Evaluating C++ Design Pattern Miner Tools

Lajos Fülöp, Tamás Gyovai and Rudolf Ferenc
University of Szeged, Department of Software Engineering
evosoft Hungary Ltd.
{flajos|ferenc}@inf.u-szeged.hu
tamas.gyovai@evosoft.com

Abstract

Many articles and tools have been proposed over the years for mining design patterns from source code. These tools differ in several aspects, thus their fair comparison is hard. Besides the basic methodology, the main differences are that the tools operate on different representations of the subject system and that the pattern definitions differ as well.

In this paper we first provide a common measurement platform for three well-known pattern mining systems, Columbus, Maisa and CrocoPat. Then we compare these tools on four C++ open-source systems: DC++, WinMerge, Jikes and Mozilla. Columbus can discover patterns from the C++ source code itself, while Maisa and CrocoPat require the representation of a software system in a special textual format, so we extended Columbus to provide the common input for the two other tools.

We compared these tools in terms of speed, memory consumption and the differences between the hits. While the first two aspects showed comparable results, the recognition capabilities were quite diverse. This is probably due to the significant difference in how the patterns to be recognized and formalized by the tools. Therefore we conclude that a more precise and formal description of design patterns would be desirable.

Keywords

Design pattern mining, Tool evaluation, Columbus, Maisa, CrocoPat, Program understanding

1 Introduction

Design patterns are well-known structures in the software development community. They describe good and well-tried solutions for common recurring problems. Gamma et. al. [12] collected several object oriented design patterns, and they gave an informal definition for them. It is very important that these definitions were informal, because this way a pattern can be used in a wider context. Due to the imprecise pattern definitions the implemented structures based on them could vary in different contexts. Design patterns also differ in the aspect that they are used intentionally or only in a casual way. A programmer can apply a design pattern, without actually knowing about it.

Design patterns are employed in many areas of software development. Originally, their main aim was to develop better software systems by using good solutions. Another good property of design patterns is, that their documentation in a software system can simplify maintenance and program understanding. This is especially true for large software systems. Unfortunately, the developers usually do not provide this documentation, so there is a big need to discover design patterns from source code. Therefore, in the past years various design pattern miner tools have been developed. These tools differ in several aspects.

One of the important aspects is the programming language. There are tools for discovering patterns from Java source code like Ptidej [13, 21] and Fujaba [11], and tools also exist for mining patterns from C++ code like Columbus [2, 9]. Certain tools like CrocoPat [3] and Maisa [20] work based on special own textual format which describe facts about the source code needed to find design patterns. These tools are general, because they can work on any programming language, only the appropriate textual input has to be prepared from the source code.

Another aspect is the method used to discover design patterns, which can be quite diverse. Columbus uses graph matching, while Maisa solves constraint satisfaction problems (CSP). CrocoPat has a new method to find structures in large graphs, it makes an effective representation of relations in graphs. Other special and interesting methods also exist, like pattern inference, which was presented by Tonella and Antoniol [22].

Our aim in this article was to compare three design pattern miner tools: Columbus, Maisa and CrocoPat. We chose these tools, because it was possible to prepare a common

input for them with our front end, Columbus. Our previous work enabled us to provide the input for Maisa [10], while in case of CrocoPat we created a new plug-in for Columbus which is able to prepare the appropriate input. Finally, the tools have been compared in three aspects: differences between the hits, speed and memory requirements. We think that these are the most important aspects in a design pattern miner tool. We did not analyze if a found design pattern hit is true or false, we examined these tools only with the consideration of structural hits and differences in this aspect.

We will proceed as follows. In Section 2, we will discuss some works similar to ours. In Section 3 we will introduce the design pattern miner tools, which were compared. Section 4 describes our comparison approach, and our results are presented in Section 5. Finally, in Section 6 we will present some conclusions and outline directions for future work.

2 Related Work

In this section we show some similar works to ours, and we also present new and interesting methods in the area of design pattern mining.

In our previous work we have presented a method to differentiate true and false hits [8]. We employed machine learning methods to filter out false design pattern hits. First, we ran our design pattern miner tool that discovers patterns based on structural descriptions. Afterwards, we classified these hits as being true or false, and finally we calculated predictive information for the hits. We trained a decision tree based on classified values and on the predictive information, from which we were able to mine true design pattern hits more accurately.

