TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Informatics
Chair of the Basics of Informatics

Sergei Opristsenko

INFERRING GOF DESIGN PATTERNS THROUGH FORMAL CONCEPT ANALYSIS

Master's Thesis

Supervisor: Ants Torim, PhD

Tallinn 2014

Author's Declaration

I hereby declare that I have written current thesis independently and it have not been submitted for any degree or examination to any other university. All ideas, major viewpoints and data from different sources by other authors are used only with a reference to the source.

…….. …….

(Date) (Author signature)

Autorideklaratsioon

Deklareerin, et käesolev magistritöö on minu iseseisva töö tulemus ja pole varem kaitsmisele esitatud kusagil mujal. Kõikidele töödele, seisukohtadele, ideedele, mis pärinevad teistelt autoritel ja kasutatud käesolevas töös, on viidatud.

…….. …….

(kuupäev) (lõputöö kaitsja allkiri)

Abstract

Having tools that produce brief but to the point automatic documentation directly from a source code becomes increasingly important, especially for developer who needs to accustom oneself to different projects.

Current thesis attempts to address problem of producing insight of code by focusing on recovering design level information called design patterns. For the one inspecting source code, design patterns could reveal valuable information on the system.

The thesis makes use of data analysis method Formal Concept Analysis (FCA). Through FCA groups consisting of sets of classes could be derived from a source code automatically based on the common attributes/characteristics that sets of classes share. Characteristics of resulting groups would be compared to the "Gang of Four" design pattern library. Additional filtering would be applied to allow flexibility and surfacing of variations of design patterns. Positive matches would indicate existence of patterns, and corresponding sets of classes (pattern instances) would be presented as a result.

Current work has set following goals to fulfill: develop a process that is using FCA to infer GoF design patterns from a Java source code; based on the process construct a tool that could assist beginners when studying design patterns.

The thesis is in English and contains 55 pages of text, 6 chapters, 16 figures, 13 tables.

Annotatsioon

Kui arendaja ülesanne on tundma õppida tarkvara lähtekoodi, siis seisab ta silmitsi probleemiga – kuidas õppida võimalikult efektiivselt. Ainuüksi dokumentatasioonile loota ei saa, sest muudatusi, mis koodiga on seotud, on tihti rohkem kui kajastatakse. Seetõttu teatud abistavate vahendite kasutamine oleks kohane. Vahendite, mis produtseeriksid ülevaatliku ja lühikese dokumentatsiooni automaatselt otse lähtekooodist. Tänapäeva üha kasvava tarkvara süsteemide arvu tõttu, muutub mainitud abivahendite käeulatuses olemine üha olulisemaks.

Käesolev magistri töö ("GoF disaini mustrite avastamine formaalse kontseptianalüüsi meetodil") püüab omapoolse lahenduse anda probleemile, mis puudutab ülevaate saamist koodist. Töö fokuseerub disaini taseme informatsiooni ehk täpsemalt disaini mustrite avastamisele koodist ja ülevaatlikule esitamisele.

Lähenedes tuttavale probleemile kalduvad kogemustega arendajad taaskasutama varasemat tööd. Disaini mustrid on lahendused korduvatele olukordadele või probleemidele disainis. Samas, uurides projekti lähtekoodi esmakordselt, võib mustri juhuslik avastamine anda vihjeid seotud osade rollidest ja probleemidest mida osad tervikuna püüavad lahendada – korduvad mustrid kirjeldavad koodi.

Leidmaks mustreid esindavaid klasside hulki otse lähtekoodist kasutab käesolev töö andmete analüüsimise meetodit – formaalne kontseptianalüüs (Formal Concept Analysis – FCA). FCA eelisteks on muuhulgas, et kattuvate omadustega klasside hulkade leidmist automatiseeritakse. Tulemusena saadud igat unikaalset omaduste komplekti võrreldakse "Gang of Four" (GoF) raamatu mustrite omadustega ning töödeldakse läbi filtri, mis võimaldaks avastada ka disaini mustrite võimalikke variante. Kattuvused viitavad, et ollakse avastanud nii mustri kui ka klasside hulgad mis mustrit kooodis realiseervad.

Töö eesmärkideks on seatud: töötada välja protsess, mis võimaldaks etteantud Java koodist FCA abil avastada GoF disaini mustreid; kasutades arendatud protsessi realiseerida vahend, mis aitaks disaini mustreid õppijatel valitud lähtekoodist mustrite olemasolu tuvastada ja mustreid realiseerivate klasside kohta ülevaate saada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 55 leheküljel, 6 peatükki, 16 joonist, 13 tabelit.

Contents

List Of Figures

List Of Tables

1 Introduction

1.1 Problem Background

Overall number of software projects increases. As a result, those responsible for different maintenance or development goals, enthusiasts, or students of software engineering need more often approach source code of projects yet unknown to them. Right learning strategy to study code efficiently enough is needed. Unfortunately projects have sparse documentation, as documentation might not keep up with faster pace of changes that software projects are affected – the entropy, time and decay will make its changes. What is more, documentation might provide only limited information or might be missing entirely. Therefore, tools that create documentation directly from the code could be highly beneficial in order to help one steer away from relying on documentation and effects of inefficient learning strategy.

By using re-documenting tools that automatically extract valuable information from a code and output overview from different perspective (i.e. design, dependencies) entailing diverse approaches and technologies (i.e. diagrams, maps using analytical or data mining methods), address different granularity level (i.e. metrics, "bird eye view" on whole code), could improve approachability and graspability of code. Also the advancements in computer science – new approaches, technologies, improvements in hardware – could allow altogether richer comprehension of the code than was possible years ago.

Current work addresses problem of producing insight of code by focusing on recovering design level information called design patterns. Design patterns represent expert experience, solutions to problems that reappear. Design patterns are widely used, for the reasons such as: they are catalogued and publicly available; experienced developers gravitate towards reuse, idea which design patterns advocate.

Design pattern present clues on why parts of the code were written – revealing the problem being solved and roles that classes play. To automate finding design patterns existing in source code different design pattern recovery processes have been proposed. Unfortunately no current design pattern processes present perfect enough results.

1.2 Goal Setting

The primary goal of current thesis is to develop a process that uses Formal Concept Analysis to infer "Gang of Four" (GoF) design patterns (found in book Gamma et al. [1]) from a Java source code. Based on the process it should be possible to construct a tool.

Also second goal has been set: using previously developed process achieve in developing a tool that beginner could use when learning design patterns. As learning from real world examples is one of the best ways to learn, novice often faces abundance of choice in the form of open source projects. Therefore when having no prior knowledge one chooses source code randomly. Often it takes time to find out if chosen project is any good for studying, as project might simply lack patterns. In order to accelerate search time, tool is needed simple enough that could assist in giving overview of occurrences of patterns. Once having overview one could discard the code, if project is no good for study purpose. Having found suitable source code, one could rely on the same tool to study every particular instance further, using the info (where pattern instance occurs, what classes are involved, what role classes play) that tool provides. Besides, many tools used today that are detecting (throughout work, synonyms *detecting* and *inferring* would be used interchangeably) patterns, have either steep learning curve (complicated setup, being part of larger more complex tool), provide limited information on instances found (i.e. provide number of instances found but no reference to them). Thesis tries to provide such simple enough to use tool.

In order to reach goal of inferring design patterns current work relies on the data analysis method Formal Concept Analysis (FCA). FCA employs mathematics, which makes it suitable to handle large data sets, and certain philosophical theories, which makes it applicable to any domain of expertise. FCA automatically derives groups of objects that share similar attributes. In parallel, a design pattern could be viewed as a group of classes described by unique set of attributes. For this reason, FCA could be successfully employed to automate finding instances (set of classes) belonging to one or the other design pattern (group). FCA is automating finding groups, according to prepared data set, and attributes. Having results from FCA, actual confirming whether those groups represent design patterns would be done, while comparing found groups with actual pattern library.

1.3 Outline of the Thesis

Current work has been divided into six chapters. The very next chapter would introduce reader to the basics of Formal Concept Analysis. In order to navigate through background knowledge and terms, simplified example would be presented.

Third chapter focuses on design patterns: discusses some benefits and shortcomings of design patterns, different aspects of detection, some possible approaches to detection, tools developed, also the use of FCA. Problems related to design pattern detection are listed.

Fourth chapter describes the solution. Here developed process is explained in detail. By beginning with deciding what characteristics to collect from source code, then continuing with parsing the source code. After constructing context from the collected info, algorithm would be applied on the context for constructing all concepts. As a last step in solution, concepts would be compared with the GoF patterns to collect all possible candidates through the use of different filtering that have been defined, and results would be presented.

Chapter five will evaluate tool that is built based on the process through three aspects: detection precision, time performance, usability. Possible future improvements are described.

Final chapter would conclude whether and how goals have been reached and summarize work done.

Appendixes contains abbreviations used, list of design patterns described through agreed characteristics, mapping information to guide parsing, and finally some evaluation information referenced by current work.

2 Formal Concept Analysis

2.1 Brief Introduction

Formal Concept Analysis represents a theory of data analysis and also branch in applied mathematics. Being introduced by Rudolf Wille in 1981, FCA originated from activities of restructuring mathematical order and lattice theory (Wille [2]) where it has taken most of its mathematics (Ganter and Wille [3]).

Though FCA was built on mathematical background, it has strong philosophical supporting – called concept theory. Mentioned theory presents world through concepts, and enables to define FCA as: "/.../ mathematical theory of concepts and concept hierarchies /.../ to support the rational communication of humans by mathematically developing appropriate conceptual structures which can be logically activated" (Wille [2]). FCA allows to break thinking (specially digested information, a ***context***) into 'units of thoughts' (***concepts***), express meaningful relation and order between units (***subconcept-superconcept relation***) for exploring/discovering additional sub-meaning, also visualize those relations and units (***lattice diagram***).

FCA is a data analysis method. Having gained wider use in different fields, it has found, among others, use in computer science and software engineering. Computers, after all, are generating and processing tremendous amount of data and increasing appetite to analyze and to makes sense of, is a growing trend. FCA presents some means for analyzing such information. Current work benefit from FCA by extracting concepts that could represent different design patterns in code.

2.2 Terminology Explained with Example

2.2.1 Concept

The center of FCA is idea of concepts. There are different aspects to concepts, but current text would only touch subject enough to understand the terminology of FCA used in current work.

Concept could be described as a form of knowledge limited into scope (dictated by its domain) described through objects and their common attributes. In other words,

concepts could be understood as a unit of objects (or unit of thoughts) and attributes that hold some subjective meaning (a knowledge). For instance, concept called *tree* could represent truly tree in a forrest when it has leaves, trunk and roots. In other subjective background all those attributes might have another representation (a *plant*). Or if an object is round, is yellow and warm, might be in one surrounding called a *star* in heaven, but in other, a freshly baked *pancake*. Appears as if through concepts world could be explained.

2.2.2 Formal Concept, Context, Extent, Intent

In FCA, the concept consist of extent and intent. The ***extent*** (in philosophical theory of concepts, called extension), represent all the objects (elements) that belong to boundaries of that concept. The ***intent*** (might be also called intension), represent all the attributes (properties, meanings) for which all the objects in extent hold true.

As mentioned, FCA expresses concepts through mathematics. FCA provides means to derive concepts through calculation from the given formal context (data set, inside which binary relations are specified). In the following, we will turn to mathematics described by Wille [2] (p 2) to define terms: formal context, intent, extent and concept.

Formal context is expressed as a set structure K = (G, M, I), where G is a set of objects (***formal objects***), M set of attributes (***formal attributes***) and I is a set of binary relations between G and M (i.e. $I \subseteq G \times M$). Formal context is best presented through two dimensional table consisting of all objects and attributes (respectively correspond to rows and columns) with the relations that holds between them (marked with crosses).

An example of formal context is presented in Table 2.1, which is based on "Gang of Four" patterns. Formal objects are represented by all creational design patterns (abbreviated according to list given in Appendix A), and formal attributes by possible properties that object could have. We will be using this example hereafter while explaining rest of the terms.

	Instantiation	Inheritance	Delegation	Static component	<4 classes	4 classes	>4 classes
AF	X	X	.	.	.	.	X
BUI	X	X	X	.	.	X	.
FM	X	X	.	.	.	X	.
PROT	X	X	X	.	.	X	.
SIN	X	.	.	X	X	.	.

Table 2.1: An example of formal context (GoF creational patterns)

With the help of derivation operators formal concepts of formal context K could be defined in following manner.

First let us take arbitrary set of objects $X \subseteq G$, then:
$X' = \{m \in M \mid \forall g \in X : (g,m) \in I\}$

In other words, applying operator ' gives attributes that are common for *all* object in provided set X; i.e. {PROT, SIN}' = {Instantiation}. The purpose of this is to approach context with query: *what attributes are common for given objects* (Yevtushenko [4], p6)*?*

Also, let us take set of attributes $Y \subseteq M$, then:
$Y'=\{g \in G \mid \forall m \in Y : (g,m) \in I\}$

In other words, resulting with objects that *all* elements in given attribute set Y have in common; i.e. {Instantiation, Inheritance}' = {AF, BUI, FM, PROT}. The purpose is to find answer to question: *what objects are common for given attributes?*

Now, when there is a pair (A, B) and relations $A \subseteq G$, $B \subseteq M$, $A = B'$, $B = A'$ are satisfied, then pair is called ***formal concept***; A is extent and B is intent. In other words: *give all objects that share the attributes with given objects* (Yevtushenko [4], p6).

For instance, if to derive common attributes of {FM} we get {Instantiation, 4 classes, Inheritance}. And then to derive again common objects of {Instantiation, 4

classes, Inheritance} we get {BUI, FM, PROT}. Now it is not possible to derive any further {BUI, FM, PROT} as we get initial {Instantiation, 4 classes, Inheritance}. ({BUI, FM, PROT},{Instantiation, 4 classes, Inheritance}) forms a concept here.

