This paper has been accepted at WCRE 2004.

Fingerprinting Design Patterns*

Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi
Département d'informatique et de recherche opérationnelle
Université de Montréal – CP 6128 succ. Centre Ville
Montréal, Québec, H3C 3J7 – Canada

E-mail: {guehene,sahraouh,zaidifar}@iro.umontreal.ca

Abstract

Design patterns describe good *solutions to common and recurring problems in program design. The solutions are* design motifs *which software engineers imitate and introduce in the architecture of their program. It is important to identify the design motifs used in a program architecture to understand solved design problems and to make informed changes to the program. The identification of* micro-architectures *similar to design motifs is difficult because of the large search space, i.e., the many possible combinations of classes. We propose an experimental study of classes playing roles in design motifs using metrics and a machine learning algorithm to* fingerprint *design motifs roles. Fingerprints are sets of metric values characterising classes playing a given role. We devise fingerprints experimentally using a repository of micro-architectures similar to design motifs. We show that fingerprints help in reducing the search space of micro-architectures similar to design motifs efficiently using the Composite design motif and the* JHotDraw *framework.*

1 Motivations

Design patterns [9] collect experts' knowledge in object-oriented software design. They name and describe (1) problems recurring when designing software, (2) *good* solutions to these problems, and (3) the implications of these solutions on the design. The solutions offered by design patterns are described by means of *design motifs*: Prototypical micro-architectures from which software developers draw inspiration to design their programs. Design motifs declare actors and the relationships among actors, as sketched on Figure 1. A program architecture may contain several *micro-architectures* similar to design motifs when software developers used design patterns. Figure 2 sketches a meta-model to describe micro-architectures similar to design motifs in object-oriented programs.

*This work has been partly funded by the Natural Sciences and Engineering Research Council of Canada.

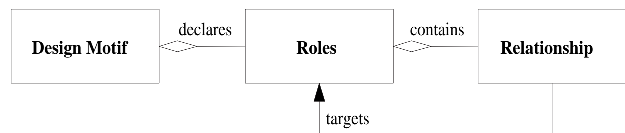

Figure 1. Design motifs meta-model

Figure 2. Micro-architectures meta-model

It is desirable to know the design motifs used by software developers when reverse-engineering a program architecture. Indeed, knowledge about applied design motifs leads to a better understanding of the design problems solved by software developers when designing the program architecture. Understanding of the design problems is a necessary step towards informed changes of the program and of its architecture.

The acquisition of knowledge on design motifs used by software developers in a program architecture requires finding all micro-architectures similar to design motifs in the architecture, *i.e.*, finding all classes (or interfaces) which structures and organisations are similar to design motifs. Thus, the identification of design motifs is a problem of finding patterns in complex graphs and is difficult—high algorithmic complexity, time- and space-consuming, results with low precision and low recall—because of the many possible combinations of classes and of the size of programs [2].

A possibility to reduce the difficulty of design motifs identification is to conceive and to apply heuristics that reduce the search space, *i.e.*, that reduce the number of potential combinations of classes. We are exploring this possibility by conducting an experimental study of micro-architectures similar to design motifs. We seek to find quantitative signatures common to classes playing given roles in design motifs to ease their identification in programs architectures and thus to reduce the complexity of design motifs identification algorithms. Quantitative signatures for design motifs are similar to fingerprints for individuals, they allow efficient and automated identification.

As interesting side effects to our experimental study, we can use quantitative signatures to assess the quality characteristics of suggested design motifs quantitatively. Therefore, we can attempt to predict the quality of programs architectures when using design patterns. Also, we can assess whether or not classes which structure and organisation respect a design motif—a *good* solution to a recurring design problem—break sound principles of software engineering, such as low coupling and high cohesion.

