Detecting Implicit Collaboration Patterns *

Gabriela Arévalo, Frank Buchli, Oscar Nierstrasz
Software Composition Group - Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland
{arevalo, buchli, oscar}@iam.unibe.ch

Abstract

A key problem during software development and maintenance is to detect and recognize recurring collaborations among software artifacts that are implicit in the code. These collaboration patterns *are typically signs of applied idioms, conventions and design patterns during the development of the system, and may entail implicit contracts that should be respected during maintenance, but are not documented explicitly. In this paper we apply* Formal Concept Analysis *to detect implicit collaboration patterns. Our approach generalizes Antoniol and Tonella one for detecting classical design patterns. We introduce a variation to their algorithm to reduce the computation time of the concepts, a language-independent approach for object-oriented languages, and a post-processing phase in which pattern candidates are filtered out. We identify collaboration patterns in the analyzed applications, match them against libraries of known design patterns, and establish relationships between detected patterns and their nearest* neighbours.

1. Introduction

One of the key difficulties faced by developers who must maintain and extend complex software systems, is to know what are the *implicit contracts* in the system. Such contracts are typically manifested as recurring patterns of software artifacts, which may represent design patterns, architectural constraints, or simply idioms and conventions adopted for the project. We introduce the term *collaboration pattern* to cover all these cases.

In most applications, the implicit contracts may be recovered by recognizing occurrences of collaboration patterns in the source code [4] [14]. However this task is anything but trivial in medium-sized to large applications. In most cases, the documentation of the systems is out-of-date, and the information we seek is not explicit in the code [7] [17].

We explore an approach for detecting collaboration patterns that refines and extends that which was proposed by Tonella and Antoniol [18] for detecting classical design patterns. We take the source-code of an object-oriented application as our main information source and extract structural relationships between classes. We then apply Formal Concept Analysis (FCA) to identify recurring "concepts" (*i.e.*, patterns) in the software. Our approach consists in:

- improvements to the pattern detection algorithm used by Tonella and Antoniol to avoid redundancy in the representation of structural relationships and improve the time performance of calculating concepts,
- generalization of the technique to a language-independent approach,
- the introduction of a filtering phase to narrow the scope of candidate patterns.

Based on the results from our experiments, the main contributions of this paper are:

- the detection of both classical and non-classical patterns in the different applications using simple structural relationships between classes. We are not limited to known design patterns but can detect *any* recurring collaboration between classes in the analyzed applications.
- the possibility of establishing relationships, called *pattern neighbourhoods*, over detected patterns. With the *neighbours* of the patterns, we can detect either missing relationships between classes needed to complete a pattern, or excess relationships between classes that extend a *pattern*. We can also analyze the connections of the identi-

* In Proceedings of WCRE 2004 (11^{th} Working Conference on Reverse Engineering) pp.122-131, IEEE Computer Society Press, 2004

fied patterns with the classes implemented in the analyzed application.
- the incremental construction of a pattern library to match candidates against known design patterns and detected patterns after each case study.

For the sake of conciseness, we use the term *patterns* to refer to *collaboration patterns* in the rest of the paper.

The paper is structured as follows: In section 2 we describe in detail the steps of the pattern detection approach in section. We describe and evaluate our experimental results in sections 3 and 4. We give an overview of related work in section 5, and we conclude with some remarks concerning related and future work.

2. Recognizing Collaboration Patterns with FCA

Formal concept analysis (FCA) [9] is a branch of lattice theory that allows us to identify meaningful groupings of "elements" that have common "properties" (referred to, respectively, as *objects* and *attributes* in the standard FCA literature[1]). The sets of elements, properties and binary relation (represented in an *incidence table*) between them are known as *context*, and the groupings based on the common properties of the elements are named as *concepts*. The set of all the concepts of a given context forms a *complete partial order*. Thus the set of concepts constitutes a *concept lattice* $\mathcal{L}(\mathcal{T})$ and there are several algorithms for computing the concepts and the concept lattice. For more details, the interested reader should consult Ganter and Wille [9].

We will now explain (i) how the source code is mapped in terms of elements and properties, (ii) how the concepts are computed, and (iii) how we apply post-filtering to reduce the set of candidate patterns.

2.1. FCA Mapping: Setup of the Formal Context

In order to use FCA, we need to define the elements and properties of a context $\mathcal{C}$. The **elements** $\mathcal{E}_o$ are tuples of classes from the analyzed application. The length of these tuples is defined as the *order* o. The **properties** $\mathcal{P}_o$ are *relations* inside one class tuple. Whenever such a relation $p_i \in \mathcal{P}_o$ within the tuple $e_j \in \mathcal{E}_o$ is fulfilled we add the relation (p_i, e_j) to the incidence table $\mathcal{I}$. We use a simple example to clarify the terms and the definitions. Figure 1 introduces an example consisting of seven classes. The key information of interest to us is the relationships that hold among classes.