But ({FM}, {Instantiation, 4 classes, Inheritance}) is not concept because when A = {FM}, A' = {Instantiation, 4 classes, Inheritance}, but when B = {Instantiation, 4 classes, Inheritance} the result derivation is B' = {BUI, FM, PROT}. Therefore $A \neq B'$ ({FM} ≠ {BUI, FM, PROT}).

2.2.3 Calculating Concepts

After construction of the context next step would be to construct concepts. Latter could be done manually, but even for small data sets automation of the process is sensible. For such purpose there are various algorithms.

One of the simplest algorithm to calculate concepts is the intersection method either realized through top-down or bottom-up approach.

Bottom-up approach finds the bottom most concept first and derives concepts from sets holding unique object. Then taking account relationship (subconcept-superconcept relation) and order between concepts that exist in FCA, all upper neighbor concepts are found. Approach is considered to be one of the simplest to understand and to implement, but not efficient for large data sets. To note, there are other algorithms (section 5.3.1) but current work would rely on simplest bottom-up.

Bottom-up approach

Guided by chosen context (Table 2.1) we will use bottom-up approach to illustrate how concepts are found and continue explaining other terms of FCA.

At first, bottom concept is calculated. Being the concept with the maximum intent there is usually no matching extent (presence of object for which all attributes hold true is unlikely).

Step 1. Find BOTTOM concept (M'=?).

c_0=({Instantiation, 4 classes, >4 classes, <4 classes, Static component, Inheritance, Delegation})'=∅

Concepts derived from each single object are calculated next – by moving row by row in object list in context, taking each object and finding corresponding objects that have common attributes with taken object.

Step 2. Derive concept from each separate object or in other words find atomic concepts ((G')' =?).

(({AF})')' = ({Instantiation, >4 classes, Inheritance})'={AF}
(({BUI})')' = ({Instantiation, 4 classes, Inheritance, Delegation})={BUI, PROT}
(({FM})')' = ({Instantiation, 4 classes, Inheritance})={BUI, FM, PROT}
(({PROT})')' = ({Instantiation, 4 classes, Inheritance, Delegation})={BUI, PROT}
(({SIN})')' = ({Instantiation, <4 classes, Static component})={SIN}

Thus found atomic concepts are (second and third line represent the same concept):
c_1=({AF}, {Instantiation, Inheritance, >4 classes})
c_2=({BUI, PROT}, {Instantiation, Inheritance, Delegation, 4 classes})
c_3=({BUI, FM, PROT}, {Instantiation, Inheritance, 4 classes})
c_4=({SIN}, {Instantiation, Static component, <4 classes})

As all concepts are connected with each other either directly or through other concept (they form a *complete partial order*), concepts could be derived from previous and vice versa.

Subconcept A (notated as $\sqsubseteq$) of concept B is concept that have extent smaller than concept's B extent, but intent larger than concept's B intent.

Formally, let there be two concepts (A1, B1) and (A2, B2). Then (A1, B1) $\sqsubseteq$ (A2, B2), if $A1 \subseteq A2$ and $B2 \subseteq B1$ hold true. For instance, c_2 is subconcept of c_3, and c_3 is ***superconcept*** of c_2.

Approaching likewise every two concepts reveal if they are in subconcept-superconcept relation and structure of connections could be built. All concepts based on subconcept-superconcept relation form concept lattice (represented as diagram, Figure 2.1). By moving down, in the direction of subconcepts, two concepts come together under condition of meet or infimum (notated as $\sqcap$) according to:
$(A1, B1) \sqcap (A2, B2) = (A1 \cap A2, (A1 \cap A2)')$

Or when moving up, in the direction of superconcept, join or supremum (notated as ⊔) according to: (A1, B1) ⊔ (A2, B2) = ((B1 ∩ B2)', B1 ∩ B2).
This way ***lower*** and ***upper neighbors*** are found.

Getting back to description of bottom-up algorithm. Any two sets are compared to find pairs of sets not in subconcept-superconcept relation (not *A1⊆A2 and B2⊆B1, and not A1⊇A2 and B2⊇B1*). Paris not in relation and not yet in work list (special list to process) are added to work list. Then each pair in work list will be calculated under join condition (if to express visually, algorithm is moving up, to find upper concepts). Every new concept found would be again compared with previous concepts and additional pairs could be added to work list for processing. Similar concept finding and work list refilling would be done until there is no element in work list to check.

Step 3. Populate initial work list.

To simplify and guide through process of work list elements finding, we will use table in following form to visualize finding concepts not subset or superset of the other.

	c_1, AF	c_2, BUI, PROT	c_3, BUI, FM, PROT	c_4, SIN
c_1, AF	-	⊄, ⊉	⊄, ⊉	⊄, ⊉
c_2, BUI, PROT	-	-	⊂	⊄, ⊉
c_3, BUI, FM, PROT	-	-	-	⊄, ⊉
c_4, SIN	-	-	-	-

Table 2.2: Findings concepts not in superconcept-subconcept relation

Therefore: work list = [(c_1⊔c_2), (c_1⊔c_3), (c_1⊔c_4), (c_2⊔c_4), (c_3⊔c_4)]

Step 4. Start processing work list elements (until new concept is found).

c_1⊔c_2 = ({AF, BUI, FM, PROT}, {Instantiation, Inheritance}) = c_5

Step 5. Update work list.

	c_1, AF	c_2, BUI, PROT	c_3, BUI, FM, PROT	c_4, SIN	c_5, AF, BUI, FM, PROT
c_1, AF	-	⊄, ⊉	⊄, ⊉	⊄, ⊉	⊂
c_2, BUI, PROT	-	-	⊂	⊄, ⊉	⊂
c_3, BUI, FM, PROT	-	-	-	⊄, ⊉	⊂
c_4, SIN	-	-	-	-	⊄, ⊉
c_5, AF, BUI, FM, PROT	-	-	-	-	-

Table 2.3: Findings concepts not in superconcept-subconcept relation (update 1)

Work list = [(c_1⊔c_3), (c_1⊔c_4), (c_2⊔c_4), (c_3⊔c_4), (c_4⊔c_5)]

Step 6. Continue processing work list elements (until new concept is found).

$c_1 \sqcup c_3$ = ({AF, BUI, FM, PROT}, {Instantiation, Inheritance}) = c_5

$c_1 \sqcup c_4$ = ({AF, BUI, FM, PROT, SIN}, {Instantiation}) = c_6

As c_6 *includes all objects, we have also found TOP concept.*

Step 7. Update work list, again.

	c_1, AF	c_2, BUI, PROT	c_3, BUI, FM, PROT	c_4, SIN	c_5, AF, BUI, FM, PROT	c_6, AF, BUI, FM, PROT,
c_1, AF	-	$\not\subset, \not\supset$	$\not\subset, \not\supset$	$\not\subset, \not\supset$	$\subset$	$\subset$
c_2, BUI, PROT	-	-	$\subset$	$\not\subset, \not\supset$	$\subset$	$\subset$
c_3, BUI, FM, PROT	-	-	-	$\not\subset, \not\supset$	$\subset$	$\subset$
c_4, SIN	-	-	-	-	$\not\subset, \not\supset$	$\subset$
c_5, AF, BUI, FM, PROT	-	-	-	-	-	$\subset$
c_6, AF, BUI, FM, PROT,	-	-	-	-	-	-

Table 2.4: Findings concepts not in superconcept-subconcept relation (update 2)

Work list = [($c_2 \sqcup c_4$), ($c_3 \sqcup c_4$), ($c_4 \sqcup c_5$)]

Step 8. Continue processing work list elements, again.

$c_2 \sqcup c_4$ = ({AF, BUI, FM, PROT, SIN}, {Instantiation}) = c_6

$c_3 \sqcup c_4 = c_6$

$c_4 \sqcup c_5 = c_6$

Step 9. After elements in the work list have been processed produce concepts.

BOTTOM=c_0=({∅}, {Instantiation, Inheritance, Delegation, Static component, <4 classes, 4 classes, >4 classes})
c_1 = ({AF}, {Instantiation, Inheritance, >4 classes})
c_2 = ({BUI, PROT}, {Instantiation, Inheritance, Delegation, 4 classes})
c_3 = ({BUI, FM, PROT}, {Instantiation, Inheritance, 4 classes})
c_4 = ({SIN}, {Instantiation, Static component, <4 classes})
c_5 = ({AF, BUI, FM, PROT}, {Instantiation, Inheritance})
TOP=c_6=({AF, BUI, FM, PROT, SIN},{Instantiation})

Table 2.5: Resulting concepts for example context

2.2.4 Drawing Lattice Diagram

To add visual and intuitive aspect to analyze process resulting lattice is usually visualized by labelled diagram (also called Hasse diagram or line diagram (Yevtushenko [4], p 8)). On such diagram every concept is represented by node and the subconcept-superconcept relation with its direct neighbor concept is depicted by line.

There are many ways to represent lattice diagram. Simplest is the full labeling form, where all the extent and intent components are shown. But is considered to overwhelm with information, especially for larger contexts. Most often lattice is presented with reduced labeling (Ganter and Wille [3], p 3) technique (see diagram for previous example on Figure 2.1, created with ConExp tool (Yevtushenko [5])).

In case of reduced labeling every attribute and object occurs only once. The labels that represent each node could be found as if moving twice through full labeling diagram, down and up: when moving down, knowing concepts where certain attribute exists only on the concept where that attribute first occurs that attribute is shown on diagram, rest appearances are not shown; similar is done when moving up when adding objects to labels, only first appearance of object is shown. Later, knowing how information was left out, one can collect all concept information likewise in reverse.

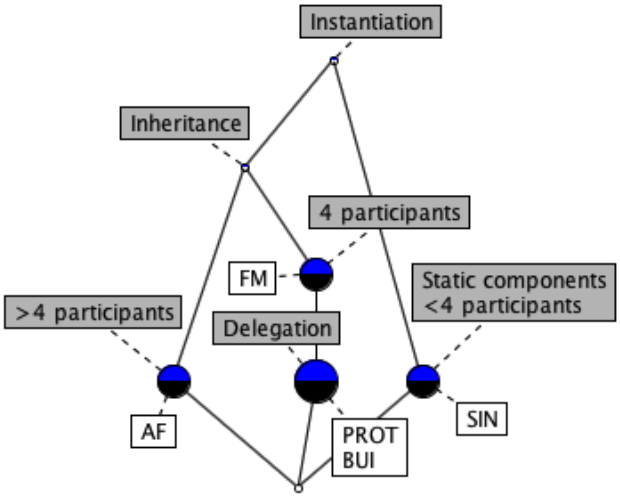

Figure 2.1: Concept lattice for the creational patterns example

Lattice does not reduce complexity and retains all original information presented in context, therefore even for not very large size contexts diagram becomes hard to grasp. Approaches exist to overcome and to organize/present/browse huge lattice with reduced complexity. One approach is to show only parts of the lattice (i.e. Eklund et al. [6] shows current concept with upper and lower neighbors when navigating between exhibits in virtual museum).

In current work lattice diagrams are not implemented for presenting information, instead they are used in section 4.6 for explanatory purpose.

3 Design Patterns and Detection

3.1 Design Patterns

Design patterns are collections of documented design approaches. They are solutions to reappearing problems that have emerged from work of many developers. They are advocating idea of reuse.

To begin with example, let there be program that needs to be built to represent large amount of objects, such as stars in simplified simulation of cosmos. Large number of similar objects needs to be created and destroyed. If design is realized in too complicated or naive manner, performance could suffer. After analyze becomes clear that objects entail differences but they also appear to share common components. To be able to avoid performance issues, one might consider to permit reuse of objects. As situation is described and brought into attention, the solution or approach could be found to be already existing by one of such design pattern (i.e. Flyweight), and so applied. By using already tried approach could mitigate the effects on performance by reusing objects when large number of objects needs to be presented.

For another example of known situation to consider is a program representing directory structure, where directory contains many others elements (i.e. other directory) which yet consist of others and so on. How to design and possibly implement directory structure, which appear to have some recursive properties? For solution there could be already pattern (i.e. Composite) that allows to overcome the problem.

Such collection of solutions are found by experts investigating similar code coming up time and time again. As the solutions worked well, thus they were collected and collection of new kind of experiences emerged. "Gang of Four" book by Gamma et al. [1], containing 23 patterns, serves as one such collection and is among the first that have introduced design patterns notion to wider public of software engineering. The term design pattern was originally borrowed from the field of architecture, from works (The Timeless Way of Building, A Pattern Language: Towns, Buildings, Construction) of architect Christopher Alexander. Though GoF book was published 20 years ago, design patterns idea (or use of design pattern language) have been well adapted and is evolving in software engineering on many different aspects.

As of presenting book Gamma et al. [1], there have been many other patterns introduced in software engineering (they are created either for different domains of applications (i.e. gaming, security, web), based on how much detail or granularity they contain, or classified/categorized/grouped in other sense making way).

As with the example of cosmos, analyze might report that Flyweight is not efficient enough choice. Maybe there is need for other and more sophisticated pattern that allow better performance gain, such that makes use of latest computer processing capabilities. Such could be one that makes possible use of concurrency.

Use of design patterns could represent benefits but also shortcomings. From the side of benefits, design patterns first of all propagate reuse. Reuse in software engineering is important idea – to not reinvent solutions to problems which already have quite efficient solutions, found by developer himself or by other developers in past. As finding right design solution takes time, often many attempts, reuse of previous knowledge or experience is most welcome. What is more, by having clear background knowledge of the solution (pattern) and situation (problem, background, purpose, intent), parts of software could be built faster.