In this paper, we describe a first experimental setting and preliminary results from our experimental study of micro-architectures similar to design motifs. We introduce *roles fingerprints* inferred using a propositional rule learner algorithm from a set of metric values computed on classes playing these roles in design motifs. Roles fingerprints are quantitative signatures of design motifs roles that can be used to reduce efficiently the number of classes playing potentially a role in a design motif. Also, we show experimentally using roles fingerprints that the use of design motifs leads to good designs generally, *i.e.*, designs respecting software engineering principles.

Section 2 summarises related work and concludes on the limitations of previous approaches. Section 3 introduces the theoretical background of our experimental study of micro-architectures similar to design motifs. Section 4 introduces a first experimental setting to infer roles fingerprints for design motifs roles. Section 5 details the results of our study and discuss the fingerprints from the qualitative and quantitative point of views. Section 6 presents an example of the use of fingerprints to ease the identification of the Composite design motif. Finally, Section 7 concludes on this first experimental study and presents future work.

2 Related Work

The work presented in this paper overlaps two research fields, namely design motifs identification and the study of the impact of design motifs on software quality. This section presents related work in both fields briefly.

On the one hand, several work tackled the problem of design motifs identification. Most of the approaches use structural matching between groups of classes—micro-architectures—and design motifs. Different techniques are used: Rule inference [18, 27], queries [7, 16], fuzzy reasoning nets [14], constraint programming [11, 22]. For example, Wuyts [27] developed the SOUL environment in which design motifs are described as Prolog predicates and programs entities as facts (classes, methods, fields...). Then, a Prolog inference algorithm unifies predicates and facts to identify classes playing roles in design motifs. The main problem of structural approaches is the inherent combinatorial complexity of identifying subsets of classes matching design motif descriptions, which corresponds to a problem of subgraph isomorphism [8]. Antoniol *et al.* proposed an alternative approach to reduce the search space of micro-architectures [2]. They designed a multi-stage filtering process to identify micro-architectures identical to design motifs using metrics. 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. The expected values are derived from design pattern descriptions. The main limitation of their work is the assumption that the implementation (micro-architectures) accurately reflects the theory (design motifs), which is often not the case. Moreover, the theoretical quantification of roles, when possible, does not reduce the search space significantly.

On the other hand, some work studied the impact on quality of design patterns. Wendorff [25] reported on a large commercial software project in which design patterns had been applied too eagerly, resulting in maintenance problems. Hahsler [12] proposed a study of over 1,000 open source projects and provided evidence that the use of design patterns improve both communication among developers and source code documentation. Finally, in a work similar to ours, Masuda *et al.* [21] performed a quantitative evaluation and an analysis of the application of design motifs. They built two sets of programs: One set containing two programs implemented without design motifs; One set containing the two same programs, rewritten to use design motifs. They computed Chidamber and Kemerer's metrics [6] on the two sets of programs and compared the results. They concluded that the use of design motifs deteriorates certain metric values and suggested that Chidamber and Kemerer's metrics might not be appropriate to assess the

quality of programs implemented with design motifs. However, their experimental study bears on two sets of two programs only, which are rather small sets of data to be significant.

The work that we propose in this paper builds on the ideas from Antoniol *et al.* and Masuda *et al.* and circumvents the problem of defining the expected metric values, the values of reference, by mining these values from a repository of micro-architectures directly. Also, our work aims at reducing the search space by considering a large set of structural metrics. Finally, we compare the metric values determined experimentally to well-accepted software engineering principles, such as *low coupling* or *high cohesion*.

3 Fingerprinting Design Motifs Roles

In a not-so-distant past, individuals could be identified by external attributes only, such as height, weight, colour of hair, of eyes, of skin; for example using Alphonse Bertillion's system of bodily measures [17]. Thus, it was difficult to identify with certainty an individual uniquely, almost impossible without eyewitnesses. This situation changed when Sir Edward Henry devised and introduced in 1896 his classification system to identify criminals in Bengal using their fingerprints [19]. The use of fingerprints, imprints made by the pattern of ridges on the pad of a human finger, allows to distinguish among individuals: To our best knowledge, no two individuals have ever been found to have identical fingerprints.