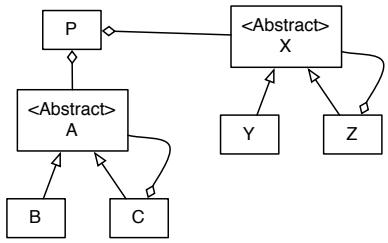

Figure 1. Example class diagram

Elements: Permutation of Classes. We have said that the elements are tuples of classes. We build the elements as all the permutations of the classes with the length of order o:

$$\mathcal{E}_o = \{(x_1, \cdots, x_o) \mid x_i \in C, 1 \leq i \leq o\}$$

We adapt the algorithm proposed by Tonella and Antoniol [18]. This is an inductive context construction algorithm that avoids the combinatorial explosion which results when generating all possible tuples of classes. The underlying hypothesis is that the patterns consist of classes which are *all connected* together by their relations (unrelated classes are not interesting). In the initial step of the algorithm, all pairs of related classes are collected. In the inductive step, the class sequence from the previous iteration is augmented with all the classes having some relation with the classes in the sequence.

In our variation of the algorithm, we avoid generating all permutations of class sequences. For example, if the tuple (C A P) is generated, and we subsequently generate (A P C) or (C P A), we only keep one of these as being representative of all three alternatives.

In case of Figure 1 all possible combinations of the class tuples of order $o = 3$ would lead to 210 elements[2], while the inductively constructed context contains only seven elements, listed in first column of Table 1.

Properties: Class Relations and Characteristics. The properties are all the possible combinations of a relation $C \times C$ inside a tuple $e_t \in \mathcal{E}$ together with the unary relations of each single class C:

1 We use the terms *element* and *property* instead to avoid the unfortunate clash with object-oriented terminology.

2 $\binom{7}{3} 3! = 210$

	$(1,2)_{Sub}$	$(3,1)_{Sub}$	$(3,2)_{Sub}$	$(2,1)_{Acc}$	$(1,2)_{Acc}$	$(3,2)_{Acc}$	$(2,3)_{Acc}$	$(1)_{Abs}$	$(2)_{Abs}$	$(3)_{Abs}$
{C A P}	×				×	×			×	
{C A B}	×		×		×				×	
{Z X Y}	×		×		×				×	
{Z X P}	×				×	×			×	
{A P B}		×		×				×		
{A P X}				×			×	×		×
{Y X P}	×					×			×	

Table 1. Order 3 context for the example in Figure 1

$$\mathcal{P}_o = \{(i,j)_t \mid (x_i, x_j)_t \in R_B, 1 \leq i,j \leq o\} \cup \{(i)_t \mid (x_i)_t \in R_U, 1 \leq i \leq o\}$$

Each property has one or two indices that refer to the position of the class to be analyzed inside the tuple, and a subindex t to indicate the name of the property. For example, the property $(3,2)_{Sub}$ applied to the element {C A B} means that the class B is a subclass of the class A. Using indices instead of names allows disjunct tuples to share common properties. In the example result (Table 2) the tuples {C A B} and {Z X Y} have the common properties $(1,2)_{Sub}$, $(2)_{Abs}$, $(1,2)_{Acc}$ and $(3,2)_{Sub}$.

2.2. ConAn Engine: Calculation of the Concepts

There are several algorithms to calculate the concepts and its lattice [16]. We use the Ganter algorithm [9], which is one of the fastest algorithms known [12].

The example of Figure 1 yields ten concepts for the order $o = 3$. They are listed in Table 2.

Let's consider the specific case of a well-known pattern, such as *Composite Pattern* [8]. We can reduce and generalize the structural information to the relationships *isSubclass*, *isAbstract* and *accesses* and see it as in Figure 2. This simplified Composite Pattern is detected twice in the example of Figure 1: {C A B} and {Z X Y} as concept 2.

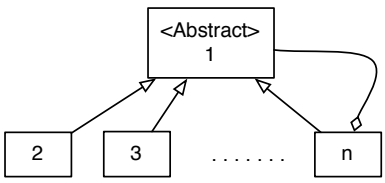

Figure 2. Structural relationships of the Composite Pattern

2.3. Concept Lattice: Post Filtering

Once the concepts are calculated, each concept is a *candidate* for a pattern. But not all concepts are relevant. Therefore a post processing is needed to filter out concepts that are not meaningful. Two particularly useful filters are: *removing disconnected patterns* and *merging equivalent patterns*, which can be applied to a graph representation of concepts for our domain.

Intent relation graph: An *intent relation graph* is a graph whose nodes are the indices of the properties of the binary relation R_B and whose edges are binary relations R_B between the indices.