Design patterns can not be taken as a definitive guide to guarantee working solution in every situation, they are merely expert found principles to follow when applicable. GoF book has presented patterns in a way where patterns are organized into templates and categories. It is not the only way to organize patterns and can be challenging to read especially for novice who studies patterns.

3.2 Detection Approaches, Tools and Problems

Just after first design pattern catalogs were published works attempting to detect design pattern directly from the code emerged (first work on detection by Krämer and Prechelt [7], according to Rasool and Mäder [8], p 243). By now many tools have been developed (to note, different works might be using different synonyms to refer to "detection", synonyms as inference, recognition, exploration, or mining).

3.2.1 Tools

Considerable number of tools have been developed for inferring design patterns. Tools differ in recovery precision, in patterns they cover, how results are presented (i.e.

provide number of patterns detected but do not provide location of instance), how known detection problems are tackled (section 3.2.3), what techniques/methods/approaches were used, programming language applicable to, in usability, ease of set up, etc. None of the tools produce perfect results, because of the problems related with detection (most listed in section 3.2.3). For the reason that large number of different tools and approaches exist, author decided to leave out the complete overview (could be consulted from Rasool and Mäder [8], Dong et al. [9]) and instead mention approaches and tools which are closely related to current work.

3.2.2 Related Approaches and FCA

Many works (i.e. Lee et al. [10], Heuzeroth et al. [11], Rasool and Mäder [8]) separate tasks into two groups: static analysis and dynamic analysis. Former dealing with the scanning the structure of the code and passing initial analyze resulting with candidate list. Latter collect info when code is run, adds more precision to detection by filtering candidates with extra conditions. Also there could be additional approaches/technologies that differ between works.

- Static structural analysis – source code is presented in structured model or form that could make information in code modularly approachable and queryable. Some source code modeling techniques, such as AST (Abstract Syntax Tree), ASG (Abstract Syntax Graph), have been used in such cases.
- Dynamic behavioral analysis – as some patterns are described by behavioral characteristics, such as method call tracing, which can not be extracted from structural info, additional information is extracted through another set of technologies (i.e. JDI – Java Debug Interface) while code is being executed. Pattern instances found during static analysis, could be now further filtered.
- Additional approaches/technologies. For instance:
    - "Pattern candidate rating" (applied between static and dynamic analysis) approach is used by Detten et al. [12]. By measuring each candidate found according to how well they follow the specification of pattern, they are not restricting to firm detection process, so allowing flexibility in detection of variants and giving possibility to present info on how reliable findings are.

- Rasool and Mäder [8] redefines each design pattern through set of repeating attributes (called "feature types") which reflect structural, relational and behavioral characteristics of the patterns; to find pattern instances meeting certain criteria (agreed set of feature types representing known pattern) either source code is queried with source code parser (using compiler generator) or queried through model representing source code via SQL queries.

Current work could be viewed as consisting of static and dynamic analysis, and additional approach. At first, code is parsed to collect info (agreed characteristics (section 4.2) are collected) with AST. Then dynamic analysis could add more information *(this phase was left out from current work, because of time scope)*. Later FCA (additional technique used) is applied to extracted info to automatically derive groups of instances (pattern candidates) holding similar characteristics.

FCA has been used in software development in different aspects and in the scope of DP detection following works were studied.

- Tonella and Antoniol [13] uses concept analysis to extract set of classes that have similar relations, attributes, and without intentionally aiming to look for certain patterns in code (without consulting design pattern library). Having found set of classes with similar attributes, attributes could be then compared with library to confirm existence of design pattern.
- Buchli [14] approaches similarly to Tonella and Antoniol, and infers patterns with FCA without any library knowledge. Aside detection logic different set of GUI tools were created for context editing, lattice/concepts browsing for visual validation of patterns. Tools are implemented in Smalltalk and are part of larger framework that collections different other re-engineering tools.

FCA addresses some obstacles (described by next section), such as detecting overlapping patterns, detecting variants, or patterns consisting of another pattern (other tools could miss as described by Rasool et al. [15], p 817). In addition, FCA does not need to consult library of design patterns, instead infers possible modules that happen to represent candidates automatically (benefit stressed by Tonella and Antoniol [13]). What is more, FCA could be used to find yet undocumented patterns.

Ability to find unknown and also the known patterns, depends largely on consciously chosen characteristics that would represent attributes in FCA context. Therefore first of all specific scope and goal needs to be defined that would dictate the choice of characteristics. Within current work, attributes would reflect characteristics of design patterns, that can be read from formal GoF design pattern definitions (further in section 4.2.1).

Nevertheless FCA adds one obstacle of its own: computational complexity raises considerably when large number of concepts are involved (Buchli [14], p 55).

3.2.3 Problems with Detection

Following is a list of potential obstacles that may need to be addressed when attempting design pattern detection.

1. Design pattern could have very abstract definition (for this reason detection needs to be as flexible as possible, leaving room for different interpretations):
   - pattern could have variations (or variants) that represent different interpretations of the pattern, but that still follow specification (there is no full catalog of those variations available (i.e. Rasool and Mäder [8], p 246));
   - sometimes no definition available on what parts of design patterns are reliably fixed and what parts allow varying interpretation (i.e. instead of Abstract parent class concrete class is used, nonetheless inheritance relation dictated by definition of pattern could still hold).
2. Similarity or overlapping in structure, or one pattern inside another:
   - similar by structure but different purpose (Strategy and State);
   - overlapping structure (by definition one patterns could be part of another (i.e. Composite inside Decorator));
   - pattern could be hidden, part of another pattern (Singleton inside Builder, Abstract Factory or Prototype).
3. Different programming language in use. Programming languages differ in syntax, semantics and language constructs they use to represent object-oriented idea (i.e. language might not allow to inherit from two classes). Design patterns might have details that can not be realized in one language but can be in other

(i.e. double inheritance with Adapter class scope pattern).

4. Not up to date definition text:
   - GoF book was written decade ago, and some information is not topical and up to date, and hence difficult to fully comprehend, allowing misinterpretation;
   - programming languages evolve and they include more features (old detection tools might not be flexible to future changes).
5. Large systems have a lot of code. Performance issues – tools take considerable time to analyze, might need improved algorithms, approaches, better hardware resources.
6. New DP appear. DP detection processes/tools might not allow to detect new design patterns, if there is no possibility to modularly add new patterns to detection information.

To consider listed problems and primary goal that have been set, current work attempts to detect only design patterns of GoF book (Gamma et al. [1]), overcome some mentioned problems with the help of FCA, discover variations of patterns, and analyze only source code written in Java language.

4 Solution

4.1 Overview of Approach

Current work bases its approach on the works of Buchli [14] and Tonella and Antoniol [13], where FCA have been used to identify concepts that represent design patterns. Every concept would hold unique set of attributes in its intent, and sets of classes corresponding to these attributes in its extent.

Central to current approach is design pattern detection process (Figure 4.1). First of all, characteristics to be collected from source code are defined. Then source code is parsed and modeled in Abstract Syntax Tree from where actual info is collected. Based on the extracted class info, context is built consisting of sets of classes and the characteristics they hold. FCA concepts finding algorithms is applied – resulting with list of concepts (pattern candidates). Finally each concept and their corresponding unique set of attributes are compared with the known pattern definitions (pattern library). Positive matches indicate known design pattern found (detection precision and flexibility should depend on the filtering applied), and objects of that concept will indicate actual instances (sets of classes realizing this pattern).

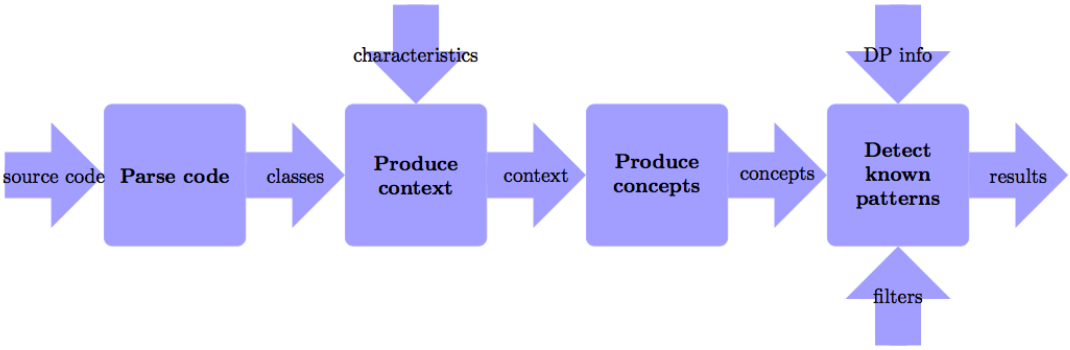

Figure 4.1: Overview of detection process

4.2 Characteristics

It is clear that every design pattern is somehow unique and then again, some properties (***characteristics*** from now on) are overlapping. But, how to describe design patterns? What possible ways are there to describe patterns for detection processes?

For possible answers to these questions, other approaches were studied:

- Rasool and Mäder [8] re-classifies design patterns through set of recurring attributes (called feature types); 44 different feature types (such as: pattern "has class", pattern "has super class") were used to describe every design pattern. Though authors also involve variants of design patterns, work presents overview of how many different attributes were needed for describing all patterns.
- Shi and Olsson [16] have classified patterns into five categories according to search strategy to apply. Structure driven category patterns are identified by containing: declarations, generalization, association, delegations. Behavior-driven patterns could be specified by properties such as: object creation, number of objects created (aggregation). Language-provided patterns concern by detecting built in classes, interfaces, that programming language libraries provide. Domain-specific patterns need additional information provided by domain where the software is used. Generic concepts patterns cover patterns that are too general to detect and are difficult to specify their detectable aspects.
- Tonella and Antoniol [13], based on FCA, used initially simple attributes possible (structural, such as association and inheritance), then added more non-structural attributes (such as to represent if class x owns method m, if class x calls method m of another class x2) for enriching characterization of patterns found and decreasing number of false positives. Buchli [14] stresses also the need to keep characteristic choice simple and concentrate only on some attributes, as to avoid unnecessary raise in complexity.

Additionally instead of formal definition, pattern could be implemented in the form of variation (as described by problem 1, section 3.2.3). For instance, Adapter pattern could make use of interfaces, abstract classes, or mix of them. On the one hand, there is possibility, that if to classify design pattern too strictly (i.e. too many characteristics), then detection could miss variations. In order to overcome this, some flexibility to detection should be added. One solution to this problem is to disable some characteristics (for instance, some patterns might have "Client" participant which could be discarded), the other is to have record of each such possible variant separately. On the other hand, if to specify too few characteristics, detection could place found

positives to different patterns (false positives). Latter could happen, for example when structural properties of design patterns overlap (also described by problem 2, section 3.2.3).

The subconcept-superconcept relations could come into advantage, as it allows to find concepts that have slightly less ("almost" patterns, Buchli [14], p 28) or more characteristics ("overloaded" patterns, Buchli [14], p 29) with less effort – by following the direct neighbor relation. In that case more patterns could be found (also variations), adding some flexibility to detection process.

Alternatively variations could be found by additional filtering, where each filter will target different set of characteristic (specifics in section 4.7). This should give room for finding patterns even when candidates with precise ("exact" pattern) characteristics that follow the formal definition of patterns are not present.

4.2.1 Deciding Characteristics to Collect

Following guidelines were defined to keep choice of characteristics within limits of scope:

- Characteristics to collect would be dictated by definition of known design patterns from GoF book (Gamma et al. [1]).
- Context would look similar to example of Buchli [14] and Tonella and Antoniol [13], where sequences of classes represent objects (i.e. {ACD}, where A, C, D are class names). Characteristics (i.e. isAbstract, isReferringTo) plus indexes (that refer to specific classes in corresponding sequence), would represent attributes (details in section 4.3).
- Collectable characteristics are affected by how or if at all Java syntax supports such characteristic – it must be after all collectable from Java code.
- Define not too few but also not too many characteristics.

Analyzing Design Patterns: Static Aspects

Most of the design patterns can be viewed as interconnected classes. They are relying on the notion of *inheritance* (either one class is superclass of other or one implements interface) or *reference* (in Java, a reference variable could refer to single object or group of objects (array, collections); or reference, returned by method return type).

In GoF book (Gamma et al. [1]) every pattern is described by agreed format – template. One section in template is called Structure. Structure uses OMG (Object Modeling Technique) type class diagrams to provides compressed overview of the pattern structure. Relying first of all on mentioned diagrams and then on other template information, design patterns would be now analyzed and reasoned in our attempt to choose characteristics that would describe static aspects.

Inheritance Relations

Regarding inheritance there are two different aspects to consider: class inheritance and interface inheritance. According to book, numerous design patterns depend on the distinction of the two (Gamma et al. [1], p 17). Class inheritance is mechanism to extend functionality by reusing functionality in parent class. With interface inheritance class can use interface as a set of requests which it can respond to (Gamma et al. [1], p 16). Having no implementation, interface is also called *pure* abstract class.

Many design patterns rely on inheritance (except Singleton, Facade, Memento), but some depend on distinction of class and interface inheritance. For instance, some patterns could make use of interfaces (Strategy, Sate, Composite, Abstract Factory), while others can have only part abstraction (Template Method). Then again, some patterns could be constructed via combination of both: part or pure abstraction (Adapter, class scope pattern). Therefore choosing characteristics that would be considering such distinction, would allow more accurate detecting.

From the inheritor point of view inheritance relations between classes in Java is realized through two distinct keywords: *extends* (to indicate relation to superclass) and *implements* (to indicate relation to interface). While from the parent point of view through keywords: *abstract class* (to indicate superclass) and *interface* (to indicate super interface).

From inheritor point of view to represent relation when class is inheriting from the other, characteristic isInheriting(1, 2) could be collected (from now on, let 1 and 2 represent indexes of hypothetical sequence of classes (A, B); so isInheriting(1, 2) could be read: class 1 is inheriting from class 2).