Likewise, we seek to identify classes playing roles in design motifs using their external attributes. The most consensual attributes for classes in object-oriented programming languages are:

- Size, *e.g.*, number of methods, of fields.
- Filiation, *e.g.*, number of parents, number of children of a class in 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.

However, unlike individuals, two or more classes may have identical values for a given set of external 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, external attributes cannot be used to distinguish uniquely among classes paying roles in design motifs.

Yet, external attributes can be used to reduce the search space of micro-architectures similar to design motifs. We can use external 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 the impossibility to identify classes uniquely with their external attributes, or attempted to find quantifiable *commonalities* among classes playing a given role in a design motif experimentally.

Therefore, we study the use of external attributes of classes to quantify design motifs roles: We devise a kind of *fingerprints* for design motifs roles using external attributes of classes. We group these fingerprints 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 Building Roles Fingerprints

4.1 Overview

Figure 3 presents an overview of the process of fingerprinting 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 *parse* the programs in which we found micro-architectures similar to design motifs to obtain models of these programs. We compute *metrics* on these models of programs to associate a set of values for the external 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.

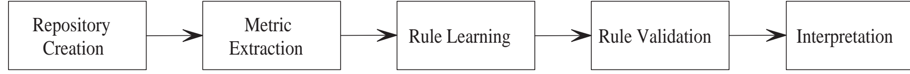

Figure 3. Process of fingerprinting design motifs roles

4.2 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 DPL[1]. We create this repository using different sources:

- Studies in the literature, such as the original study from James Bieman *et al.* [3], 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[2] [1, 10], which revolves around a constraint solver to identify micro-architectures similar to design motifs in programs.
- One assignment in a graduate course, during which students performed analyses of two Java programs, QUICKUML and LEXI.

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

- For each program, design motifs for which we found similar micro-architectures.
- For each design 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 DPL contains data from 6 programs, for a total of 7,068 classes and 93 micro-architectures representing 15 different design motifs. Table 1 summarises the data in the DPL. 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.

[1] DPL is a *Design Pattern Library*.

[2] PTIDEJ stands for *Pattern Trace Identification, Detection, and Enhancement in Java*.

```
<program type="LANGUAGE">
    <name>NAME</name>
    <designMotif name="NAME">
        <microArchitectures>
            <microArchitecture n="NUMBER">
                <roles>
                    <ROLES1>
                        <ROLE1>
                            <class>
                                NAME
                            </class>
                        </ROLE1>
                        ...
                    </ROLES1>
                    ...
                </roles>
            </microArchitecture>
            ...
        </microArchitectures>
    </designMotif>
    ...
</program>
...
```

Figure 4. Structure of the repository

We record this data in a XML file, which allows us to traverse the data to compute metrics and various statistics automatically. Figure 4 shows the general structure of the XML file: A program is written in a LANGUAGE and has a (unique) NAME; Each design motif has a (unique) NAME also; A micro-architecture possesses a unique NUMBER and associates each possible role in the design motif, ROLE1, ROLE2..., ROLEN, with the classes NAMEs (if any) playing this role.

	JHotDraw v5.1	JRefactory v2.6.24	JUnit v3.7	Lexi v0.0.1α	NetBeans v1.0.x	QuickUML 2001	Total	Number of roles [9]	Number of classes playing a role per design motif
Number of classes	173	575	157	127	5812	224	7,068		
Design motifs[5]	Number of micro-architectures similar to design motifs per program								
Abstract Factory					12	1	13	5	217
Adapter	1	17			8		26	4	230
Builder		2		1		1	4	4	24
Command	1				1	1	3	5	67
Composite	1		1			2	4	4	107
Decorator	1		1				2	4	64
Factory Method	3	1					4	4	67
Iterator			1		5		6	5	30
Observer	2		3	2		1	8	4	93
Prototype	2						2	3	32
Singleton	2	2	2	2		1	9	1	9
State	2	2					4	3	32
Strategy	4						4	3	36
Template Method	2						2	2	36
Visitor		2					2	4	138
						Total	93	55	1182