Design pattern detection was also accomplished by the integration of two existing tools – Columbus [9] and Maisa [20] – in our previous work [10]. This method combined the extraction capabilities of the Columbus reverse engineering system with the pattern mining ability of Maisa. First, the C++ code was analyzed by Columbus. Then the facts collected were exported to a clause-based design notation understandable for Maisa. Afterwards, this file was analyzed by Maisa, and instances were searched that matched the previously given design pattern descriptions. Maisa approached the recognition problem as a constraint satisfaction problem. We will get back to this method in Section 3.1.

Beyer et. al. [3, 4] have developed a system that is able to work with large graphs effectively. The effectiveness of the system is based on binary decision diagrams which represent the relations compactly. They have developed the relation manipulation language (RML) for manipulating n-ary relations and a tool implementation (CrocoPat) that is an interpreter for the language. The RML language is very expressive thus it is able to describe design patterns, design metrics or other structures. In section 3.2 we will present this tool in detail.

Arcelli et. al. [1] proposed three categories for design pattern mining tools, considering the information which was used during the detection process. These categories are the “entire” representation of design patterns, the minimal set of key structures that a design pattern consists of, and the sub-components of design patterns. They have dealt with the last category, and two tools concerning this, FUJABA and SPQR were introduced and compared. The base of the comparison was how a tool decomposes a design pattern into smaller pieces. The conclusion was, that the decomposition methods of the two examined systems are very similar, and finally they argued the benefits of sub-patterns.

Guéhéneuc et. al. [14] introduced a comparative framework for design recovery tools. The purpose of the authors’ framework was not to rank the tools but to compare them with qualitative aspects. This framework contained eight aspects: Context, Intent, Users, Input, Technique, Output, Implementation and Tool. These aspects were sorted into 53 criteria which were demonstrated on two systems, Ptidej and LiCoR. The major need for this framework is that, although there are a lot of design recovery tools, the comparison between them is very hard due to the fact that they have very different characteristics in terms of representation, output format and implementation techniques. This framework provides an opportunity for comparing not only similar systems, but also systems, which are different. We note that this comparison differs from ours because we compare systems in a practical way. Namely, we want to show and compare how many patterns a tool can find, and how much time and memory it needs for searching, while their comparison is rather theoretical, it does not compare the discovering effectiveness of tools.

Kaczor et. al. [16] proposed a bit-vector algorithm for design pattern identification. The algorithm initialization step converts the design pattern motif and the analyzed program model into strings. To model the design patterns and the analyzed program, six possible relations can be used between elements: association, aggregation, composition, instantiation, inheritance and dummy. The authors gave an efficient Iterative Bit-vector Algorithm to match the string representation of the design patterns and the analyzed program. They compared their implementation with explanation-based constraint programming and metric-enhanced constraint programming approaches.

Costagliola et. al. [6] based their approach on a visual language parsing technique. The design pattern recognition was reduced to recognizing sub-sentences in a class diagram, where each sub-sentence corresponds to a design pattern specified by an XPG grammar. Their process consist of two phases: the input source code is translated into a

class diagram represented in SVG format; then DPRE (Design Pattern Recovery Environment) recovers design patterns using an efficient LR-based parsing approach.

Tonella and Antoniol [22] presented an interesting approach to recognize design patterns. They did not use a library of design patterns as others did but, instead, discovered recurrent patterns directly from the source code. They employed concept analysis to recognize groups of classes sharing common relations. The reason for adapting this approach was that a design pattern could be considered as a formal concept. They used inductive context construction which then helped them to find the best concept.

3 Participating Systems

In this study we compared the design pattern mining capabilities of three tools, namely Maisa, CrocoPat and Columbus. Maisa and CrocoPat cannot analyze source code, so we extended our Columbus framework (whose original task was to analyze C++ source code and build an ASG – Abstract Semantic Graph representation from it) to produce input files for the two tools. So, our study was based on the same input facts, this way ensuring a fair-minded comparison, because eventual parsing errors affected all tools in the same way. We illustrate this process in Figure 1.

In the next sections we will introduce the compared design pattern miner tools. We will show every tools' design pattern description language on the well-known Factory Method design pattern.

3.1 Maisa

Maisa is a software tool [20] for the analysis of software architectures developed in a research project at the University of Helsinki. The key idea in Maisa is to analyze design level UML diagrams and compute architectural metrics for early quality prediction of a software system.