Also some design patterns (or their variants) contain either pure abstract classes (java construct: interface), part abstract classes that could contain some implementation (java construct: abstract class), and *concrete class*es that contain no abstraction.

Therefore from parent point of view characteristics isAbstract(1) and isInterface(1) could be collected.

There is more information that inheritance could reveal. For instance, the inheritor explicitly overrides method from the parent for most of the patterns (except Mediator) that rely on inheritance. **Here another characteristic called overrides(1, 2)<M> (could be read: class 1 overrides method M from class 2) could be added.**

Reference Relations

Focusing on reference relation of patterns presented on diagrams (referring to Structure). Being described "plain arrowhead line indicates that a class keeps a reference to an instance of another class" (Gamma et al. [1], p 21), all patterns that include mentioned line must hold reference in their representing code. But additionally, some relations in diagrams represent aggregations – "an arrowhead line with a diamond at its base denotes aggregation" (Gamma et al. [1], p 23). First comes to mind to ask: what is the difference? Plain arrowhead describes acquaintance and arrowhead with diamond describes aggregation. For the reason that distinction between those two is blur (design pattern could be defined as with aggregation but at the same time alternatively as with simple acquaintance reference (Gamma et al. [1], p 23)), to not complicate detection process, aggregation and acquaintance could be represented by one characteristic.

From the perspective of Java constructs, reference to another object could be realized in Java through *simple reference variable*, an *array* holding references to objects and reference to *collection* holding list of references to objects.

- Simple reference variable (in UML context, called simple "association"). Structure diagrams depicts this as plain arrowhead line.

  Many patterns are using mentioned simple mechanism: Abstract Factory, Factory Method, Prototype, Singleton, Adapter (class and object), Bridge, Component, Decorator, Facade, Flyweight, Proxy, Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Visitor.

  Also, Builder, Decorator, could use simple reference variable (instead of the more formal aggregation) (example respectively, p 101–104, p 180–182, Gamma et al. [1]). Same with State, Memento (in case when Caretaker needs to

remember only last memento).

Only Template Method could be without reference (relying more on inheritance and part abstraction).

- An array of references or references via collection (in UML, aggregation). Following patterns could include aggregation: Builder, Bridge, Composite, Decorator, Flyweight, Command, Interpreter, Memento, State, Strategy.

To conclude, characteristics representing collection and simple reference could be described by one characteristic: isReferringTo(1, 2).

Method *return type*. Abstract Factory, Factory Method, Prototype, Iterator, Memento could depend on what type of object method returns. For instance, Factory method pattern has method FactoryMethod() that has return type of specified product class. **To be able to know method return types characteristic hasMethodWithReturnTypeRefTo(1, 2) could be collected.**

Analyzing Design Patterns: Behavioral Aspects

Relying first of all on class and object diagrams under Structure, interaction diagrams under Collaboration (Gamma et al. [1]) and then on other template information, design patterns would be now analyzed to choose characteristics that would describe behavioral aspects.

Regarding references to newly created object (in UML, *instantiation* of objects) – all Creational patterns, but also Flyweight, Command, Iterator, Memento create objects. **Characteristic creates(1, 2) could be collected to describe one class creating instance of the other class.**

Method from one class could call method from another class. Many patterns describes such behavioral side between participants (Builder, Command, Memento, Observer, Visitor) or calling of other class' method is depicted in Structure diagrams (Builder, Prototype, Adapter (object), Bridge, Composite, Proxy, Command, Memento, Observer, State, Visitor). **For previous reason following characteristics could be added: calls(1, 2)<M> (class 1 calls method M of class 2), calls(1, 2)<M1,M2> (method M1 from class 1 calls method M2 from class 2).**

Object of type class could be created by certain method. **Another characteristic could be defined here: createsIn(1, 2)<M>.**

Some overridden method could actually return different type (but still subtype of parent) than what was declared by parent class (which is legal and allowed from Java 5). **New characteristic could be: hasMethodActuallyReturns(1, 2).**

Actual Characteristics To Collect

At this point number of characteristics have been suggested, but in order to keep approach as simple as possible, detection process would be limited to following characteristics (Appendix B, Table B.12): ***isInheriting(1, 2)***, ***isAbstract(1)***, ***isInterface(1)***, ***isReferringTo(1, 2)***, ***hasMethodWithReturnTypeRefTo(1, 2)*.**

4.2.2 Characterizing Design Patterns

Previously characteristics were defined to be extracted from Java files. Based on them FCA concepts would be constructed. But in order to get concepts representing known patterns, there is need to compare concepts with the library of actual design patterns. For this reason each design pattern would be re-classified by characteristics previously given and additionally by guiding characteristics.

Adding Guiding Characteristics to IImprove Filtering

If to restrict to previous characteristics only, the variations or instances that lack some characteristics but could still be representing known pattern, might be missed. As mentioned at the beginning of the section 4.2 , such solution would be provided by filtering. Due to this reason additional characteristics will be added, called *guiding characteristics*. Guiding characteristics could allow to search design patterns from different viewpoints (from now on ***filters***). These characteristics would not be used by FCA or code parser when collecting info on Java files but by final comparing process (section 4.7) that compare concepts with pattern library.

As already mentioned, some patterns might contain participant "Client". This role could be conditioned and removed in some cases, as Client is not always present. **Therefore, to depict that that certain design pattern has role Client, characteristic isClient(1) could be added.**

Other possible characteristics to add could be: canBeConcrete(1), canNotBePureAbstract(1), refCanBeCollection(1) (summary of characteristics in

Appendix B, Table B.13). But to limit once again, work makes use only ***isClient(1)*** characteristic.

Design Patterns Characterized

After agreeing on characteristics GoF patterns could be re-classified.

Patterns would be organized according to minimum number of participants (Appendix B, Table B.14). For convenience (some logics of context creating could be reused here), patterns and their characteristics are presented through binary relation table (Appendix B, Table B.15 – B.21), similar to how context is represented containing information collected from code. Below combined table (Table 4.1) is given, where all patterns and characteristics describing them are joined together. Furthermore, resulting format enable to add more characteristics and new patterns (thus addressing problem 6, section 3.2.3).

	>=2 participants	>=3 participants	>=4 participants	>=5 participants	>=6 participants	>=8 participants	>=10 participants	(1)a	(2)a	(3)a	(4)a	(5)a	(6)a	(8)a	(2,1)inh	(3,1)inh	(3,2)inh	(3,4)inh	(4,2)inh	(4,3)inh	(5,2)inh	(5,3)inh	(5,4)inh	(6,4)inh	(6,5)inh	(7,5)inh	(7,6)inh	(8,6)inh	(9,8)inh	(10,8)inh	(1,2)r	(1,3)r	(1,4)r	(1,5)r	(1,8)r	(2,2)r	(2,3)r	(2,4)r	(2,5)r	(3,1)r	(3,2)r	(3,4)r	(4,2)r	(4,3)r	(5,3)r	(5,4)r	(5,6)r	(1,2)hmrt	(2,2)hmrt	(2,4)hmrt	(2,5)hmrt	(2,8)hmrt	(3,1)hmrt	(1)client
AF	.	.	.	.	.	.	X	.	X	.	.	X	.	X	.	.	X	.	X	.	.	.	.	.	X	X	.	.	X	X	X	.	.	X	X	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	.	X
BUI	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
FM	.	.	X	.	.	.	.	X	.	X	.	.	.	.	X	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.
PROT	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	X
ADAo	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	X
BRIDG	.	.	.	.	X	.	.	.	X	.	X	.	.	.	.	.	X	.	.	.	.	.	X	X	.	.	.	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X
COMP	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	X
DECOR	.	.	.	X	.	.	.	X	.	X	.	.	.	.	X	X	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.
FLYW	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	X	.	X	X	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X
PROX	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	X
CHOR	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X
COMM	.	.	.	X	.	.	.	.	.	.	X	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	X	.	.	.	.	X	.	.	.	.	.	.	X
INTERP	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	X	X	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	X
ITER	.	.	.	X	.	.	.	.	X	.	X	.	.	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	X	.	.	.	X
MEDI	.	.	.	X	.	.	X	.	X	.	.	.	.	X	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
MEM	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	X	.	.	.	.	.	.	.	.	.
OBS	.	.	X	.	.	.	X	.	X	.	.	.	.	X	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.
STAT	.	.	X	.	.	.	.	X	.	.	.	.	.	.	.	.	X	.	X	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
STRAT	.	.	.	X	.	.	.	X	.	.	.	.	.	.	.	.	X	.	X	.	X	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
TM	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
VISIT	.	.	.	.	.	X	.	.	X	.	.	.	X	.	.	.	X	.	X	.	.	.	.	.	.	.	X	X	.	.	X	.	.	X	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	X

Table 4.1: Characteristics of all patterns (except Facade and Singleton)

4.3 Context Building

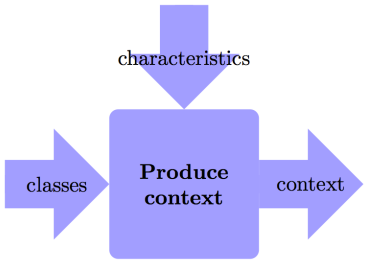

Figure 4.2: Context building

Every context is custom to problem scope. For current background the aim is to analyze source code in order to detect repeating design patterns. For such mentioned scope current work takes example of the work of Tonella and Antoniol [13].

Tonella and Antoniol [13] presents context in following way. Formal object is represented by sequence of classes and every sequence must have same fixed length *o* (***order*** *o*). Formal attribute of context is representing relation R (belonging to group of predefined relations (in current work *characteristics*); i.e. represent association or inheritance) between classes in sequences, indexes, instead of class names are used for generalization and possibility to reuse attributes. Hence attribute is corresponding to certain object for which relation defined by attribute holds (i.e. (1,2)e – first class in sequence extends the second class).

Tonella and Antoniol ([13], p 4) devise algorithm called inductive context construction algorithm. The algorithm collects all variations of **order** *o* (i.e. *o*=3) from all available list of n classes (all classes to be analyzed). Meaning, if to analyze 6 classes, result is 120 sets ($V_n^o = n!/(n-o)!$), each containing 3 elements. Next allow to show only those sets, where element within each set is connected or related with the other elements in same set through at least one relation R (thus number of sets decreases). Tonella and Antoniol's algorithm allow elements in class to have different order (permutation allowed) (i.e. set {BAC} could coexist with {BCA}). Buchli [14] on the other hand, modified the algorithm to allow only combination of elements to occur (i.e. having already {BAC}, {BCA} should not exist), therefore decreased number of objects in context.

Current work approaches differently and separated tasks related to context creation into parts and devises algorithms for each to be executed in order. Three of the algorithms are:

- Formal context object creation (Figure 4.3). Responsible of constructing objects. Relation R would represent all possible characteristics that were defined previously (section 4.2.1). Attributes will be put together from variations of given order *o* (LEVEL in algorithm), by 2 elements (if order 3, then isInheriting(1,2) indexes could have following orders (1,2), (1,3), (2,1), (2,3), (3,1), (3,2)). Additional logic applies for exceptions: if isReferringTo or hasMethodWithReturnTypeRefTo, then recursive reference can exist (i.e. (1,1), (2,2), (3,3)).
- Formal context attributes creation.
- Finding binary relation between above two (Figure 4.4). Determines binary relation between objects and all possible attributes.

```
origSET ← getAllClasses()
LEVEL ← number of classes to be in every sequence
R ← set of all possible characteristics (isInheriting, isReferringTo,
hasMethodWithReturnTypeRefTo) that might connect two classes

#calculateContextObjects:
IN: origSET, LEVEL
objSET ← origSET
for 1 to LEVEL do
        objSET ← calculateNextLevel(origSET, objSET)
OUT: objSET

#calculateNextLevel:
IN: origSET, SET1
for all γ ∈ SET1 do
        SET2 ← {∅}
        for all i ∈ origSET − { ∪ ∀j ∈ γ} do
                SET2 ← SET2 ∪ i
        for all ε ∈ SET2 do
                SET3 ← γ ∪ ε (set γ elements plus element ε)
                if (∃k ∈ γ, (ε, k) ∈ R ∨ (k, ε) ∈ R) ∧ ∄permutation(SET3) ∈ SET4
                  then
                        SET4 ← SET4 ∪ SET3
OUT: SET4
```

Figure 4.3: Algorithm for calculating context objects

```
objSET ← all sequences of classes (formal objects)
attrSET ← all possible attributes (formal attributes)
R ← all characteristics that describe class (isAbstract, isInterface) or relation between
classes (isInheriting, isReferringTo, hasMethodWithReturnTypeRefTo)

#fillContextCrossTableWithValues:
IN: objSET, attrSET
ROWS ← {∅}
for all α ∈ objSET do
        ROW ← {∅}
        for all attr ∈ attrSET do
                position1 ← attr.getPosition1()
                R ← attr.getR()
                if  R.isUnaryRelation()
                        if (α.getElementAt(position1)) legal for R then
                                ROW ← ROW ∪ {X}
                        else
                                ROW ← ROW ∪ {.}
                else if  R.isBinaryRelation()
                        position2 ← attr.getPosition2()
                        if (α.getElementAt(position1), α.getElementAt(position2))
                           legal for R then
                                ROW ← ROW ∪ {X}
                        else
                                ROW ← ROW ∪ {.}
        ROWS ← ROWS ∪ ROW
OUT: ROWS
```

Figure 4.4: Algorithm for setting crosses between objects and attributes

4.4 Parsing Source Code: Collecting Data From Java Code

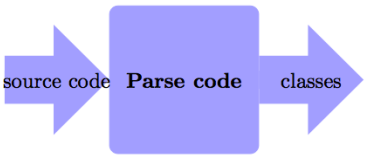

Figure 4.5: Parsing source code

Initial step in detection process would be gathering of information from the source code. For every characteristic decided to collect following questions are asked:

- what information to collect from source code to match characteristics?
- what are the means (technologies) used to collect such information?