The intent graphs of concepts 2, 4 and 8 from Table 2 are shown in Figure 3. The edge between node 1 and node 2 in graph $c2$ represents the property $(1,2)_{Sub}$ or $(1,2)_{Acc}$ and the second edge between node 2 and 3 is from the property $(3,2)_{Sub}$. We similarly build the graph of concepts 4 and 8. As soon as at least one relation between two nodes holds, the edge exists.

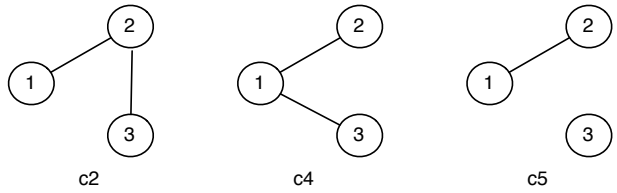

Figure 3. The intent graph of concepts 2, 4 and 8 of Table 2

Removing Disconnected Patterns. A concept is meaningful when the intent (all properties which are true for this concept) is a set of structurally connected nodes.

A *connected pattern* is a pattern whose intent relation graph (definition 2.3) is connected. In the example result (Table 2) the following concepts are disconnected: 6, 7, 8 and the top concept.

Merging Equivalent Patterns. Suppose we have a system with the classes as shown in Figure 4. It might

top	(*all elements* G, $\emptyset$)
8	({ {C A P}, {Z X P}, {C A B}, {Z X Y}, {Y X P} }, { $(1,2)_{Sub}$, $(2)_{Abs}$ })
7	({ {A P X}, {A P B} }, {$(2,1)_{Acc}$, $(1)_{Abs}$ })
6	({ {C A P}, {Z X P}, {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abs}$, $(1,2)_{Acc}$ })
5	({ {C A P}, {Z X P}, {Y X P} }, { $(1,2)_{Sub}$, $(2)_{Abs}$, $(3,2)_{Acc}$ })
4	({ {A P B} }, { $(3,1)_{Sub}$, $(1)_{Abs}$, $(2,1)_{Acc}$ })
3	({ {A P X} }, { $(2,1)_{Acc}$, $(1)_{Abs}$, $(3)_{Abs}$, $(2,3)_{Acc}$})
2	({ {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abs}$, $(1,2)_{Acc}$, $(3,2)_{Sub}$ })
1	({ {C A P}, {Z X P } }, { $(1,2)_{Sub}$, $(2)_{Abs}$, $(1,2)_{Acc}$, $(3,2)_{Acc}$ })
bottom	($\emptyset$, *all properties* M)

Table 2. Concepts of the example in Figure 1

then happen that during the concepts and lattice calculation, we find the two concepts shown in Table 3.

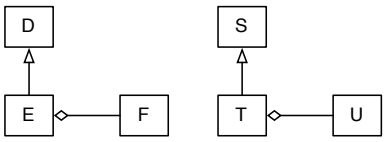

Figure 4. Twice the Adapter Pattern

c_1	({ {D E F} }, { $(2,1)_{Sub}$, $(3,2)_{Acc}$})
c_2	({ {T U S} }, {$(1,3)_{Sub}$, $(2,1)_{Acc}$ })

Table 3. Concepts of the example in Figure 4

Even though {D E F} and {T U S} are exactly the same pattern, the algorithm treats them separately. This happens because when generating the class sequences, we just keep one representative of each possible combination of classes. This means we just look at {D E F} which represents the class sequence {(D F E), (E D F), (E F D), (F D E), (F E D)}.

Two concepts, representing collaboration patterns, are *equivalent* if a permutation of the indices of the intent properties exists such that each property from the first concept can be transformed into a property of the second concept by that permutation, and vice versa [18].

In our example we find a permutation $\alpha = \{3 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3\}$, which transforms the tuple: {T U S} $\overset{\alpha}{\longmapsto}$ {S T U}. Concept c_2 (from Figure 4) can now be removed when the translated extent of this concept is added on concept c_1.

In the main example (Table 2) concepts 4 and 5 are equivalent: The permutation $\alpha = \{3 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2\}$ translates the properties of concept 5 into those of concept 4.

Applying these two filters (*removing disconnected patterns* and *merging equivalent patterns*) on the main example leads to the four patterns presented in Table 4. The first three patterns are directly taken from the first three concepts. Pattern p_4 is merged from concepts 4 and 5. The elements of concept 5 are translated into {P C A}, {P Z X} and {P Y X}, and are appended to {A P B}.

2.4. Pattern Neighborhood

One of the advantages of the FCA approach is that the generated concepts are related within a *complete partial order*. Thus given a concept c, we can identify the *superconcept* (also known as cover concept) and *subconcept*(also known as subordinate concept) of c in a lattice.

With these two ideas, we define the idea of *neighborhood*. We define two kinds of neighbours. Considering that each concept c is a potential pattern, we define:

Almost pattern: An *almost pattern* X of a pattern Y is a pattern X which is contained in the superconcept of the pattern Y in the lattice.