In addition to calculating traditional (object-oriented) software metrics such as the Number of Public Methods, Maisa looks for instances of design patterns (either generic ones such as the well-known GoF patterns or user-defined special ones) from the UML diagrams representing the software architecture. Maisa also incorporates metrics from different types of UML diagrams and execution time estimation through extended activity diagrams.

Maisa uses constraint satisfaction [17], which is a generic technique that can be applied to a wide variety of tasks, in this case to mining patterns from software architectures or software code. A constraint satisfaction problem (CSP) is given as a set of variables and a set of constraints restricting the values that can be assigned to those variables. Maisa's design pattern description language is very similar to Prolog. Figure 2 shows the description of the Factory Method pattern in Maisa.

```
class("Product").
class("ConcreteProduct").
extends("ConcreteProduct","Product").
!same(Product,ConcreteProduct).
class(Creator).
method("Creator.FactoryMethod()").
has("Creator","Creator.FactoryMethod()").
returns("Creator.FactoryMethod()","Product").
class("ConcreteCreator").
extends("ConcreteCreator","Creator").
method("ConcreteCreator.FactoryMethod()").
has("ConcreteCreator","ConcreteCreator.FactoryMethod()").
creates("ConcreteCreator.FactoryMethod()","ConcreteProduct").
implements("ConcreteCreator.FactoryMethod()","Creator.FactoryMethod()").
returns("ConcreteCreator.FactoryMethod()","Product").
!same(Creator,ConcreteCreator).
!same(Product,Creator).
!Object.
binded(Object).
same(Product,Object).
```

Figure 2. Factory Method pattern in Maisa

3.2 CrocoPat

In Section 2 we already introduced the CrocoPat tool [3] briefly, and now we will describe it in detail. First we will show how the CrocoPat interpreter works, and then we will introduce the relational manipulation language. Finally we will also mention the binary decision diagrams (BDD).

Previously, we mentioned that CrocoPat is an interpreter, and it executes RML programs. First, CrocoPat reads the graph representation in rigi standard file format (RSF) [19] from the standard input. Afterwards, the RML description is processed and a BDD representation is created from it. Finally, the RML program is executed and an RSF output is produced.

The RML (Relational Manipulation Language) is very similar to logic programming languages like Prolog, but it contains techniques of imperative programming languages too. Hence, it is very expressive and it can describe design patterns among other structures. Unfortunately, we have not found any design pattern library in RML, so we had to create the descriptions of the patterns by ourselves. Figure 3 shows the Factory Method description in CrocoPat.

To sum up, the main goals of Beyer et. al. [4] was efficiency and easy integration with other tools when they developed CrocoPat. Integration was facilitated by the import and export of relations in the simple Rigi Standard Format (RSF), and efficiency was achieved by representing the relations as binary decision diagrams [5].