Table 1. Overview of the data set[6]: Programs, design motifs, micro-architectures, and roles

4.3 Metric Extraction

We parse the programs surveyed in the DPL 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[3] meta-model and its parsers; Second, we compute metrics using POM[4], an extensible framework for metric calculation based on PADL; Third, we store the results of the metric calculation, names and values, in the DPL, by adding specific attributes and nodes to the XML tree representation.

We use metrics from the literature to associate values with external attributes of classes playing a role in a design motif. Table 2 presents the metrics computed on classes related to the external attributes that we consider: Size, filiation, cohesion, and coupling. For size, we use the metrics by Lorenz and Kidd on new, inherited, and overridden methods and on the total number of methods [20], and the count of methods weighted with their numbers of method invocations by Chidamber and Kemerer [6]. 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 [6] and the number of hierarchical levels below a class, class-to-leaf depth, by Tegarden *et al.* [24]. For cohesion, we use the metric 'C' measuring the connectivity of a class with the rest of a program by Hitz and Montazeri [13] and the fifth metric on lack of cohesion in methods by Briand *et al.* [5]. Finally, for coupling, we use two metrics on class-method import and export coupling by Briand *et al.* [4] and the metric on coupling between objects by Chidamber and Kemerer [6].

4.4 Rule Learning and Validation

The DPL 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 [26].

We do not provide JRip with all the data in the DPL, 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 its 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 [9], to increase their likeliness with the

[3] PADL is the acronym of *Pattern and Abstract-level Description Language*.

[4] Metric extraction is based on *Primitives Operators Metrics*.

[5] Design motifs for which we did not identify similar micro-architectures are: Bridge, Chain of Responsibility, Façade, Flyweight, Interpreter, Mediator, Memento, Proxy.

[6] For clarity, an empty cell has value zero.

	Acronyms	Descriptions	References
Size	NM	Number of methods	[20]
	NMA	Number of new methods	[20]
	NMI	Number of inherited methods	[20]
	NMO	Number of overridden methods	[20]
	WMC	Weighted methods count	[6]
Filiation	CLD	Class-to-leaf depth	[24]
	DIT	Depth in inheritance tree	[6]
	NOC	Number of children	[6]
Cohesion	C	Connectivity 'C'	[13]
	LCOM5	Lack of cohesion in methods 5	[5]
Coupling	ACMIC	Ancestors class-method import	[4]
	CBO	Coupling between object	[6]
	DCMEC	Descendants class-method export	[4]

Table 2. External attributes for classes and corresponding metrics

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 [23].