Overloaded pattern: An *overloaded pattern* X of a pattern Y is a pattern X which is contained in the subconcept of a pattern Y in the lattice.

As a generic example, Figure 5 shows the structure of *almost* and *overloaded* patterns of a concept representing the structure of a *Composite Pattern*.

In the lattice of the example (shown in Figure 6), we see that concept 5 is an almost pattern of concept 1. For example, tuple {Y X P} is missing the property $(1,2)_{Acc}$, so belongs to concept 5 instead of concept 1.

But the problem with our concrete approach is that the side-effect of having *equivalent patterns* has to be taken into account here as well. As we have seen concept 4 is equivalent to concept 5. This new concept

p_4	({ {A P B}, {P C A}, {P Z X}, {P Y X}}, { $(3,1)_{Sub}$, $(1)_{Abstr}$, $(2,1)_{Acc}$ })
p_3	({ {A P X} }, { $(2,1)_{Acc}$, $(1)_{Abstr}$, $(3)_{Abstr}$, $(2,3)_{Acc}$})
p_2	({ {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Sub}$ })
p_1	({ {C A P}, {Z X P } }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Acc}$ })

Table 4. Resulting Patterns after the merging of equivalent patterns from the concepts of Table 2

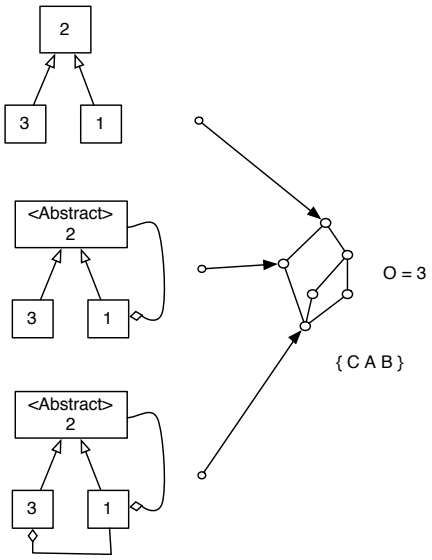

Figure 5. Almost and overloaded patterns of a Composite Pattern

(from the union of concepts 4 and 5) has two main consequences:

- the union of concepts 4 and 5 is now an almost pattern of concept 1
- new connections with other concepts can appear. In this specific case, if we translate the properties of concept 2 with the permutation $\alpha = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$, we see that this transforms the *union* of the concepts 4 and 5 in an almost pattern of the *new* concept 2 considering the intents of the concepts. But the union of the concepts 4 and 5 have to add the *transformed* elements of the extent of concept 2. Thus the elements of p_2 have to be added in p_4: {C A B} $\overset{\alpha}{\longmapsto}$ {A C B} and {Z X Y} $\overset{\alpha}{\longmapsto}$ {X Z Y}.

After the post-filtering process, where we modify the extent of some concepts, finding *equivalent* patterns and removing *disconnected* patterns, we no longer have a valid lattice, but simply a *partial order* [6]. Now the patterns have reached their final state and are listed in Table 5.

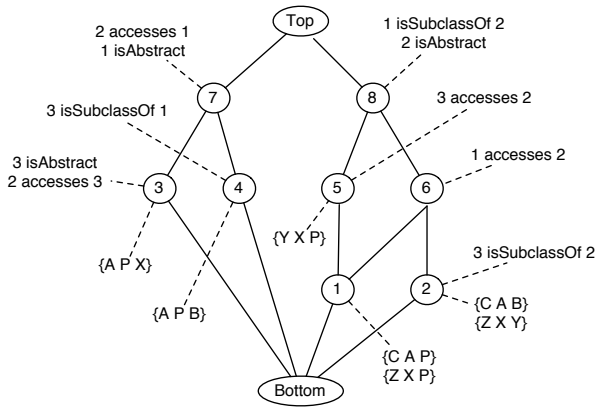

Figure 6. Lattice of Incidence Table 1

Almost and overloaded patterns remain in the same order o. There are as well related patterns when we calculate the lattice for the order $o-1$ and the order $o+1$. Patterns in a higher order have subpatterns from the lower order. They can be detected by subgraph matching techniques.

Cover pattern: C_2 is a *cover pattern* of the pattern c_1, if the intent relation graph (Definition 2.3) of the pattern c_1 is a subgraph of the intent relation graph of c_2. Cover patterns are the connection links to the patterns in the order $o+1$.

Subpattern: C_2 is a *subpattern* of the pattern c_1, if the intent relation graph (Definition 2.3) of the pattern c_2 is a subgraph of the intent relation graph of c_1. Subpatterns are the connection links to the patterns in the order $o-1$. Linking the different orders of the patterns is made *after* applying the post processing filter.