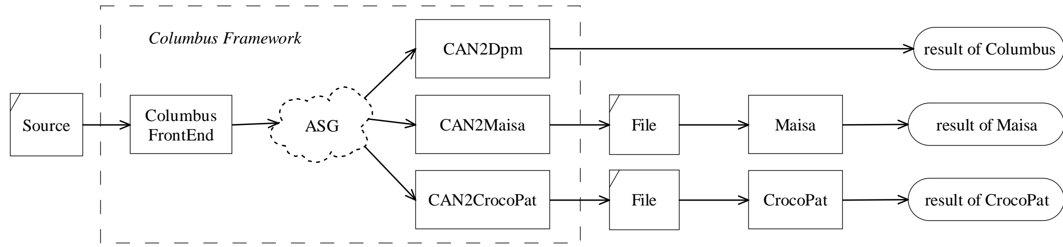

Figure 1. Common framework

```
AbstractClass(X) := CLASS(X) & ABSTRACT(X);
Product(X) := AbstractClass(X);
ConcreteProduct(Cpr,Pr) := CLASS(Cpr) & Product(Pr) &
        TC(INHERITANCE(Cpr,Pr));

Creator(Cr,Pr) := AbstractClass(Cr) & ASSOCIATION(Cr,Pr) & Product(Pr) &
        Cr != Pr;
CreatMethods(Cr,Pr,M) := Creator(Cr,Pr) & HASMETHOD(Cr,M);
CreatorFM(Cr,Pr,FM) := CreatMethods(Cr,Pr,FM) & VIRTUAL(FM) &
        PUREVIRTUAL(FM) & RETURNS(FM,Pr);
CreatorAM(Cr,Pr,AM,FM) := CreatMethods(Cr,Pr,AM) &
        CreatorFM(Cr,Pr,FM) & CALLS(AM,FM);

ConcreteCreator(Ccr,Pr,Cr,Cpr) := CLASS(Ccr) & ASSOCIATION(Ccr,Pr) &
        Product(Pr) & Creator(Cr,Pr) & TC(INHERITANCE(Ccr,Cr)) &
        ConcreteProduct(Cpr,Pr) & Ccr != Pr;
CCreatorFM(Ccr,Pr,Cpr,M) := ConcreteCreator(Ccr,Pr,_,Cpr) &
        HASMETHOD(Ccr,M) & VIRTUAL(M) & !PUREVIRTUAL(M) &
        RETURNS(M,Pr) & CREATES(M,Cpr);

FactoryMethod(Prod,Creat,CProd,CCreat,CreatFM,CreatAM,CcreatFM) :=
       Product(Prod) &
       Creator(Creat,Prod) &
       ConcreteProduct(CProd,Prod) &
       ConcreteCreator(CCreat,Prod,Creat,CProd) &
       CreatorFM(Creat,Prod,CreatFM) &
       CreatorAM(Creat,Prod,CreatAM,CreatFM) &
       CCreatorFM(CCreat,Prod,CProd,CcreatFM) &
       CProd != CCreat &
       Creat != CProd;
```

Figure 3. Factory Method pattern in CrocoPat

3.3 Columbus

Columbus is a reverse engineering framework, which has been developed in cooperation between FrontEndART Ltd., the University of Szeged and the Software Technology Laboratory of Nokia Research Center. Columbus is able to analyze large C/C++ projects and to extract facts from them. The main motivation to develop the Columbus system has been to create a general framework to combine a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework tool which supports project handling, data extraction, data representation, data storage, filtering and visualization. All these basic tasks of the reverse engineering process for the specific needs are accomplished by using the appropriate modules (plug-ins) of the system. Some of these plug-ins are provided as basic parts of Columbus, while the system can be extended to meet other reverse engineering requirements as well. This way we have got a versatile and easily extendible tool for reverse engineering.

One of the plug-ins is CAN2Dpm, which discovers design patterns. The design patterns were described in DPML (Design Pattern Markup Language) files, which store information about the structures of the design patterns. CAN2Dpm recognizes design patterns in the following way. First, Columbus analyzes the source code and builds an Abstract Semantic Graph (ASG) that contains all the information about the source code. Then CAN2Dpm loads a DPML file which also basically describes a graph. Afterwards it tries to match this graph to the ASG using our algorithm described in previous work [2]. Figure 4 shows the Factory Method description in Columbus.

The other two plug-ins of Columbus shown in Figure 1, CAN2Maisa and CAN2CrocoPat, are responsible for creating the input files for Maisa and CrocoPat, respectively.

4 Comparison Approach

In this section we will present the comparison approach of the investigated design pattern mining tools concerning:

- *The found design pattern instances.* Differences are caused by several reasons. The different tools use different techniques to define and describe what is a design pattern. The recognition algorithms are also different. The comparison of the found design pattern instances is just one of the several kinds of evaluation that should be considered, therefore we measure other important characteristics like speed and memory.