4.5 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:
    - Fingerprint 1, confidence 1,
    - Fingerprint 2, confidence 2,
    - ...
    - Fingerprint N, confidence N.
```

where

$$\texttt{Fingerprint 1} : \{metric_1 \in V_{11}, \ldots, metric_m \in V_{m1}\}$$
$$\vdots$$
$$\texttt{Fingerprint N} : \{metric_1 \in V_{1n}, \ldots, metric_m \in V_{mn}\}$$

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 fingerprint 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 fingerprints reduce the search space for micro-architectures similar to design motifs efficiently.

5 Interpretation of Roles Fingerprints

We decompose the data in the DPL in 56 subsets σ and infer as many rules with the rule learner, which decompose in 78 fingerprints. 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 fingerprints 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:

- Design motifs roles with few corresponding micro-architectures and with a unique (or a few) occurrence in a micro-architecture. 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 3, from the 36 remaining rules. All these roles have an error rate less than 10% (less than 5% for 16). Most of the rules removed because of their low recall ratio are known to be non-key roles in design motifs and thus do not have a particular fingerprint 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.)

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 3. Roles with inferred rules with recall ratio greater than 75%

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

Figure 5. Rules inferred for the role of Target **in the** Adapter **design motif**

5.1 Qualitative Study

We notice that in many cases we obtain a unique fingerprint 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 5 (the degree of confidence is less than 1 because this fingerprint misclassifies one class, its error rate is 4%, as shown in Table 3). Such a low complexity is actually expected because of the architecture and of the behaviour suggested by the Adapter design motif. Likewise, many other fingerprints 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 fingerprint for a role. An example is the role of Concrete Visitor in the Visitor design motif. On the one hand, the most frequent fingerprint 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 fingerprint 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 fingerprint is particular to the program and thus unveils design choices specific to the program or to a coding style.

The fingerprints we have discovered confirm common claims from and beliefs on design motifs, hence we are surprised that no existing design motifs identification tools use such claims and beliefs. Moreover, when heuristics are used, as in the work by Antoniol *et al.* [2], the knowledge is abstracted from theory only (design pattern descriptions) and is not validated on actual micro-architectures.

5.2 Quantitative Study