Figure 7 show the subpatterns of the Composite Pattern p_2 in the lower order $o = 2$, and the cover pattern in order $o = 4$.

The pattern neighborhood is now the union of all the above mentioned sets:

Pattern Neighborhood: A *pattern neighborhood* of a pattern c is the union of the almost, overloaded, cover and subpatterns of the pattern c.

The approach of Antoniol and Tonella [18] deals with patterns as isolated entities. With our approach we relate the patterns within the same lattice and also

p_4	({ {A P B}, {P C A}, {P Z X}, {P Y X}, {A C B}, {X Z Y}}, { $(3,1)_{Sub}$, $(1)_{Abstr}$, $(2,1)_{Acc}$ })
p_3	({ {A P X} }, { $(2,1)_{Acc}$, $(1)_{Abstr}$, $(3)_{Abstr}$, $(2,3)_{Acc}$})
p_2	({ {C A B}, {Z X Y} }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Sub}$ })
p_1	({ {C A P}, {Z X P } }, { $(1,2)_{Sub}$, $(2)_{Abstr}$, $(1,2)_{Acc}$, $(3,2)_{Acc}$ })

Table 5. Final Patterns after applying the post filters to the concepts from Table 2

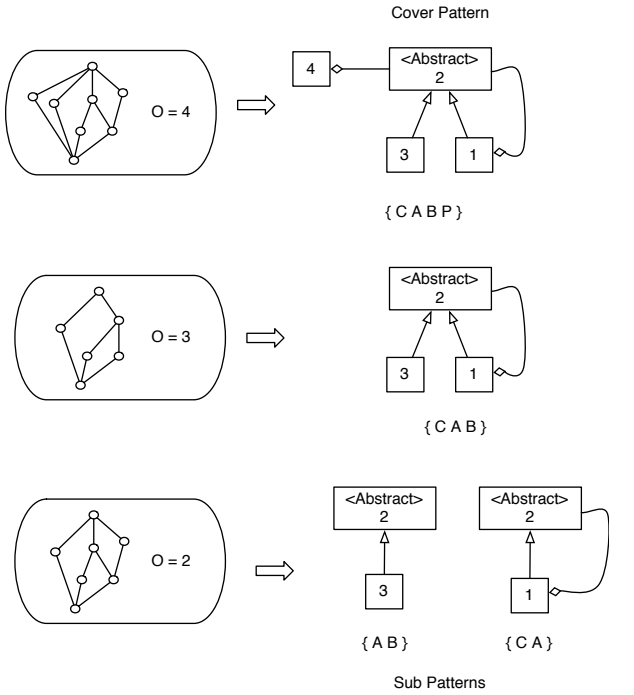

Figure 7. Sub and cover patterns of the Composite Pattern (p_2)

with other patterns calculated in higher and lower orders. Thus, we are able to analyze not only the detected patterns but also the relationships to other patterns in the applications.

3. Validation: Case Studies

We have validated our approach by applying the tool we have implemented called ConAn PaDi to three mid-sized Smalltalk applications: ADvance and CodeCrawler.

- **ADvance**[3] is a system round-trip engineering tool from IC&C. It is a multidimensional OOAD-tool for supporting object-oriented analysis and design, reverse engineering and documentation.
- **CodeCrawler** is a language independent software visualization tool[4]. CodeCrawler supports reverse engineering through the combination of metrics and software visualization [13].

We have applied the approach as outlined in section 2 to each case study. After the concepts have been generated, we classify the patterns in terms of the properties that are used to describe them. We have built two *Classifiers A* and *B*. The *Classifier A* contains three properties: `isSubclass`, `hasAsAttribute`, `isAbstract` and the *Classifier B* contains two properties: `isSubclass`, `hasAsAttribute`.

Thus, for example, we take the *Classifier B* and we get all the patterns that can be described with a set of properties that include `isSubclass` or `hasAsAttribute`. In the specific case of CodeCrawler in order= 3, we get 14 patterns which are distributed in 300 tuples of classes in total (Table 6). This means that *Classifier B* gives us an average of 21 tuples per pattern (300 / 14).

The complete analysis of the quantitative impact of the classifiers is seen in the Table 6. A classifier with less properties gives a clearer image of the situation. Applying *Classifier B* with less properties heavily reduces the number of different patterns whereas the found patterns in total are much less reduced. This leads to patterns which have more tuples as elements. The patterns of *Classifier A* are too "noisy". Comparing again the case of CodeCrawler in order=3, we see that the *Classifier A* has an average of 13 tuples per pattern (431 / 32) meanwhile the *Classifier B* has an average of 21 tuples per pattern (300 / 14).