- *Speed.* Speed is measured by the amount of the time taken by the tool to perform the selected design pattern mining on the selected C++ project. The differences in the time of the measuring process in the examined systems are described in Section 5.2.

```
<?xml version=’1.0’?>
<!DOCTYPE DesignPattern SYSTEM ’dpml-1.6.dtd’>

<DesignPattern name=’Factory Method’>
    <Class id=’id10’ name=’Creator’ isAbstract=’true’>
        <Association ref=’id30’ />
        <Operation id=’id11’ name=’FactoryMethod’ kind=’normal’
                            isVirtual=’true’ isPureVirtual=’true’>
            <hasTypeRep ref=’id50’/>
        </Operation>
        <Operation id=’id12’ name=’AnOperation’ kind=’normal’>
            <calls ref=’id11’/>
            <hasTypeRep ref=’id54’/>
        </Operation>
    </Class>

    <Class id=’id20’ name=’ConcreteCreator’>
        <Base ref=’id10’ />
        <Association ref=’id30’ />
        <Operation id=’id21’ name=’FactoryMethod’ kind=’normal’
                            isVirtual=’true’ isPureVirtual=’false’>
            <creates ref=’id40’ />
            <hasTypeRep ref=’id50’/>
        </Operation>
    </Class>

    <Class id=’id30’ name=’Product’ isAbstract=’true’>
    </Class>

    <Class id=’id40’ name=’ConcreteProduct’ isChangeable=’true’ >
        <Base ref=’id30’ />
    </Class>

    <TypeRep id=’id1’/>

    <TypeRep id=’id50’>
        <TypeFormerPtr/>
        <TypeFormerFunc>
            <hasReturnTypeRep ref=’id52’/>
        </TypeFormerFunc>
    </TypeRep>

    <TypeRep id=’id52’>
        <TypeFormerType ref=’id30’/>
    </TypeRep>

    <TypeRep id=’id54’>
        <TypeFormerFunc>
            <hasReturnTypeRep ref=’id56’/>
        </TypeFormerFunc>
    </TypeRep>

    <TypeRep id=’id56’>
        <TypeFormerType ref=’id1’/>
    </TypeRep>
</DesignPattern>
```

Figure 4. Factory Method pattern in DPML

- *Memory usage.* Memory usage is another performance measure. We measured the total memory required for the design pattern mining task. It was complicated because the memory usage of CrocoPat is fixed, and only the Columbus source code was available to extend it to provide us with memory usage statistics. The applied memory measuring method for the examined tools are described in Section 5.3.

We did the comparison on four open source small-to-huge systems, to make the benchmark results independent from system characteristics like size, complexity and application domain. These four real-life, freely available C++ projects are the following.

- *DC++ 0.687.* Open-source client for the Direct Connect protocol that allows to share files over the internet with other users [7].
- *WinMerge 2.4.6.* Open-source visual text file differentiating and merging tool for Win32 platforms [23].
- *Jikes 1.22-1.* Compiler that translates Java source files as defined in The Java Language Specification into the byte-coded instruction set and binary format defined in The Java Virtual Machine Specification [15].
- *Mozilla 1.7.12.* All-in-one open source Internet application suite [18]. We used a checkout dated March 12, 2006.

Table 1 presents some information about the analyzed projects. The first row shows how many source and header files were analyzed in the evaluated software systems. The second row lists the size of these source and header files in megabytes.

The last two rows were calculated by the metric calculator plug-in of Columbus, and gives information about the total lines of code (LOC) and the number of classes. Under the term of *LOC* we mean every line in source code that is not empty and is not a comment line (also known as “logical lines of code”).

Size info.	DC++	WinMerge	Jikes	Mozilla
No. of files	338	512	74	11,325
Size (MB)	3	5.3	3	127
LOC	12,727	49,809	52,169	1,288,869
No. of classes	68	174	258	5,467

Table 1. Size information of the projects

All tests are run on the same computer so the measured values are independent from the hardware and thus the results are comparable. Our test computer had a 3 GHz Intel Xeon processor with 3 GB memory. In the next chapter we will describe our benchmark results and evaluate them in detail.

5 Results

In this section we will present our results concerning the differences between the design pattern instances found, the running-time and the memory requirements. In the next subsection we will start with the discovered pattern instances, and then compare the time efforts of the tools. Finally, we will show the memory requirements of the design pattern mining tools.

5.1 Discovered Pattern Instances

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	0	0	0
Adapter Class	0	2	0
Adapter Object	0	0	0
Bridge	0	-	0
Builder	0	0	0
Chain Of Responsibility	0	-	0
Decorator	0	-	0
Factory Method	0	0	0
Iterator	0	0	0
Mediator	0	0	0
Prototype	0	0	0
Proxy	0	0	0
Singleton	0	0	0
State	14	-	14
Strategy	14	-	14
Template Method	0	-	0
Visitor	0	0	0

Table 2. DC++ hits

In this section we will present our experiments regarding pattern instances found by the compared design pattern miner tools. Unfortunately, Maisa did not contain descriptions of the patterns Bridge, Chain of Responsibility, Decorator, State, Strategy and Template Method (the results for these are marked with dashes in our tables). We have investigated the differences between the tools manually, so we checked and compared the found instances and the descriptions of design patterns in all of the cases. We will not explain every difference, because there is not enough space for it, but we will present the most common causes.