From the point of view of the design quality, no fingerprint of the 20 roles which rules have a recall ratio greater than 75% have a high coupling or a low cohesion. Classes playing these roles comply with sound principles of software engineering: Low coupling and high cohesion. Generally, Figure 6 shows that the majority of classes playing roles in design motifs—including removed roles—exhibit low coupling (low CBO), high cohesion (low LCOM5), and low complexity (low WMC). Thus, we confirm experimentally that the use of design motifs lead to programs architectures with *good* quality characteristics. Figure 6 shows also that some metrics are irrelevant to fingerprint design motifs roles: No fingerprint uses the NOC,

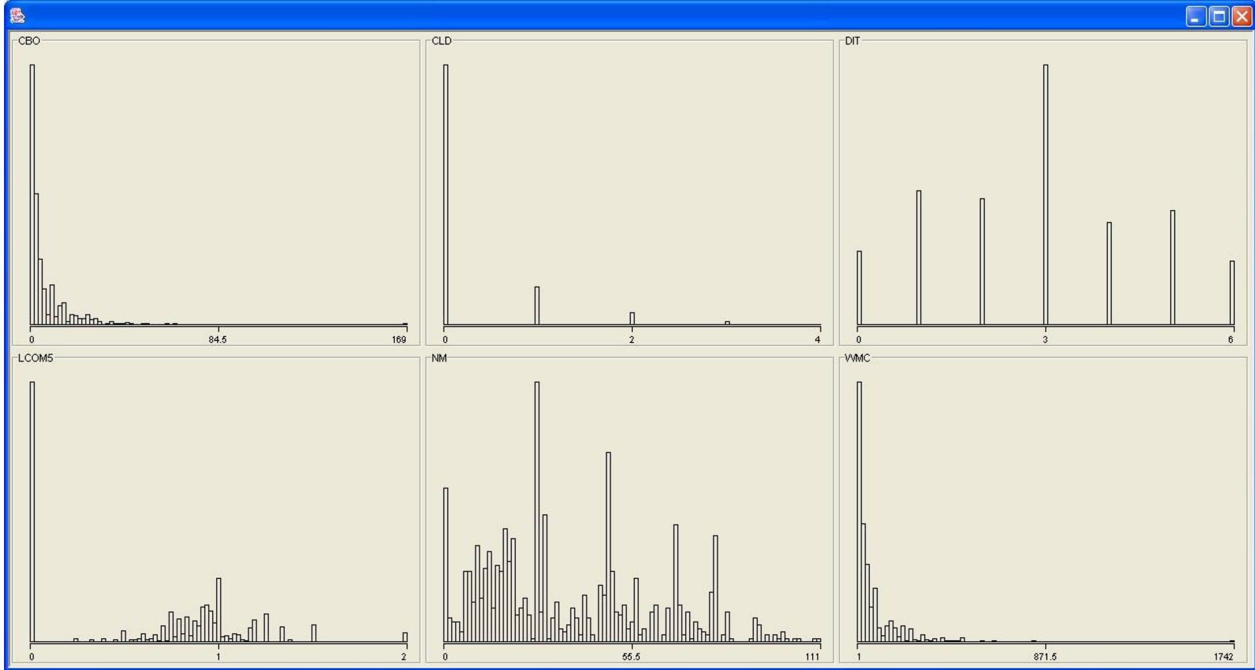

Figure 6. Summary of the metric values for all the rules

C, ACMIC, and DCMEC metrics. (The NMA, NMI, and NMO metrics do not appear, because, to our best knowledge, no software engineering principles involve these metrics.)

6 Application of Roles Fingerprints

As stated in Section 3, fingerprints inferred by the rule learner from metric values computed on classes playing roles in design motifs must not be used to identify design motifs. Indeed, two or more classes may have theoretically the same fingerprints as do two or more roles. The use of fingerprints to identify roles would lead to many false positives. For example, the rule shown in Figure 5 for the Target role corresponds to many classes in a program.

However, fingerprints help in reducing the search space for micro-architectures similar to design motifs efficiently. When searching for classes which structures and organisations are similar to a design motif, the use of fingerprints allows to remove from the search space all the classes which fingerprints do not match expected fingerprints for the design motif roles. For example, when searching for micro-architectures similar to the Adapter design motif and according to the rule in Figure 5, we can remove from the set of candidate classes for the role of Target any class with a high complexity, *i.e.*, with a medium or high value for its WMC.

We integrate fingerprints with our constraint-based tool suite for design motifs identification and program understanding, PTIDEJ. In previous work [1, 10], we described 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 7, 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.

With fingerprints, we can reduce the search space in two ways:

- We can assign a different domain to each variable containing only those classes which fingerprints match the expected fingerprints for the role.

- We can add unary constraints on each variable to match the fingerprints of the classes in its domain with the fingerprint of the corresponding role.

These two ways achieve a same result: They remove from the domain of a variable all the classes which fingerprints do not match the expected role fingerprint, thus reducing the search space by reducing the domains of the variables.

Among the roles of Client, Component, Composite, and Leaf, the rule learner infers a rule with a recall ratio greater than 75% for the role of Leaf only, as shown in Table 3. The rule and the associated fingerprints for the role of Leaf are