To better compare the three cases we selected eight reference patterns which are introduced in Tables 7 and 8. *Subclass Star* is a tuple where one class has all the others as subclass. In the *Subclass Chain* the classes form an inheritance chain, whereas in the *Attribute Chain* the classes form an access chain. *Attribute Star* is a pattern with a class which is used as attribute in all the other classes from the tuple. The next four patterns (*Facade*, *Composite*, *Adapter* and *Bridge*) have all names from the collection of Gamma *et. al.* [8]. It

3 http://www.io.com/~icc/

4 http://www.iam.unibe.ch/~scg

o		ADvance Classifier A	ADvance Classifier B	CodeCrawler Classifier A	CodeCrawler Classifier B
2	# different patterns	12	5	7	3
	# patterns in total	215	181	116	85
3	# different patterns	57	32	32	14
	# patterns in total	1103	907	431	300
4	# different patterns	329	218	110	58
	# patterns in total	7521	6093	1423	983

Table 6. Classifier statistics

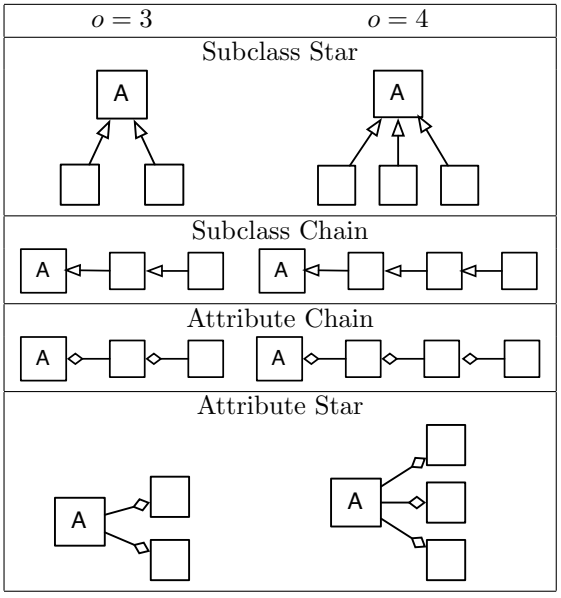

Table 7. Structure of investigated patterns

is important to see that they are simplified to the used structural definitions of inheritance, aggregation and abstractness of a class. If our tool identifies a found pattern with such a reference pattern it just means that it *could* be a candidate for this pattern. The letter A in a box means that the class should be abstract.

Table 9 shows all the found patterns of those eight references for the order $o = 2, 3, 4$. Most of the patterns appear twice: Once in the first line (*e.g.*, *Composite*) where the reference pattern lacks the property *isAbstract*; whereas in the second line the *isAbstract* property is taken into consideration. As Classifier B has no property *isAbstract* it is obvious that no patterns containing an *isAbstract* property can be found.

One interesting observation concerns the two zeros marked with an asterisk: *Subclass Chain* and *Attribute Chain* of order $o = 4$ of the CodeCrawler application. Applying Classifier A no *Subclass Chain* nor *Attribute Chain* of order $o = 4$ is found. Nevertheless Classifier B detects 12 instances of *Subclass Chain* and 15 instances of *Attribute Chain*. Classifier A does not detect those patterns because CodeCrawler has no chain with an abstract class *exactly* at the beginning and the rest of the chain consisting of non abstract classes. Just one representative with an abstract class on top would be enough that the FCA approach would detect the rest of the 12 (resp. 15) patterns. This effect shows that having too many properties can be counterproductive and the basic patterns cannot be detected if there is too much noise.

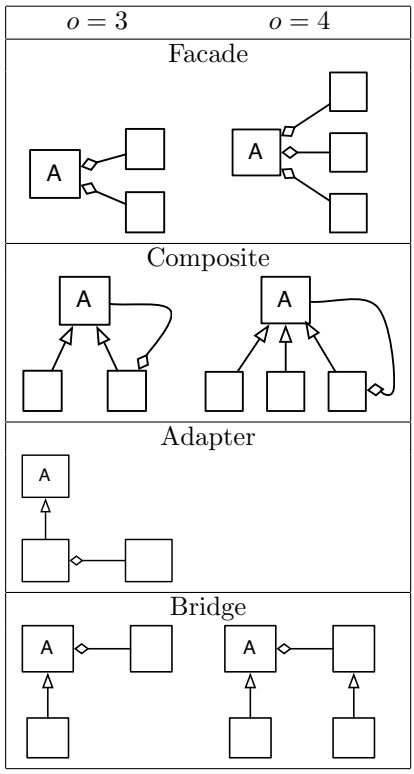

Table 8. Structure of investigated patterns

4. Evaluation of the Results

Based on the results, we are able to evaluate our approach from different viewpoints:

Detect class dependencies: As all the different relations (inheritance, access, invocation) are shown, the dependencies derived from those relations are then available. Looking again at CodeCrawler, we see that *e.g.*, the class `CCTool` cannot have a lot of dependencies because this class is in none of the patterns, whereas `CCNodePlugin` is in 47 patterns up to order $o = 4$ and must therefore have several dependencies.