First, we summarize our results on DC++ in Table 2. This was a small software system, so it did not contain too many design pattern instances. Maisa found two Adapter Classes, while CrocoPat and Columbus found none. This is due to the fact that the definition of the Adapter Class in Maisa differed from those in Columbus and CrocoPat. In Maisa the Target participant class was not abstract and the Request method of the Target class was not pure virtual, while in Columbus and CrocoPat these features were requested. We have examined the two Adapter Class hits in Maisa, and we have found that the Targets were not abstract in these cases and the Request operations were not pure virtual. Columbus and CrocoPat found 14 State and 14 Strategy design pattern instances. The cause of the identical number of hits is that the State and Strategy patterns share the same static structure, so their description in the tools were the same as well [2].

Table 3 shows the results of the tools in the case of WinMerge. Maisa found two more Adapter Objects in WinMerge than Columbus. In the first case the difference was caused by the fact that the Request method of a participant Adapter Object class was defined virtual in Columbus while Maisa did not have this precondition. In the second case the found pattern in Maisa had a Target participant that was not abstract, which was a requirement in Columbus. If we relaxed the description of this pattern in Columbus, it found these two instances too. The best solution would be if an exact definition existed for this pattern in both tools. CrocoPat found six Adapter Object instances, while Columbus found only three. The cause was that if Columbus found a pattern instance with certain participant classes and another pattern instance existed with the same participant classes but participating with different methods, *Columbus considered it as being the same pattern*. This is a very important difference between Columbus and CrocoPat, so we will refer to this difference several times. Maisa found a Builder in WinMerge but the two other tools did not, because in Maisa the Builder pattern representation did not contain the Director participant while the two other tools did contain it. In the case of State, Strategy and Template Method the differences were due to that Columbus counted pattern instances participating with different methods only once, like in the case of Adapter Object.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	0	0	0
Adapter Class	0	0	0
Adapter Object	3	5	6
Bridge	0	-	0
Builder	0	1	0
Chain Of Responsibility	0	-	0
Decorator	0	-	0
Factory Method	0	0	0
Iterator	0	0	0
Mediator	0	0	0
Prototype	0	0	0
Proxy	0	0	0
Singleton	0	0	0
State	3	-	10
Strategy	3	-	10
Template Method	2	-	42
Visitor	0	0	0

Table 3. WinMerge hits

Next, we will describe our experiments on design pattern instances found in Jikes (see Table 4). Maisa found an Adapter Class, while Columbus and CrocoPat did not. The reason was the same as in the case of DC++, namely that in Maisa the Target participant class was not abstract and the Request method of the Target class was not pure virtual but in Columbus and CrocoPat these features were required. In the case of Adapter Object Maisa missed a lot of hits, while Columbus and CrocoPat could discover a lot of design pattern instances. It looked like CrocoPat found more instances because Columbus counted repeating patterns with different operations only once. Actually, these tools *found the same pattern instances*. In Maisa the Builder pattern representation did not contain the Director participant, so Maisa found

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	0	0	0
Adapter Class	0	1	0
Adapter Object	78	10	94
Bridge	0	-	0
Builder	0	1	0
Chain Of Responsibility	0	-	0
Decorator	0	-	0
Factory Method	0	0	0
Iterator	0	0	0
Mediator	0	4	0
Prototype	84	0	84
Proxy	53	74	66
Singleton	0	0	0
State	170	-	334
Strategy	170	-	334
Template Method	4	-	4
Visitor	0	23	0

Table 4. Jikes hits

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	5	1	9
Adapter Class	0	59	0
Adapter Object	65	57	247
Bridge	880	-	1100
Builder	0	11	0
Chain Of Responsibility	0	-	0
Decorator	0	-	0
Factory Method	0	67	0
Iterator	0	0	0
Mediator	0	2	0
Prototype	83	25	901
Proxy	0	1	0
Singleton	8	0	20
State	722	-	7662
Strategy	722	-	7662
Template Method	279	-	522
Visitor	0	30	0

Table 5. Mozilla hits

an incomplete Builder instance in Jikes. CrocoPat did not find any Mediator in Jikes, while Maisa found four. It is due to that Maisa described Mediator in a very special way, so that it contained a Mediator with two Colleagues, but Concrete Mediators were missed. The description of Mediator in CrocoPat required a Mediator abstract class with a child ConcreteMediator class, too. In the case of Proxy, every tool discovered 53 instances, but CrocoPat counted also repeating patterns with different methods. Maisa found 21 instances more because it did not require an abstract Proxy participant class in the Proxy design pattern. In the case of State and Strategy it seemed that Columbus found less design pattern instances but it counted every repeated pattern with different methods only once. Maisa found 23 Visitor patterns, that the two other tools did not. This is due to the loose description of this pattern in Maisa.