```
Rule for "Leaf" role:
    - NMI >= 26 and DIT >= 5, 23/67
    - NMI >= 25 and NMO <= 2, 45/67
    - DIT >= 3 and NM <= 12, 9/67
```

We compute the metrics in Table 2 on the classes of JHotDraw and apply the fingerprints successively to assess the search space reduction. Table 4 shows the percentages of search space reductions with each different fingerprints. The reduction of the search space lies between 69.00% and 89.15% for classes playing possibly the role of Leaf in a micro-architecture similar to the Composite design motif. It takes less than 2 seconds to compute all the metric values on an AMD Athlon 64bits processor at 2GHz (we need to compute these metrics only once for any given program) while computing all possible micro-architectures similar to the design motif in the JHotDraw framework takes more than 4 hours using the reference implementation of the PaLM explanation-based constraint solver [11, 15]. Computing all the micro-architectures with a reduced search space for the role of Leaf takes between 2 and 3 hours only. Therefore, fingerprints are invaluable to ease design motifs identification by reducing the search space efficiently.

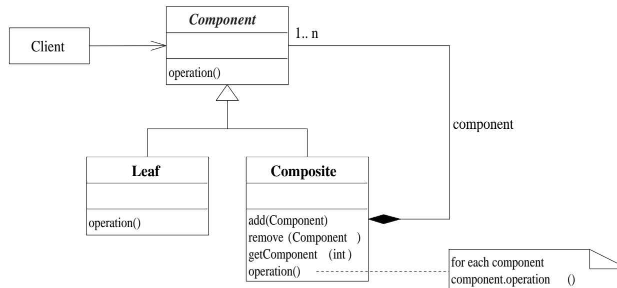

Figure 7. The Composite design motif [9]

Fingerprints	Numbers of classes matching a fingerprint	Reductions of the search space (%)
NMI >= 26 and DIT >= 5	20	69.00
NMI >= 25 and NMO <= 2	7	89.15
DIT >= 3 and NM <= 12	10	84.50

Table 4. Space search reduction with Leaf role fingerprints in JHotDraw

7 Conclusion and Future Work

We presented the first results of an experimental study of micro-architectures similar to design motifs. We built a repository of such micro-architectures using different sources. We computed several metrics on classes playing roles in design motifs. We applied a rule learner (machine learning algorithm) and found that classes playing certain roles in design motifs share same fingerprints (quantitative signatures). We used these fingerprints to improve design motif identification algorithms. Indeed, the identification process now decomposes in two steps: (1) To identify candidate classes for key-roles in design motifs by eliminating classes that do not match expected fingerprints and (2) to identify candidate classes for the remaining roles starting from key-role candidates and using structural matching. This process reduces the search space efficiently, in particular in the case of large programs by removing many classes that obviously do not play a role in a design motif.

The secondary objective of our experimental study was to assess whether programs implemented using design motifs conform with software engineering principles generally. We did not find transgression of well-accepted principles, such as low coupling, high cohesion, and low complexity. However, our analysis considered general principles only. We plan to conduct a thorough analysis of design motifs and of the impact of their use on quality by conducting an experimental evaluation of the claims of each design pattern regarding software quality characteristics (extensibility, understandability...) independently.

Also, we are preparing a replication of our study with a larger repository of micro-architectures similar to design motifs. In this replication study, we will improve the experimental setting. One important

change concerns the composition of the learning samples for each design motif role: The number of counter-examples and there nature shall be chosen as close as possible to real situations of identifications to produce fingerprints with greater accuracy. Also, other object-oriented metrics shall be investigated.

Acknowledgment

We thank James Bieman, Greg Straw, Huxia Wang, P. Willard, and Roger T. Alexander [3] for kindly sharing their data, our graduate students for helping in the creation of the repository, and Hervé Albin-Amiot for his comments on earlier versions of this paper.

References

[1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *proceedings of the 16th conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

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

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

[4] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *proceedings of the 19th International Conference on Software Engineering*, pages 412–421. ACM Press, May 1997.

[5] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for cohesion measurement. In *proceedings of the 4th international Software Metrics Symposium*, pages 43–53. IEEE Computer Society Press, November 1997.

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

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

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

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

[10] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *proceedings of the 39th conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

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

[12] M. Hahsler. A quantitative study of the application of design patterns in Java. Technical Report 1/2003, University of Wien, January 2003.

[13] M. Hitz and B. Montazeri. 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, October 1995.

[14] J. H. Jahnke, W. Schäfer, and A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *proceedings of the 6th European Software Engineering Conference*, pages 193–210. ACM Press, September 1997.

[15] N. Jussien and V. Barichard. The PaLM system: Explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint Programming Systems*, pages 118–133. School of Computing, National University of Singapore, Singapore, September 2000. TRA9/00.

[16] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *proceedings of the 21st International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.

[17] M. Kelly. Biometrics – The basics, August 2004.

[18] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *proceedings of the 3rd Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.

[19] London Metropolitan Police. Sir Edward Henry – Fingerprint pioneer and founder of the Fingerprint Bureau at Scotland Yard, July 2001. See www.met.police.uk/so/100years/henry.htm.

[20] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1st edition, July 1994.

[21] G. Masuda, N. Sakamoto, and K. Ushijima. Evaluation and analysis of applying design patterns. In *proceedings of the 2nd International Workshop on the Principles of Software Evolution*. ACM Press, July 1999.

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

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

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

[25] P. Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In *proceedings of 5th Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

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

[27] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *proceedings of the 26th conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.