		ADvance		CodeCrawler	
		Classifier		Classifier	
o	Pattern	A	B	A	B
2	Subclasses	95	95	57	57
	Attributes	80	80	26	26
3	Subclass Star	271	271	140	140
	Abst. Sub. Star	46	-	22	-
	Subclass Chain	44	44	28	28
	Abst. Sub. Chain	10	-	11	-
	Attribute Chain	108	108	25	25
	Facade	214	214	42	42
	Abst. Facade	0	-	15	-
	Attribute Star	44	44	9	9
	Abst. Attrib. Star	3	-	1	-
	Composite	6	6	0	0
	Abst. Composite	2	-	0	-
	Adapter	32	32	4	4
	Abst. Adapter	13	-	1	-
	Bridge	37	37	19	19
	Abst. Bridge	6	-	12	-
4	Subclass Star	1073	1073	313	313
	Abst. Sub. Star	87	-	15	-
	Subclass Chain	12	12	0*	12
	Abst. Sub. Chain	1	-	3	-
	Attribute Chain	137	137	0*	15
	Facade	627	627	56	56
	Abst. Facade	0	-	20	-
	Attribute Star	15	15	0	0
	Abst. Attrib. Star	1	-	0	-
	Composite	3	3	0	0
	Abst. Composite	1	-	0	-
	Bridge	20	20	6	6

Table 9. Investigated Patterns

Identify the possible presence of classical design patterns: Candidates for classical design patterns are found. Some of these turn out to be *false candidates*, *i.e.*, structural patterns which superficially resemble design patterns, but are not in fact instances of those design patterns. The reasons for the misinterpretation are: (1) Not all the properties are absolutely reliable. For example, the extraction of the type of an attribute is based on a heuristic [1], because we work with Smalltalk, which is a dynamically typed language; (2) The collaboration of the detected structural pattern matches that of the known design pattern, but not its intent. This happens mainly with the Facade, Adapter and Bridge patterns in our case studies. Consider, for instance, the Bridge pattern of order $o = 3$ in Table 7. A class that has a subclass and accesses another class is a *candidate* for a Bridge, but there is no guarantee that such a class is actually serving the purpose of a *Bridge*.

Identify the neighborhood of a pattern: The neighborhood can be analyzed by navigating through the almost, overloaded, sub and cover patterns. This can be important to detect all candidates for a classical pattern. For example, we consider the *Abstract Composite* pattern of order $o = 3$ of the ADvance application. Applying Filter A ConAn PaDi detects two *Abstract Composite* patterns, but in the neighborhood we find four more *Composite* patterns without an abstract composite root.

Mining patterns: As our approach with FCA detects *any* kind of pattern we found numerous "new" patterns, meaning that they are not referred to in the literature. Whether those patterns are useful and make sense as Design Patterns is another issue.

Identify coding styles: We have noted that occurrence and frequency of certain types of patterns in a system may be a matter of coding style. In our case studies we have seen that CodeCrawler makes heavy use of *Subclass Star* and *Facade*, whereas ADvance is the only application with the *Composite* pattern.

5. Related Work

The starting point for our work was the approach of Tonella and Antoniol [18], and we have already summarized our improvements to their approach. Other related work focuses on the detection of design patterns *à la Gamma* as opposed to more general *software* patterns. We cite some of these approaches.

Brown [2] presents in his Masters thesis a tool to detect design patterns in Smalltalk environments. He explains how to deal with the typeless language Smalltalk. The detection itself is then based on Corman's cycle-detection technique [5]. There is no general abstraction proposed to encode patterns. In particular, Brown does not demonstrate a clearly generalizable approach to detect patterns: for each pattern, a specialized detection algorithm must be developed.

Seemann and von Gudenberg [15] use a compiler to generate graphs from the source code. This graph acts as the initial graph of a graph grammar that describes the design recovery process. The validation is made with respect to well-known design patterns such as Composite and Strategy in the Java AWT package.

Keller et al. [10] present an environment for the reverse engineering of design components based on the structural descriptions of design patterns. Their validation is made with SPOOL on three large-scale C++ software systems. They store the meta-model as UML/CDIF and query then this model for patterns.

Niere et al. [14] provide a method and a corresponding tool which assist in design recovery and program understanding by recognizing instances of design patterns semi-automatically. The algorithm works incrementally and needs the domain and context knowledge given by a reverse engineer. To detect the patterns they use a special form of annotated abstract syntax graph (ASG). Using a subgraph matching algorithm allows them as well to define a pattern neighborhood as we gain out of the lattice. An evaluation of the approach is made with the Java AWT and JGL libraries.