Table 5 shows our experiments in the case of Mozilla. A lot of design pattern instances were found, like in the case of State, where CrocoPat found 7662 and Columbus discovered 722 instances. This huge difference was due to the fact that the found design pattern instances were not grouped by CrocoPat, that is, if a design pattern contained a class with child classes where the child classes could be of arbitrary number, every repeated child class with the common parent appeared as a new hit. *Columbus recognized this situation* and handled it correctly. In the case of Adapter Class the causes of differences were the same as in Jikes and in DC++ examined earlier. Columbus did not count repeated instances in the case of Adapter Object, so it actually found the same instances as CrocoPat, but Maisa missed some because of its different pattern description. CrocoPat and Columbus found the same instances of the Bridge pattern but Columbus counted the repeating patterns with different operations only once. Maisa found false Builder instances again, because the description of this pattern did not contain the Director participant class. Maisa found Factory Methods instances while the two other tools did not. This is due to that the two other tools defined Factory Method with an abstract Product and an abstract Creator participant class, while Maisa did not require these participants to be abstract. CrocoPat did not find any Mediator instance in Mozilla, while Maisa discovered two instances. This is due to that Maisa described Mediator in a very special way, so it contained a Mediator with two Colleagues, but Concrete Mediators were missing. In the case of Prototype, Singleton, State, Strategy and Template Method the differences were caused again by that CrocoPat counted every repeated pattern instance while Columbus counted these repeated ones with different operations only once.

Because of space limitation we cannot explain every difference, but we have shown the common reasons. Basically, the found design pattern instances would be the same in most of the cases if we could disregard the following common causes of differences:

- *Different definitions of design patterns.* We have found that there were some specific reasons for that the tools discovered different pattern instances. The main reason was in some cases that a design pattern description missed a participant like in the case of the Builder pattern in Maisa. In this case the pattern definition did not contain the director participant, thus the instances discovered by Maisa differed from the results of the other tools. For example, the results of Maisa in WinMerge for the Builder pattern differed from those of CrocoPat and Columbus for this reason.

- *Precision of pattern descriptions.* Another difference was how precise and strict the pattern descriptions were. For example, in the case of Jikes the differences in the numbers of found Adapter Class instances were

caused by the fact that CrocoPat and Columbus defined the Target as abstract while Maisa did not.

- *Differences in algorithms.* We have perceived differences in the design pattern miner algorithms, too. Columbus and Maisa counted the repeated instances with different operations only once while CrocoPat counted every occurrence.

5.2 Pattern Mining Speed

In this section we will present and compare the speed performance of the three assessed design pattern miner tools. We wanted to measure only the search time for patterns, therefore we divided the running time into two parts, an initialization part and a pattern mining part. Tables 6, 7, 8 and 9 contain the values of the pattern mining time only. Table 10 contains the initialization time of the tools (time format: hh:mm:ss).

The design pattern mining time was measured in the following way:

- *Columbus.* We took into account only the graph matching time, so we did not consider the time while the ASG was loaded. The graph loading time is presented in Table 10.

- *CrocoPat.* In the case of CrocoPat, we have prepared a small tool which executed CrocoPat and measured its running time. We measured the time needed for every pattern mining procedure for every subject software system. Next, we also measured the time for the subject systems with an empty RML program, because this way we could measure the time necessary to reserve the memory and to prepare the BDD representation (initialization time). These results are shown in Table 10. Finally, we subtracted the initialization time from the full running time for every result, and this way obtained the pattern matching times.

- *Maisa.* Maisa created statistics for every pattern mining procedure, which contained information about the time necessary for pattern mining, so we used these generated statistics. Contrary to CrocoPat, there was no need to extract the initialization time, because time values in the generated statistics measured only the pattern mining phase. However, we also show the initialization time for Maisa in Table 10.

