Design motifs	Roles	Error (%)	Recall (%)
Iterator	Client	0.00	100.00
Observer	Subject	0.00	100.00
Observer	Observer	2.38	100.00
Template Method	Concrete Class	0.00	97.06
Prototype	Concrete Prototype	0.00	96.30
Decorator	Concrete Component	4.17	89.58
Visitor	Concrete Visitor	0.00	88.89
Strategy	Context	3.70	88.89
Visitor	Concrete Element	2.04	88.78
Singleton	Singleton	8.33	87.50
Factory Method	Concrete Creator	4.30	87.10
Factory Method	Concrete Product	3.45	86.21
Adapter	Target	4.00	84.00
Composite	Leaf	6.47	82.09
Decorator	Concrete Decorator	0.00	80.00
Iterator	Iterator	0.00	80.00
Command	Receiver	6.67	80.00
State	Concrete State	6.67	80.00
Strategy	Concrete Strategy	2.38	78.57
Command	Concrete Command	3.23	77.42

Table 5: Roles with inferred rules with recall ratio greater than 75%

Micro-Architectures	Design Patterns	$LCOM5$	min_{LCOM5}	max_{LCOM5}	NOA	min_{NOA}	max_{NOA}	Rule for learnability
Subset of the micro-architectures in JHotDraw								
MA74	Command	1.07	0.50	1.63	29.35	1.00	164.00	$(LCOM5 \leq 1.16) \wedge (NOA \leq 62.30) \Rightarrow Good$
MA85	Singleton	0.67	0.67	0.67	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA91	Strategy	0.95	0.80	1.0	553.88	221.00	792.00	$(LCOM5 \leq 0.21) \wedge (NOA \leq 218.23) \Rightarrow Fair$
JUnit								
MA65	Composite	0.65	0.25	0.95	70.10	4.00	148.00	$(LCOM5 \leq 0.72) \wedge (NOA \leq 56.33) \Rightarrow Fair$
MA66	Decorator	0.65	0.25	0.90	135.41	49.00	176.00	$(LCOM5 \leq 0.67) \wedge (NOA \leq 49.68) \Rightarrow Fair$
MA67	Iterator	0.92	0.83	0.99	30.67	1.00	48.00	$(LCOM5 \leq 0.17) \wedge (NOA \leq 18.38) \Rightarrow Fair$
MA68	Observer	0.90	0.66	1.03	112.43	1.00	191.00	$(LCOM5 \leq 0.38) \wedge (NOA \leq 74.32) \Rightarrow Fair$
MA69	Observer	0.83	0.83	0.83	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA70	Observer	0.83	0.83	0.83	11.00	11.00	11.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA71	Singleton	0.00	0.00	0.00	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA72	Singleton	0.00	0.00	0.00	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
Lexi								
MA8	Builder	0.95	0.93	0.97	7.75	1.00	12.00	$(LCOM5 \leq 0.03) \wedge (NOA \leq 4.30 \Rightarrow Fair$
MA9	Observer	0.95	0.94	0.97	9.50	1.00	18.00	$(LCOM5 \leq 0.02) \wedge (NOA \leq 6.65) \Rightarrow Fair$
MA10	Observer	0.95	0.94	0.97	61.67	35.00	94.00	$(LCOM5 \leq 0.02) \wedge (NOA \leq 23.08) \Rightarrow Fair$
MA11	Singleton	1.01	1.01	1.01	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA12	Singleton	0.99	0.99	0.99	2.00	2.00	2.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$

Table 6: Data and rules when applying the quality model to a subset of JHotDraw, JUnit, and Lexi