Knowing characteristics (section 4.2) enables to specify how Java syntax is realizing such characteristics and then how Java syntax construct match with Abstract Syntax Tree model counterpart (represented by Java Development Framework). Java

code parser would analyze source code's AST to fetch characteristics agreed. After information has been collected resulting context could be constructed.

AST is modeling source code in the form of tree with predefined components/types of nodes allowing modular access, thus allowing to collect details on source code in more organized manner than parsing directly source code could allow. JDT framework was chosen because: includes API for AST and though not newest and not only approach (other approaches are not looked into in current work), it is reliable framework to use – has documentation and includes AST support for Java 7 used in current work.

4.4.1 Mapping Characteristics with Java and AST – Static Aspects

Characteristics, concerning static aspects, where data can be collected without executing code, were covered. Tables were devised (given in Appendix B) which summarize all possible occurrences of characteristics in Java code that were covered. For each such occurrence Java construct was mapped with AST counterpart.

4.4.2 Dynamic Aspects

Due to time scope current part was decided not to be implemented in current work.

4.5 Calculating Concepts

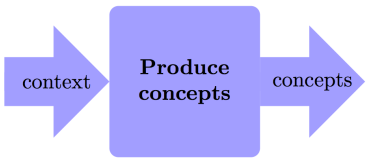

Figure 4.6: Calculating concepts

There are different variants of algorithms to calculate concepts, either top-down or bottom-up approach (one of them created by Siff [17], which was used by Tonella and Antoniol [13], p 3) or some more difficult to comprehend but faster method. However, current work devises own algorithm (Appendix C, Figure C.2) based on the simple bottom-up approach. In parallel, ConExp tool (Yevtushenko [5]) was used to confirm that number of concepts were correct.

4.6 Correcting and Merging Concepts

Problem occurred where sequences are placed into different concepts though represent the same pattern. These sequences consist of different combination of classes and are described by different attributes (thus appear in different concepts).

Same unwanted duplication of concepts were also reported by Buchli [14] (p 26-28), and Tonella and Antoniol [13] (p 233-234). Mentioned works solved problem by post filtering, where duplicate concepts were merged. In current work, different solution would be implemented. The solution would be to remove duplicates when context is being built. In case duplicates still occur in some contexts, query and filtering system (described by section 4.7) could remove them.

Solution: change the order (find new permutation) of classes (affecting corollary change in attribute indexes) in one sequence so that attributes match exactly with attributes of another sequence.

When reordering attributes of one sequence there could be existing sequence with exact same attributes. If this match exists, the original sequence would get new order. This would place matched sequences into same concept already in the phase of context creating.

To explain solution, hypothetical code consisting of 1 composite and 2 adapter patterns is used (Figure 4.7) as an example.

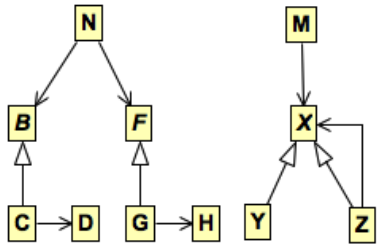

Figure 4.7: Class diagram for hypothetical source code

After constructing context for above code, while order is 4, and analyzing the lattice diagram (Figure 4.8) sequences BFGN and BCFN appear to represent the same pattern, only the order of the classes in each sequence is different as shown on Figure 4.9.

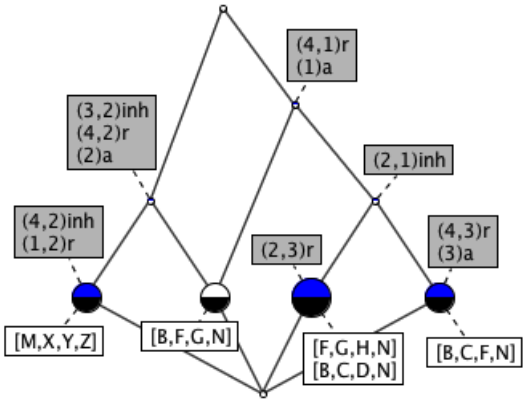

Figure 4.8: Lattice diagram before reorder of classes

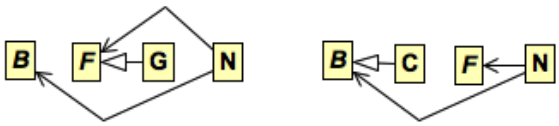

Figure 4.9: Sequences BFGN and BCFN before reorder

As solution, algorithm was devised to find sequences that have same attributes and reorder classes of one sequence according to other (Figure 3).

```
objSET ← all sequences of classes (formal objects)
concepts ← all concepts
for all i ∈ objSET.size do
        allPermutations ← getAllPossiblePermutationsOfClassesOfSeq(objSET[i])
        for all j  ∈ concepts.size do
                nextPerm ← allPermutations[i]
                newAttrs ← objSET[i].getAttributesCorrelatedWith(nextPerm)
                for all h  ∈ objSET.size do
                        if newAttrs==objSET[h].attrs && i!=h then
                                objSET[i].reorderClassesAccordingTo(nextPerm)
                                objSET[i].attrs ← newAttrs
                                duplicFound ← true
                                break out of inner loop
                if duplicFound == true then
                        break out of inner loop
        duplicFound ← false
call fillContextCrossTableWithValues to update cross table
```

Figure 4.10: Algorithm for reordering classes in sequence

After running code with the algorithm, sequence BCFN was reordered into FBCN as it was matched with attributes of BFGN (see Figure 4.12, that was drawn after the analyze of lattice diagram, Figure 4.11).

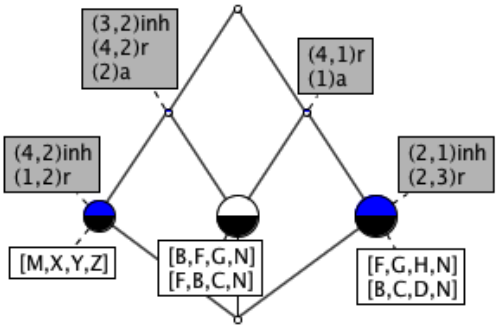

Figure 4.11: Lattice diagram after reorder of sequences representing same pattern

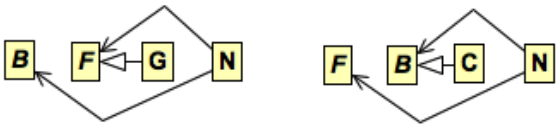

Figure 4.12: Sequences BFGN and BCFN after reorder

As a result, sequences now belong to single concept (compare Tables 4.2 and 4.3) and number of concepts have been decreased.

```
c0 = ( {[B,C,D,N], [B,C,F,N], [B,F,G,N], [F,G,H,N], [M,X,Y,Z]} , {} )
c1 = ( {[B,C,D,N], [B,C,F,N], [B,F,G,N], [F,G,H,N]} , {[(1)a, (4,1)r]} )
c2 = ( {[B,C,D,N], [B,C,F,N], [F,G,H,N]} , {[(1)a, (4,1)r, (2,1)inh]} )
c3 = ( {[B,F,G,N], [M,X,Y,Z]} , {[(2)a, (4,2)r, (3,2)inh]} )
c4 = ( {[B,C,D,N], [F,G,H,N]} , {[(1)a, (2,3)r, (4,1)r, (2,1)inh]} )
c5 = ( {[B,F,G,N]} , {[(1)a, (2)a, (4,1)r, (4,2)r, (3,2)inh]} )
c6 = ( {[B,C,F,N]} , {[(1)a, (3)a, (4,1)r, (4,3)r, (2,1)inh]} )
c7 = ( {[M,X,Y,Z]} , {[(2)a, (1,2)r, (4,2)r, (3,2)inh, (4,2)inh]} )
c8 = ( {} , {[(1)a, (2)a, (3)a, (1,2)r, (2,3)r, (4,1)r, (4,2)r, (4,3)r, (2,1)inh,
(3,2)inh, (4,2)inh]} )
```

Table 4.2: Concepts before reordering of classes

```
c0 = ( {[B,C,D,N], [F,B,C,N], [B,F,G,N], [F,G,H,N], [M,X,Y,Z]} , {} )
c1 = ( {[B,C,D,N], [F,B,C,N], [B,F,G,N], [F,G,H,N]} , {[(1)a, (4,1)r]} )
c2 = ( {[F,B,C,N], [B,F,G,N], [M,X,Y,Z]} , {[(2)a, (4,2)r, (3,2)inh]} )
c3 = ( {[B,C,D,N], [F,G,H,N]} , {[(1)a, (2,3)r, (4,1)r, (2,1)inh]} )
c4 = ( {[F,B,C,N], [B,F,G,N]} , {[(1)a, (2)a, (4,1)r, (4,2)r, (3,2)inh]} )
c5 = ( {[M,X,Y,Z]} , {[(2)a, (1,2)r, (4,2)r, (3,2)inh, (4,2)inh]} )
c6 = ( {} , {[(1)a, (2)a, (1,2)r, (2,3)r, (4,1)r, (4,2)r, (2,1)inh, (3,2)inh,
(4,2)inh]} )
```

Table 4.3: Concepts after reordering of classes

4.7 Finding Known Patterns

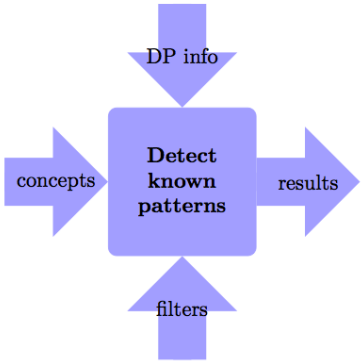

Figure 4.13: Filtering and detecting known patterns

Having found concepts (section 4.3) and characterized patterns (section 4.2.2), comparing of those two data sets is possible in order to present positive matches.

Sub system was built to handle such task, which include:

- query – consist of attributes of certain design pattern (Table 4.1) and of additional filtering criteria (Table 4.4);
- data – single concept on what the query would be applied;
- query system – processes queries and replies with concepts found for each query.

Filtering criterion	Description	Usage in single query
client included	include Client participant in design pattern attributes before matching	at least one of them must exist
client excluded	exclude Client participant from design pattern attributes before matching	
exact	concept should have exact match of attributes with the pattern	at least one, but also both can exist
overload	concept is allowed to have more attributes than pattern	
most general	among all found concepts existing on the same path (based to subconcept-superconcept relation), include only first one in results (the most general concept, with fewest attributes)	can exist only with overload; not mandatory

Table 4.4: Filtering criteria components explained

Different filtering would add flexibility and allow to detect variants (as mentioned in section 4.2.2). One filter might detect instance that other might miss.

Following algorithm explain how current query system detection works. Algorithm tries to find favorable permutation of classes in sequence (representing design pattern).

If new permutation exists, such that after reordering of elements (similarity to solution at section 4.6) match exists with intent of other concept (according to specified filtering), then candidate concept has been found with its enclosing instances.

```
concepts ← all concepts
objSET ← all sequences of classes (formal objects)
attrSET ← all possible attributes (formal attributes)
attr ← set of sorted objects representing attributes ("formal attributes"); every such
object holds also list of indexes to "formal object" (index to obj set) it has binary
relation with
dPs ← all design patterns to search for and their info

#getMatchesToEveryFiltering:
IN: concepts, attrSET
allResults ← {∅} (to hold all found results)
pc ← createQueryProcessor(attrSET, concepts, attr)
for all i ∈ dPs.size do
        for all j ∈ filters.size do
                queryToSend ← composeQuery(dPs[i], filters[j])
                queryResultSet ← pc.processQuery(queryToSend)
                if queryResultSet.size > 0 then
                        allResults ← allResults ∩ (queryResultSet, query)
OUT: allResults

#processQuery:
IN: query (consisting of dP and filter)
allPermutations ← getAllPossiblePermutationsOfClassesOfSeq(dP.getSeq)
for all i ∈ allPermutations.size do
        nextPerm ← allPermutations[i]
        newIntent ← dP.getAttributesCorrelatedWith(nextPerm)
        for all j ∈ concepts.size do
                if newIntent == concepts[j].intent OR
                  (newIntent matches with concepts[h].intent according to fitler) then
                        concepts[j].fillExtent(objSET) (extent was not set, now
                        provide full info)
                        matches ← matches ∪ concepts[j]
OUT: matches
```

Figure 4.14: Algorithm for querying and filtering the concepts

Finally, results are slightly formatted in a way to be understandable and pleasantly readable for the user of GUI (Figure 5.1).

5 Evaluation and Improvements

Current chapter describes how tool, that was built on the process, is evaluated. Evaluation is done in order to get overview on how tool fulfills set goals from following perspectives: how tool behaves and could be used (usability), how well detects (detection precision), the time process takes to complete (time performance). Also future improvements are found and suggested.

Methodology for evaluation: open source project often used by developers would be chosen; tool would be run on chosen project' source code; results would be compared with baseline information available on that project; detection precision, time performance, potential problems would be reported. While evaluating adjustments would be implemented to overcome found problems.

5.1 Choosing Project

In order to evaluate the tool, open source project needs to be chosen that could supplement with baseline knowledge: what patterns contain, how many instances are there representing each pattern and location of those instances. Additionally, when choosing project, the quality of software needs to be considered, to be able to rely on source code being written by experienced developers. Finding such ideal project containing all described posed to be difficult.

Alternatively, to find comparison and baseline information, following was considered. There are many works (i.e. Rasool and Mäder [8], Shi and Ollson [16]) which also deal with detection and evaluation of their tools. Even though documents they produce do not follow single common reporting method, which makes directly comparing results not possible ("no standard benchmarks are available that facilitate the comparison of pattern detectors" Pettersson et al. [18], p 1), still projects appearing in those works could be reused.