In Krämer and Prechtel's approach [11], the patterns are stored as Prolog rules. Their Pat tool takes the meta-information directly from the C++ header files and queries them. The validation on the C++ libraries shows that the precision is around 40 percent.

order	our approach		[18]	
	Number of Tuples	time [s]	Number of Tuples	time [s]
2	6	0.1	8	0.1
3	7	0.1	18	0.2
4	6	0.1	34	0.4
5	6	0.1	70	2.4
6	4	0.1	140	17.6
7	1	0.1	140	27.5
total	30	0.6	410	48.2

Table 10. Comparison between our inductive approach and the inductive approach from Tonella

6. Conclusions

6.1. Contributions

A complete description of the approach including the analysis of the cases studies and the tool implemented to support it is described elsewhere [3]. Although our work is based on that developed by Tonella and Antoniol, there are some notable differences:

- According to our measurements of Tonella and Antoniol's algorithm, the performance with our data was a critical issue. We propose an improvement to their algorithm to make it faster. We eliminate the redundancy in the sets of elements considered to reduce the calculation time for the formal context generation. Using as example figure 1 (Section 2.1) where there are 7 classes, we have made a comparison of both approaches (shown in table 10) in terms of time performance. The calculation time in the different orders in our approach is uniform , whereas in their approach it increases for each order. The number of tuples also increase with each order in their approach, whereas it remains relatively constant in ours.
- With our improvement in the algorithm where we keep one representative of the set of possible combinations of a class sequence, we avoid repeated information and we avoid to remove *equivalent instances* [18] inside the concepts.
- With our approach, we are not constrained to the detection of *design* patterns. We are focused on the larger scope of detecting recurring collaborations patterns implicit in the code, which we refer to as *collaboration patterns*. These collaborations may represent design patterns, architectural constraints, or simply idioms and conventions adopted for the system.
- In contrast to Tonella and Antoniol's approach, we relate the patterns to each other using the connections between the concepts given by the partial order in the lattice, and the lattices calculated with the different *orders*. This is what we named *pattern neigbourhoods*. For example, it is possible to detect patterns which are *almost* like another pattern.
- We propose to take the information from a language independent meta-model instead from the source code itself. This makes the approach more general because it can be applied to applications in different programming languages. In contrast, Tonella and Antoniol focus on experiments done with C++ applications.
- To gain an overview more quickly, as a starting point, we compare the detected patterns against a reference library of well-known design patterns in the post-processing phase. This library is incremented with new detected patterns with each application we analyze.

6.2. Future Work

- **Enhance the model with information at a higher abstraction level.** Instead of only using the structural information, we could use properties of an higher abstraction level. Such higher-level information could include properties like: isLeaf, isComponent, isFacade. Among the resulting patterns, behavioral patterns might be inferred.
- **Solve the scalability problem.** In an industrial environment ConAn PaDi has to be much faster. Results should be available in real time or at least

within seconds, otherwise the developer will not use this tool. One idea to improve the speed is not to take all orders into consideration.

- **Better name guessing.** Use a better reference library to detect well-known patterns and improve the matching algorithm for them, *i.e.*, making the matching algorithm more fuzzy.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004).

References

[1] T. Aebi. Extracting Architectural Information using Different Levels of Collaborations. Diploma thesis, University of Bern, Sept. 2003.

[2] K. Brown. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk.* Masters thesis, North Carolina State University, 1996.

[3] F. Buchli. Detecting Software Patterns using Formal Concept Analysis. Diploma thesis, University of Bern, Sept. 2003.

[4] E. J. Chikofsky and J. H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[5] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[6] B. Davey and H. A. Priestley. *Introduction to Lattices and Order: Second Edition.* Cambridge University Press, 2002.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2002.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Mass., 1995.

[9] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer Verlag, 1999.

[10] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based Reverse Engineering of Design Components. In *Proceedings of ICSE '99 Conference*, pages 226–235. IEEE Computer Society Press, May 1999.

[11] C. Kramer and L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of WCRE '96 Conference.* IEEE, Nov. 1996.

[12] S. Kuznetsov and S. Obëdkov. Comparing Performance of Algorithms for Generating Concept Lattices. In *Proceedings of International Workshop on Concept Lattices-based KDD*, 2001.

[13] M. Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization.* PhD thesis, University of Bern, May 2003.

[14] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-based Design Recovery. In *Proceedings of the 24th International Conference on Software Engineering*, pages 338–348. ACM Press, 2002.

[15] J. Seemann and J. W. von Gudenberg. Pattern-based Design Recovery of Java Software. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, 1998.

[16] M. Siff and T. Reps. Identifying Modules via Concept Analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.

[17] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96 Conference*, pages 268–285. ACM Press, 1996.

[18] P. Tonella and G. Antoniol. Object Oriented Design Pattern Inference. In *Proceedings ICSM '99 Conference*, pages 230–238, Oct. 1999.