As a result of search, project to fulfill role of real world example was chosen to be AWT library (java.awt package, supplied by JDK1.7.015). Also, to get comparison information, other detection tools could be run on the same package.

5.2 Evaluation with AWT

Setup was adjusted to order 3 and only those patterns with at least 4 participants were searched (Appendix B, Table B.17).

5.2.1 Problems and Adjustments

Some reported problems and adjustments:

- First experiment run on AWT (no filtering was used) resulted with no matchings. Reason were: 1) parameters were too general; 2) AWT package does not contain such role as "Client" (being framework, "Clients" roles are mostly declared by implementing applications).

  After adding new filter criteria ("client included", "client excluded") and guiding characteristic "icClient", the results were produced. "IsClient" gives chance to remove role "Client" from characteristics.
- Too much false positives appeared for Builder, Memento, Template Method, because they have too general characteristics. They were removed from detection process (could be added later if mend them with more attributes).
- Filter "overload" include all concepts on the same path in subconcept-superconcept relation, regardless of the number of attributes those concepts have (Appendix C, Figure C.3). After adding new filter "most general" results became more compact and observable (Figure 5.1).

5.2.2 Time Performance

Detection process took 58 minutes (from moment of providing source code to getting results). To compare results with other tools: Niere et al. [19] (p 9) reports 22 minutes to analyze AWT for detection to complete, while others report seconds (Table 5.1). However, comparing does not give complete and accurate overview, as there are different factors (i.e. detection approaches involved, different patterns covered) which makes results often not comparable. Also tools were randomly chosen and there is small number of them.

Still the cause of long time was observed and analyzed – overview of time spent on different parts of the code is given in Appendix (Appendix C). Some parts of algorithm came out to be very time consuming. Additionally, when experimenting with order 4,

the time to finish raised too high (>2 days). For this reason, in current state, if to detect patterns that have 4 and more participants, and at the same time test on large project, makes tool impractical to use.

Tool used	**Time to finish**	**Approach**
Current tool	58 minutes	FCA based
PINOT [20]	6 seconds	other
Design Pattern Detection Tool [21]	51 seconds	other
WOP Client [22]	3 minutes 14 seconds	other

Table 5.1: Comparing execution time with other tools (AWT)

5.2.3 Accuracy of Detection

Currently tool was adjusted to detect following patterns: ADAo, CHOR, COMP, PROT, PROX. Validating all found 189 instances (Table 5.2) would be tedious and inefficient undertaking.

Pattern	ADAo	CHOR	COMP	PROT	PROX
Number of instances found	99	40	10	39	1

Table 5.2: Number of instances found for each pattern (AWT)

As there is no automated means to validate all the finds, author have taken small part of the results to concentrate on instead. Focus is on the results of Composite pattern (shown on Figure 5.1). Most of the found instances on Figure 5.1 represent not strict implementation of Composite design pattern:

- MenuBar, MenuComponent, MenuItem match nearly formal definition of Composite (1 superclass, 2 subclasses – one of the subclasses has collection type reference (Vector) to superclass and methods for adding and removing superclass type elements from that collection).
- AWTEvent, SentEvent, SequenceEvent seems to be the furthest from being Composite (does contain 1 superclass, 2 subclasses, but both classes reference superclass and lack adding and removing methods).
- The rest, involving Component and Container, are almost matches (1 superclass, 2 subclasses – one subclass has collection type reference to superclass and methods for adding and removing superclass type elements from collection).

Actually, all instances here could be viewed as one instance found containing many leaf nodes (Button, Canvas, etc).

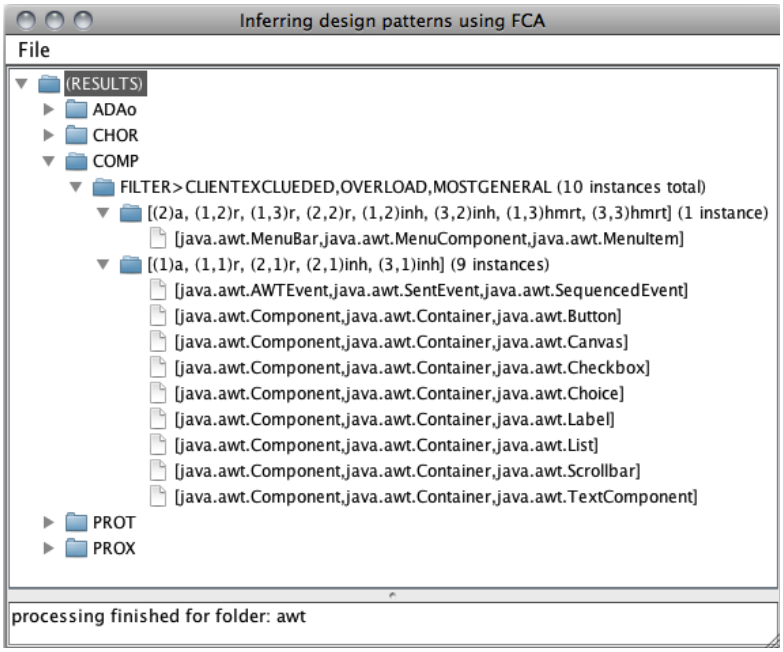

Figure 5.1: Composite pattern instances (AWT)

As appears, tool detects well Composite instances, without having too many characteristics defined (out of 10, only 1 candidate was questionable).

	Number of instances found	**Validated to be true positive**	**Comparing comment**
Current tool	10	9	–
PINOT [20]	0*	–	–
Design Pattern Detection Tool [21]	0*	–	–
DPJF [23]	0*	–	–
WOP Client [22]	2	2	did not report patterns detected by current tool; and vice versa, current tool did not find those 2, because they were variation with less attributes (could be found by adding "almost" (section 4.2) filter – allows to include upper neighbor concept with less attributes)

**did not find any instances (cause unknown: could be because of specifics in set up or executing, or follows too strictly formal definition of patterns)*

Table 5.3: Detecting composite patterns with different tools (AWT)

From other perspective, if to compare results of tool (Table 5.3) with the results of randomly chosen tools (run on same AWT), unexpected difference is found: current tool find instances where other miss. Could latter be because their approach follow too strictly definition of patterns, and for that reason do not see variations, which current approach sees? Author assumes this most likely to be the cause, but the actual cause for the difference was not looked deeper into.

5.3 Further Improvements

The tool developed during current work is not completed. As evaluation showed, incompletenesses were found that stress some possible improvements or corrections needed to be addressed.

5.3.1 Algorithms

Some tasks take long to complete, thus affected algorithms or part of them offer poor time performance (see Appendix C, lines in italic). Following table gives short overview of tasks and maps them with the algorithm involved that requires improving or replacement with alternatives.

Task	**Algorithm used**
Finding sequences	calculateNextLevel, Figure 4.3
Reordering classes in sequences	Figure 4.10
Calculating concepts	Figure C.1 and C.2 (Appendix C.1)

Table 5.4: Long time executing tasks and corresponding algorithms

Improvements to Current Algorithms

To improve performance concurrency functionality could be experimented with. Also as there are lot of collection structures involved for each of those algorithms, then usage of collections should be reviewed.

Alternatives to Current Concept Calculating Algorithm

From the perspective of time performance there are more efficient alternatives:

- Kuznetsov and Obiedkov [24] set up experiments aiming to find performance of algorithms: Godin, Close by One, Norris, Bordat. Criteria for choosing algorithm was proposed: to consider properties of input data, such as if context being large/small and sparse/dense/average. But there are other aspects to algorithm choosing discussed that affect performance and direct choosing of algorithm (i.e. the programming language used).
- Strok and Neznanov [25] analyzed performance of following algorithms: AddIntent, InClose, Norris, FCbO. They observed that size and density of dataset (context) are dictating which algorithm to use.

Analyze and experimentation is needed to find which previous algorithm might be more appropriate for current purpose.

5.3.2 User Interface

If to view GUI from the aspect of usability and how well second goal is covered, many drawbacks were found. GUI lacks controls and does not allow user to affect outcome. User interface could allow: specify patterns to detect, browsing to source location where instance are realized. Because the tool does not explain intuitively enough results found, there could be improvements in: describing findings, presenting progress (as there is no feedback of the duration of detection process), presenting additional statistics on results (such as sum of all instances found).

5.3.3 Characteristics and Filtering

As evaluation showed new filters could be applied to get more flexible detection. Also more characteristics could be tested, including those not implemented but defined by section 4.2 . What is more, within the scope of current work five patterns only were attempted to detect, rest should be added as well. Also lattice information such as upper and lower neighbors were not utilized. Latter information could be integrated into filtering process, for instance by adding "almost" filter.

6 Conclusion

In current thesis author attempted to produce process and tool for automatic design pattern inference – to find and present occurrences of Gang of Four design patterns existing in selected source code. Work was built upon existing idea and the approach of Tonella and Antoniol [13] and Buchli [14], but develops its own perspective and solution.

6.1 Summary of Work Done

Current work utilized Formal Concept Analysis. FCA has found practical use as data analyze method in different domains of expertise and offers automating process of grouping and relating elements of data set based on the attributes elements share. FCA terms related to current work are introduced to reader in second chapter of this work.

Design patterns are solutions to reoccurring design problems and are collected into catalogues – GoF book among first of them. Design patterns minimize effort of developers through reuse. Being knowledgeable of design patterns could offer benefits, such as in cases where there is minimal or no documentation available, and there is need to study code of different project. Design patterns reveal valuable information on the parts of the code. Many works have addressed detection of patterns, suggesting different approaches, including FCA. FCA overcomes obstacles that other approaches might not and could derive pattern candidates without pre-consulting pattern library. Brief introduction into design patterns and detection was presented by third chapter.

In the fourth chapter solution was provided with overview on the process to accomplish primary goal set at the beginning of work. Initially characteristics are chosen to be collected from the code; design patterns are re-classified, and for flexibility "guiding characteristics" are added. First of all, source code would be parsed to collect classes and agreed info. Secondly, context building algorithm would produce "formal context" from collected info. Some correction algorithm is implemented to correct context in the form of reordering classes to decrease number of possible concepts representing similar patterns. As a third step, algorithm based on the bottom-up approach would calculate concepts. Finally concepts and design pattern library that

contains re-classified patterns will go through filtering and all concepts representing GoF design patterns would be produced as a result with the instances they contain in extent. Gradually based on the whole process tool was implemented.

Eventually evaluation of resulting tool on AWT package was undertaken in order to pass some simple reality check that process achieves some of its intended purpose. Overview of how tool behaves, how well detects and its time performance was presented. There author reached conclusion that resulting tool detects some patterns successfully, but has limited capabilities and needs additional work with.

6.2 Goal Reaching and Conclusion

As a result of current work author have succeeded in developing process where FCA is used for inferring patterns from provided Java source code, as was set by primary goal. Tool was also implemented based on the process and could be described as being in prototype status – offering some overview of potential, but yet incomplete.

Regarding the second goal – to achieve in creating tool that could be used by novice when learning design patterns – the goal was not reached. To succeed in latter, additional effort is required in different areas.

Many improvements could be done in areas such as: for time performance, algorithms could be improved; for flexibility, more characteristics (also adding new "guiding characteristics") and new filtering criteria could be included; for usability, intuitive GUI and more controls, could be implemented. During evaluation author noticed that resulting tool finds pattern instances where other detection tools might not. In order to get clearer overview, evaluation needs to be extended to other projects beside AWT and other detection tools beside those used.

It is admitted that analyzing large software system is expensive task, but it is especially expensive when using FCA – demerit that was confirmed during the cause of evaluation. With increasing number of concepts the computational complexity raises considerably. On the other hand, the interesting challenge appeared – to improve algorithms involved.

Obviously, to get better overview of the huge source code it is not enough to have tool that presents patterns. There could be rainbow of tools, each presenting their perspective of the code, their detail of analyze, guided by the user interest, experience or

nature of the software. Nevertheless the most beneficial for those approaching new project with yet unknown to them source code could be tools presenting short and to the point overview, which would not drown learner into sea of details on the first dips. Resulting tool could be one of them. Though currently not fulfilling such purpose, the author has developed design pattern detection process and produced a prototype tool based on the process. With more effort complete tool could be developed from current work. Alternatively, information contained here could be reused or assist in other endeavors related to Formal Concept Analysis or design pattern detection.

Bibliography

[1] Design Patterns: Elements of Reusable Object-Oriented Software. /E. Gamma, R. Helm, R. Johnson, J. Vlissides. Massachusetts : Addison-Wesley, 1995.

[2] Wille, R. Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies. — Formal Concept Analysis : Foundations and Applications. Berlin, Germany : Springler-Verlag, 2005, 1–6.

[3] Ganter, B., Wille, R. Applied Lattice Theory: Formal Concept Analysis. [Online] http://www.math.tu-dresden.de/~ganter/psfiles/concept.ps (13.04.2014).

[4] Yevtushenko, S. Computing and Visualizing Concept Lattices : Doctoral Thesis. Technische Universität Darmstadt, Darmstadt, Germany, 2004. [Online] http://tuprints.ulb.tu-darmstadt.de/488/1/diss-yevtushenko.pdf (16.04.2014).

[5] Yevtushenko, S. A. System of data analysis "Concept Explorer". — In Proc. of the 7th national conference on Artificial Intelligence KII-2000 : 2000, Russia, 127–134.

[6] Eklund, P., Wray, T., Goodall, P., Lawson, A., Bunt, B., Christidis, L., Daniels, V., Van Olffen, M. Designing the Digital Ecosystem of the Virtual Museum of the Pacific. — In Proc. of the 3rd IEEE International Conference on Digital Ecosystems and Technologies : 1–3 June 2009, Istanbul, 805–811.

[7] Krämer, C., Prechelt, L. Design recovery by automated search for structural design patterns in object-oriented software. — In Proc. of the 3rd Working Conference on Reverse Engineering (WCRE'96) : 11–10 November 1996, Monterey, USA, 208–215.

[8] Rasool, G., Mäder, P. Flexible Design Pattern Detection Based on Feature Types. — In Proc. of the 26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011) : 6–10 November 2011, Lawrence, USA, 244–252.

[9] Dong, J., Zhao, Y., Peng, T. A review of design pattern mining techniques. — International Journal of Software Engineering and Knowledge Engineering, 2009, Vol 19, No 6, 823–855. [Online] WorldScientific (13.04.2014).

[10] Lee, E., Lee, H., Youn, H. A Design Pattern Detection Technique that Aids Reverse Engineering. — International Journal of Security and its Applications, 2008, Vol 2, No 1, 1–11. [Online] Science & Engineering Research Support soCiety (13.04.2014).

[11] Heuzeroth, D., Holl, T., Högström, G., Löwe, W. Automatic Design Pattern Detection. — In Proc. of the 11th International Workshop on Program Comprehension (IWPC '03) : 10–11 May 2003, Portland, USA, 94–103.

[12] Detten, M., Meyer, M., Travkin, D. Reverse Engineering with the Reclipse Tool Suite. — In Proc. of the 32nd Working International Conference on Software Engineering (ICSE '10) : 2–8 May 2010, Cape Town, 299–300.

[13] Tonella, P., Antoniol, G. Object Oriented Design Pattern Inference. — In Proc. of the International Conference on Software Maintenance (ICSM '99) : 1999,

Oxford, 230–238.

[14] Buchli, F. Detecting Software Patterns using Formal Concept Analysis : Master Thesis. University of Bern, University of Bern, 2003. [Online] http://scg.unibe.ch/archive/masters/Buch03a.pdf (13.04.2014).

[15] Rasool, G., Maeder, P., Philippow, I. Evaluation of design pattern recovery tools. — Procedia Computer Science, 2011, Vol 3, 813–819. [Online] ScienceDirect (13.04.2014).

[16] Shi, N., Ollson, R. Reverse Engineering of Design Patterns from Java Source Code. — In Proc. of the 21st IEEE/ACM International Conference on Automated Software Engineering : 18–22 September 2006, Tokyo, 123–134.

[17] Siff, M., Reps, T. Identifying Modules Via Concept Analysis. — In Proc. of the International Conference on Software Maintenance : 1–3 October 1997, Bari, Italy, 170–179.

[18] Pettersson, N., Löwe, W., Nivre, J. Evaluation of Accuracy in Design Pattern Occurrence Detection. — IEEE Transactions on Software Engineering, 2010, Vol 36, Issue 4, 575–590. [Online] IEEEExplore (13.04.2014).

[19] Niere, J., Schafer, W., Wadsack, J. P., Wendehals, L., Welsh, J. Towards Pattern-Based Design Recovery. — In Proc. of the 24th International Conference on Software Engineering : 25 May 2002, Orlando, USA, 338–348.

[20] Shi, N., Ollson, R. Pinot. [Computer software] http://www.cs.ucdavis.edu/~shini/research/pinot/pinot-src.tar.gz (13.04.2014).

[21] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., T. Halkidis, S. Design Pattern Detection Tool (4.5). [Computer software] ttp://java.uom.gr/~nikos/files/pattern-detection/pattern4.jar (13.04.2014).

[22] Dietrich, J., Elgar, C. WOP Client Eclipse Plugin (1.4.3). [Computer software] http://webofpatterns.svn.sourceforge.net/svnroot/webofpatterns/update/ (14.04.2014).

[23] Binun, A., Kniesel, G, DPD Community. DPJF Eclipse Plugin (1.4.3). [Computer software] http://sewiki.iai.uni-bonn.de/public-downloads/update-site-dpjf/ (13.04.2014).

[24] Kuznetsov, S., Obiedkov, S. Comparing Performance of Algorithms for Generating Concept Lattices. — Journal of Experimental and Theoretical Artificial Intelligence, 2002, Vol 14, Issue 2-3, 189–216. [Online] Taylor & Francis Online (13.04.2014).

[25] Strok, F., Neznanov, A. Comparing and Analyzing the Computational Complexity of FCA algorithms. — In Proc. of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT '10) : 11–13 October 2010, Bela Bela, South Africa, 417–420.

Lühikokkuvõte

Magistritöö on koostanud protsessi leidmaks etteantud lähtekoodist Gang of Four (GoF) disaini mustrid. Sellega täideti esimene tööle püsitatud eesmärk – töötada välja protsess, mis võimaldaks etteantud Java koodist formaalse kontseptianalüüsi (FCA) abil avastada GoF disaini mustreid. Protsessi koostamisel on toetutud Tonella ja Antoniol ("Object Oriented Design Pattern Inference", 1999) ja Buchli ("Detecting Software Patterns using Formal Concept Analysis", 2003) tööde viidatud ideedele.

Koostatud protsess koosneb neljast osast. Määratleti karakteristikud, mille abil kirjeldati nii GoF mustrid kui lähtekoodis korjatavad tarkvara klassid. Protsessi esimeses osas korjatakse lähtekoodist info klasside ja karakteristikute kohta, kasutatades Abstract Syntax Tree abi. Teises osas töödeldakse kogutu FCA tarbeks tabelisse, kus ridadeks klasside hulgad ja veergudeks karakteristikud. Kolmandas osas kalkuleeritakse tabeli põhjal FCA algoritmiga kontseptid ehk mustrite kandidaadid. Viimases osas sõelutakse välja kandidaadid mis tegelikult esindavad GoF mustreid – iga kontsepti atribuute võrreldakse mustrite atribuutidega ning tulemused läbivad filtri (lubamaks mustrite variatsioone). Kui atribuudid klapivad tegeliku mustriga ja filter rakendub, tagastab protsess GoF mustri ja viimast koodis esindavad klassid.

Samuti realiseeriti protsessi põhjal prototüüp rakendus. Saamaks ülevaate, kui hästi lahendus täidab talle püstiatud eesmärke, hinnati rakendust ning leiti nii puuduseid kui eelise. Puuduste poolelt avastati: algoritmid ei võimalda koguka lähtekoodi puhul leida teatud mustreid; graafiline kasutajaliides ei esita tulemeid intuitiivselt, ei võimalda kasutajal muudatusi teha. Eelisena leiti, et rakendus avastab Composite mustrite variatsioone seal kus teised sarnase otstarbega rakendused võivad mitte avastada.

Tööle püstitatud teist eesmärki – realiseerida vahend aitamaks disaini mustreid õppijatel valitud lähtekoodist mustrite olemasolu tuvastada ja mustreid realiseerivate klasside kohta ülevaate saada – ei täidetud. Saavutamaks teist eesmärk tuleks tulevikus parandada kasutajaliidest ja algoritme ning eksperimenteerida ja hinnata tulemusi palju enamate seadete korral kui antud töös seda võimalik oli teostada.

Appendix A – Abbreviations Used

FCA	Formal Concept Analysis
GoF	Gang of Four, referring to the authors of book Gamma et al. [1]
AF	Abstract Factory
BUI	Builder
FM	Factory Method
PROT	Prototype
ADAo	Object Adapter
BRIDG	Bridge
COMP	Composite
DECOR	Decorator
FLYW	Flyweight
PROX	Proxy
CHOR	Chain of Responsibility
INTERP	Interpreter
ITER	Iterator
MEDI	Mediator
MEM	Memento
OBS	Observer
STAT	State
STRAT	Strategy
TM	Template Method
VISIT	Visitor
DP	design pattern
AST	Abstract Syntax Tree
JDT	Java Development Framework
GUI	Graphical User Interface
AWT	package java.awt included in Java Development Kit
false positive	instance found, but actually not representing pattern
true positive	instance found, and does represent pattern

Appendix B – Characteristics

B.1 Mapping Characteristics between Java Construct and AST for Parsing

This section describes possible occurrences of characteristics (that were decided to collect) in Java code and for collecting them, AST representative counterpart (represented by JDT).

Inheritance Relations

Following tables will summarize locations where **isInheriting, isAbstract, isInterface** are collected.

Class level		
Specific name in Java	**Java example**	**JDT counterpart**
Sub class	`class A extends B{`	TypeDeclaration, .getSuperclassType(), SimpleType
Sub class	`class B implements C{`	TypeDeclaration, .superInterfaceTypes(), SimpleType

Table B.1: Occurrences of isInheriting characteristic in Class

Class level			
Characteristic	**Specific name in Java**	**Java example**	**JDT counterpart**
isAbstract(1)	Abstract class or superclass	`abstract class B{`	TypeDeclaration, Modifier, isAbstract()
isInterface(1)	Interface or super interface	`interface B{`	TypeDeclaration, isInterface()

Table B.2: Occurrences of isAbstract and isInterface characteristic in Class

Reference Relations

Following tables will summarize locations where **isReferringTo** is collected.

Simple reference variable

Class level		
Specific name in Java	**Java example**	**JDT counterpart**
Instance variable	`B varA;`	TypeDeclaration, FieldDeclaration, SimpleType
Class variable	`static B varA;`	TypeDeclaration, FieldDeclaration, SimpleType

Table B.3: Occurrences of isReferringTo in Class (simple reference variable)

Method level		
Specific name	**Java example**	**JDT counterpart**
Local variable	`public getH() { B var1;`	MethodDeclaration, VariableDeclarationStatement, SimpleType
Method parameter	`public getH(B var1) {`	MethodDeclaration, SingleVariableDeclaration, SimpleType

Table B.4: Occurrences of isReferringTo in Method (simple reference variable)

Block level		
Specific name in Java	**Java example**	**JDT counterpart**
Local variable (initialization block)	`{ B var1; }`	Initializer, Block, VariableDeclarationStatement, SimpleType

Table B.5: Occurrences of isReferringTo in Block (simple reference variable) (1)

Block level (belongs to Method or Initialization block or another Block)		
Specific name in Java	**Java example**	**JDT counterpart**

Local variable (block)	`{`
	`  B var1;`
	`}`
		Block, VariableDeclarationStatement

Local variable (inside basic for statement, initialization)	`for(B var1=`	Block, ForStatement, VariableDeclarationExpression, SimpleType

(inside for loop)	`for(int i = 0;`
	`i <len; i++) {`
	`  B var1;`
		Block, ForStatement, Block, VariableDeclarationStatement, SimpleType

(inside while loop)	`while(test){` `  B var1;`	Block, WhileStatement, Block, VariableDeclarationStatement, SimpleType
(inside do-while loop)	`do{` `  B var1;`	Block, DoStatement, Block, VariableDeclarationStatement, SimpleType
(inside if branch)	`if(true){` `  B var1;`	IfStatement, getThenStatement(), Block, VariableDeclarationStatement, SimpleType
(inside else if branch)	`}else if(test){` `  B var1;`	IfStatement, getElseStatement(), IfStatement, getThenStatement(), Block, VariableDeclarationStatement, SimpleType
(inside else if branch)	`}else{` `  B var1;`	..., getElseStatement(), Block, VariableDeclarationStatement, SimpleType

(inside switch statement)	`switch(i){`
	`case 1:`
	`  B var1;`
		SwitchStatement, Block, VariableDeclarationStatement, SimpleType

(inside try statement)	`try{` `  B var1;`	TryStatement, Block, VariableDeclarationStatement, SimpleType
(inside try statement)	`catch(Exception e){` `  B var1;`	TryStatement, catchClauses(), CatchClause, Block, VariableDeclarationStatement, SimpleType
(inside try statement)	`}finally{` `  B var1;`	TryStatement, getFinally(), Block, VariableDeclarationStatement, SimpleType

Table B.6: Occurrences of isReferringTo in Block (simple reference variable) (2)

- ***Simple reference variable through generics.***

Class level (belongs to Class or Method or Block or Initialization Block)		
Specific name in Java	**Java example**	**JDT counterpart**
Wildcard generics	`List<? extends B> var1;`	FieldDeclaration, ParameterizedType, typeArguments(), WildcardType, getBound(), SimpleType
... rest of the list (generics could be used in same occurrences specified by tables B.3–B.6) ...		

Table B.7: Occurrences of isReferringTo in Class (simple reference variable, generics)

An array of references or references via collection

- ***Collection through arrays.*** Array could be declared in the same locations as variables depicted by previous tables. In JDT, getting type of objects that array is holding, is through replacing SimpleType with ArrayType.

Class level (belongs to Class or Method or Block or Initialization Block)		
Specific name in Java	**Java example**	**JDT counterpart**
As array of object references	`B[] var1;`	TypeDeclaration, FieldDeclaration, ArrayType, SimpleType
... rest similar to occurrences specified by tables B.3–B.7 ...		

Table B.8: Occurrences of isReferringTo in Class (array reference variable)

- ***Collection through "varargs".*** Varargs are actually representing arrays, so that "R... varA" is exactly same as "R[] varA".

Method level		
Specific name in Java	**Java example**	**JDT counterpart**
As array of object references	`public void rel(B... n){`	MethodDeclaration, SingleVariableDeclaration (isVarargs()), SimpleType

TableB.9: Occurrences of isReferringTo in Class (array reference variable, varargs)

- ***Collection through collections provided by java (with generics).*** Collections have similar nature as arrays, to hold references to list of objects. Collection class represents here all those classes that inherit from java.util.Collection or java.util.Map interface.

Class level (belongs to Class or Method or Block or Initialization Block)		
Specific name in Java	**Java example**	**JDT counterpart**
Instance variable	`List<B> var1;`	TypeDeclaration, FieldDeclaration, ParameterizedType, typeArguments(), SimpleType
... rest similar to occurrences specified by tables B.3–B.7 ...		

Table B.10: Occurrences of isReferringTo in Class (simple reference variable, collections)

- ***Collections with no generics.*** References with no generics would not be collected and they are left out of current work scope.

Following table will show where **hasMethodWithReturnTypeRefTo** is collected.

Method level (belongs to Method)		
Specific name in Java	**Java example**	**JDT counterpart**
Method return type	`public B getX() {`	TypeDeclaration, MethodDeclaration, getReturnType2(), SimpleType

Table B.11: Occurrences of hasMethodWithReturnTypeRefTo in Method (return type)

B.2 Characteristics To Collect

Table represents characteristics planned to collect from each Java file, those actually collected are in italic.

Static aspects		**Behavioral aspects**	
Name of characteristic to collect	**Simplified abbreviation used**	**Name of characteristic to collect**	**Simplified abbreviation**
isInheriting(1, 2)	inh(1,2)	creates(1, 2)	cr(1, 2)
isAbstract(1)	abs(1)	createsIn(1, 2)<M>	crIn(1, 2)<M>
isInterface(1)	i(1)	calls(1, 2)<M>	calls(1, 2)<M>
isReferringTo(1, 2)	r(1, 2)	calls(1, 2)<M1,M2>	calls(1, 2)<M1,M2>
hasMethodWithReturnTypeRefTo(1, 2)	hmrt(1, 2)	hasMethodActuallyReturns(1, 2)	hmar(1, 2)
owns(1)<M>	own(1)<M>		

Table B.12: Summary of characteristics to be collected from source code

Following table represents guiding characteristics used to additionally describe design patterns (only isClient was used in current work).

Name of characteristic	**Simplified abbreviation used**
isClient(1)	client(1)
canBeConcrete(1)	
canNotBePureAbstract(1)	
refCanBeCollection(1)	

Table B.13: Summary of guiding characteristics to be used for filtering

B.3 GoF Design Patterns Characterized

In this section, according to minimum number of participants (Table B.14) each pattern would be described (Table B.15 – B.21) with agreed characteristics. Minimum number of participants is taken from Structure part of each design pattern description ([1]).

Minimum number of participants	**Name of the pattern**
10	Abstract Factory
8	Visitor
6	Bridge
5	Decorator, Flyweight, Command, Interpreter, Iterator, Mediator, Strategy
4	Builder, Factory Method, Prototype, Adapter (object), Composite, Proxy, Chain of Responsibility, Observer, State
3	Memento
2	Template Method
1	Singleton
Not able to specify	Facade

Table B.14: Patterns organized by their minimum number of participants

	(1)a	(2,1)inh
TM	X	X

Table B.15: Characteristics for patterns (at least 2 participants)

	(3,2)r	(1,2)hmrt
MEM	X	X

Table B.16: Characteristics for patterns (at least 3 participants)

	(1)a	(2)a	(3)a	(2,1)inh	(3,2)inh	(4,2)inh	(4,3)inh	(1,2)r	(1,3)r	(2,2)r	(3,4)r	(4,2)r	(4,3)r	(2,2)hmrt	(3,1)hmrt	(1)client
BUI	.	X	.	.	X	.	.	X	.	.	.	.	.	.	.	.
FM	X	.	X	X	.	.	X	.	.	.	.	.	.	.	X	.
PROT	.	X	.	.	X	X	.	X	.	.	.	.	.	X	.	X
ADAo	.	X	.	.	X	.	.	X	.	.	X	.	.	.	.	X
COMP	.	X	.	.	X	X	.	X	.	.	.	X	.	.	.	X
PROX	.	X	.	.	X	X	.	X	.	.	.	.	X	.	.	X
CHOR	.	X	.	.	X	X	.	X	.	X	.	.	.	.	.	X
OBS	X	.	X	X	.	.	X	.	X	.	.	X	.	.	.	.
STAT	.	X	.	.	X	X	.	X	.	.	.	.	.	.	.	.

Table B.17: Characteristics for patterns (at least 4 participants)

	(1)a	(2)a	(3)a	(4)a	(2,1)inh	(3,1)inh	(3,2)inh	(3,4)inh	(4,2)inh	(4,3)inh	(5,2)inh	(5,3)inh	(5,4)inh	(1,2)r	(1,3)r	(1,4)r	(1,5)r	(2,3)r	(2,4)r	(2,5)r	(3,1)r	(3,2)r	(5,3)r	(5,4)r	(2,4)hmrt	(1)client
STRAT	.	X	.	.	.	.	X	.	X	.	X	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.
MEDI	X	.	X	.	X	.	.	.	.	X	.	X	.	.	.	.	.	.	X	X	X	.	.	.	.	.
ITER	.	X	.	X	.	.	X	.	.	.	.	.	X	X	.	X	.	.	.	.	.	.	X	.	X	X
INTERP	.	.	X	.	.	.	.	.	.	X	.	X	.	X	X	.	.	.	.	.	.	.	X	.	.	X
COMM	.	.	.	X	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	X	.	X
FLYW	.	.	X	.	.	.	.	.	.	X	.	X	.	X	.	X	X	X	.	.	.	.	.	.	.	X
DECOR	X	.	X	.	X	X	.	.	.	X	.	X	.	.	.	.	.	.	.	.	X	.	.	.	.	.

Table B.18: Characteristics for patterns (at least 5 participants)

	(2)a	(4)a	(3,2)inh	(5,4)inh	(6,4)inh	(1,2)r	(2,4)r	(1)client
BRIDG	X	X	X	X	X	X	X	X

Table B.19: Characteristics for patterns (at least 6 participants)

	(2)a	(6)a	(3,2)inh	(4,2)inh	(7,6)inh	(8,6)inh	(1,2)r	(1,5)r	(5,6)r	(1)client
VISIT	X	X	X	X	X	X	X	X	X	X

Table B.20: Characteristics for patterns (at least 8 participants)

	(2)a	(5)a	(8)a	(3,2)inh	(4,2)inh	(6,5)inh	(7,5)inh	(9,8)inh	(10,8)inh	(1,2)r	(1,5)r	(1,8)r	(2,5)hmrt	(2,8)hmrt	(1)client
AF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table B.21: Characteristics for patterns (at least 10 participants)

Appendix C – Detection Tool

C.1 Algorithm for Calculating Concepts

```
obj ← set of sorted objects representing sequences ("formal objects"); every such
object holds also list of indexes to "formal attributes" (index to attr set) it has binary
relation with
attr ←  set of sorted objects representing attributes ("formal attributes"); every such
object holds also list of indexes to "formal object" (index to obj set) it has binary
relation with

#calculateConcepts:
IN: obj, attr
concepts ← {∅} (all concepts to be found)
maxAttrCount = attr.size
#bottom concept
O ← {∅}
for all i ∈ obj.size do
        if obj[i].getIntent().size == maxAttrCount then
                R ← O ∪ i
conc.extent ← O
conc.intent ← attr

#atomic concepts
for all i ∈ obj.size do
        conc ← {∅}
        conc.intent ← obj[i].getTrueAttrs()
        concepts ← concepts ∪ conc

#first work list
WL ← {∅}
for all i ∈ concepts.size do
        T1 ← {∅}
        T1 ← concepts[i].getIntent()
        for all j=i+1 ∈ concepts.size do
                conc ← {∅}
                T2 ← {∅}
                T2 ← concepts[j].getIntent()
                if  !T1.containsAll(T2) && !T2.containsAll(T1) then
                        conc.intent ← T1 ∩ T2
                        if !concepts.contains(conc) then
                                concepts ← concepts ∪ conc
                                WL ← WL ∪ conc
```

Figure C.1: Algorithm for calculating concepts (part 1)

```
#processing work list
for all i ∈ WL.size do
        T1 ← {∅}
        T1 ← WL[i].getIntent()
        for all j ∈ concepts.size do
                conc ← {∅}
                T2 ← {∅}
                T2 ← concepts[j].getIntent()
                if !T1.containsAll(T2) && !T2.containsAll(T1) then
                        conc.intent ← T1 ∩ T2
                        if !concepts.contains(conc) then
                                concepts ← concepts ∪ conc
                                WL ← WL ∪ conc

#top concept
conc ← {∅}
conc.intent ← {∅}
if !concepts.contains(conc) then
        conc.intent ← maxAttrCount
        for all j ∈ obj.size do
                T2 ← obj[i].getTrueAttrs()
                conc.intent ←  conc.intent ∩ T2
                if conc.intent.size==0 then
                        break inner loop;
        concepts ← concepts ∪ conc
OUT: concepts
```

Figure C.2: Algorithm for calculating concepts (part 2)

C.2 Performance and Statistics with AWT

Following tables were created to get overview of performance and statistics of the tool. They were produced for java.awt and with following setup: order 3, find patterns with minimum 4 participants (Table B.17), excluding Builder, Memento, Template Method. Most time consuming tasks are given in *italic*.

Comment				**Seconds**
All processing took				3431 (~58 m)

Table C.1: Summary of statistics gathered (AWT)

Comment				Seconds/Count
Parsing files took				21.6
	Num of files parsed			123
	Num of classes collected			214

Table C.2: Performance and statistics gathered for file parsing (AWT)

Comment				Seconds/Count
Calculating context took				*1417.1*
	Calculating context objects took			126.5
		Finding sequences	order 2 took	18.2
			Num of sequences found	565
		Finding sequences	*order 3 took*	*108.1*
			Num of sequences found	7792
	Finding all possible attributes took			0.001
		Num of attributes		30
	Finding matching crosses took			0.4
	Removing unused attributes took			0.03
		Num of attributes now		30
	Reordering classes in sequences took			*1289.7*
		Num of sequences reordered		5221

Table C.3: Performance and statistics gathered for context calculating (AWT)

Comment				Seconds/Count
Calculating concepts took	*Running bottom up algorithm took*			*1988.9*
		Finding bottom concept took		0.002
		Finding atomic concepts took		0.01
			Num of atomic concepts	2191
		Calculating first work list took		7.9
		Processing work list took		*1980.8*
			Num of work list elements checked	29686
		Finding top concept took		1.72E-4
		Num of concepts found		31877

Table C.4: Performance and statistics gathered for concepts calculating (AWT)

Comment				Seconds
Detecting and filtering took				24.1

Table C.5: Performance and statistics gathered for detection and filtering (AWT)

C.3 Instances Reoccurring with Overload Filter (AWT)

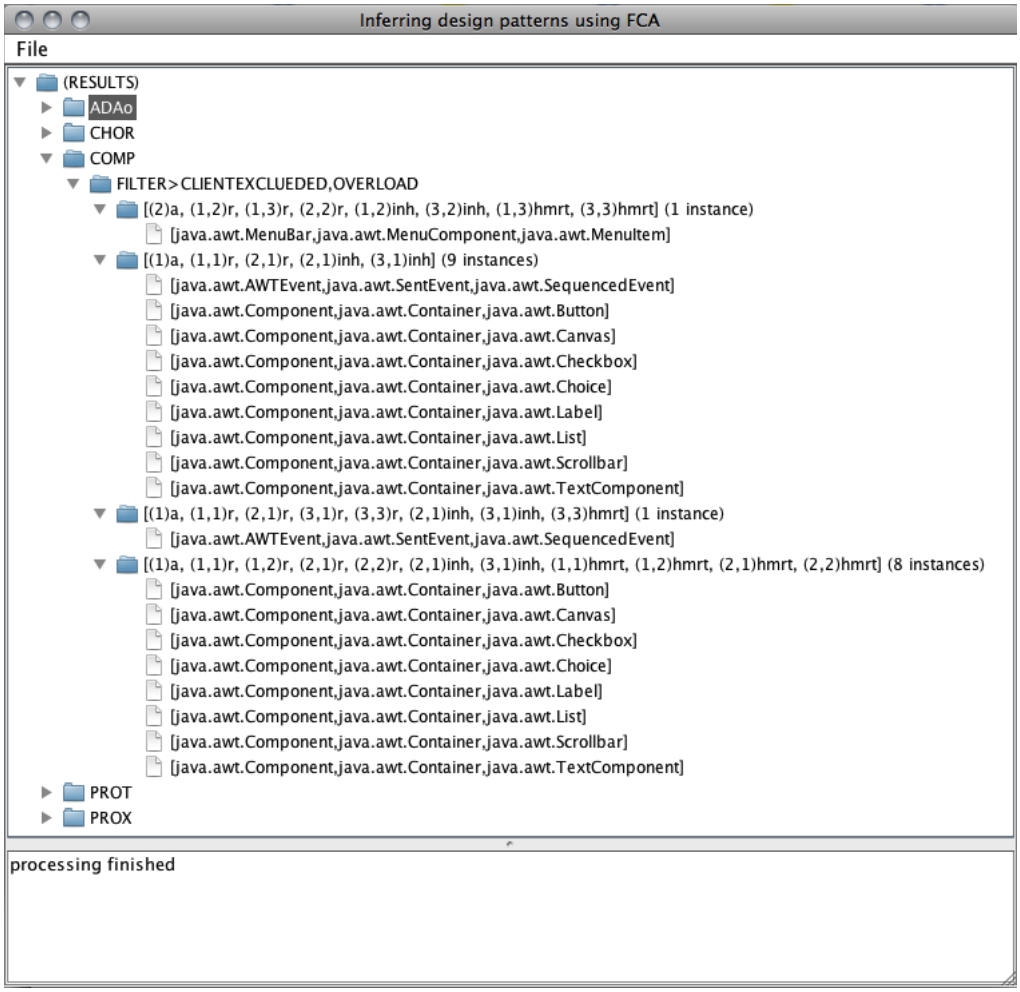

Figure C.3: Duplicates resulted for overload filter, Composite pattern (AWT)