First we show our results for DC++ (see Table 6). In this case the required time was very small for every assessed pattern miner tool, therefore they can be considered as being equal. This is due to the small size of the DC++ system, hence the design pattern instances were discovered very quickly in this system by all three tools.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	00:00:00	00:00:00	00:00:00
Adapter Class	00:00:00	00:00:00	00:00:00
Adapter Object	00:00:00	00:00:01	00:00:00
Bridge	00:00:00	-	00:00:00
Builder	00:00:00	00:00:01	00:00:00
Chain Of Responsibility	00:00:00	-	00:00:01
Decorator	00:00:00	-	00:00:00
Factory Method	00:00:00	00:00:00	00:00:01
Iterator	00:00:00	00:00:01	00:00:00
Mediator	00:00:00	00:00:01	00:00:00
Prototype	00:00:00	00:00:00	00:00:00
Proxy	00:00:00	00:00:01	00:00:00
Singleton	00:00:00	00:00:00	00:00:00
State	00:00:00	-	00:00:00
Strategy	00:00:00	-	00:00:00
Template Method	00:00:00	-	00:00:00
Visitor	00:00:00	00:00:00	00:00:00

Table 6. DC++ times

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	00:00:00	00:00:02	00:00:07
Adapter Class	00:00:00	00:00:00	00:00:07
Adapter Object	00:00:00	00:00:17	00:00:08
Bridge	00:00:00	-	00:00:08
Builder	00:00:00	00:00:24	00:00:09
Chain Of Responsibility	00:00:00	-	00:00:08
Decorator	00:00:00	-	00:00:09
Factory Method	00:00:00	00:00:02	00:00:01
Iterator	00:00:00	00:00:21	00:00:01
Mediator	00:00:00	00:00:24	00:00:04
Prototype	00:00:00	00:00:02	00:00:08
Proxy	00:00:00	00:00:22	00:00:14
Singleton	00:00:00	00:00:00	00:00:07
State	00:00:03	-	00:00:08
Strategy	00:00:03	-	00:00:08
Template Method	00:00:01	-	00:00:06
Visitor	00:00:00	00:00:00	00:00:05

Table 7. WinMerge times

The time requirements for discovering patterns in WinMerge (see Table 7) and Jikes (see Table 8) were different. Columbus was very fast in the case of larger patterns, because it could filter out [2] a lot of class candidates at the beginning of the discovering process. Opposite to this, Columbus was slower in the case of smaller patterns, because in these cases a lot of class candidates remained for the detailed discovering process. CrocoPat's and Maisa's time requirements were very balanced.

Finally, Table 9 shows the results for Mozilla. In most cases, CrocoPat delivered the best results, but in certain cases Columbus and Maisa were faster. Columbus was slow when it could filter out only a small amount of class candidates at the beginning of the discovering process. The CSP algorithm of Maisa was also slow in this case.

Our conclusion was that the best tool regarding speed in general is CrocoPat, but in some cases Columbus was faster.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	00:00:00	00:00:06	00:00:09
Adapter Class	00:00:00	00:00:04	00:00:07
Adapter Object	00:00:09	00:00:11	00:00:07
Bridge	00:00:00	-	00:00:06
Builder	00:00:08	00:00:59	00:00:07
Chain Of Responsibility	00:00:00	-	00:00:07
Decorator	00:00:00	-	00:00:18
Factory Method	00:00:00	00:00:04	00:00:02
Iterator	00:00:00	00:00:55	00:00:07
Mediator	00:00:00	00:01:05	00:00:12
Prototype	00:04:18	00:00:04	00:00:07
Proxy	00:00:00	00:01:03	00:00:13
Singleton	00:00:00	00:00:00	00:00:11
State	00:04:48	-	00:00:12
Strategy	00:04:48	-	00:00:12
Template Method	00:03:55	-	00:00:06
Visitor	00:00:00	00:00:06	00:00:14

Table 8. Jikes times

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	00:02:32	00:18:21	00:13:43
Adapter Class	00:00:06	00:18:34	00:14:34
Adapter Object	00:04:41	03:07:03	00:13:42
Bridge	04:50:29	-	00:17:20
Builder	01:39:09	04:09:22	00:14:02
Chain Of Responsibility	00:00:07	-	00:13:33
Decorator	00:00:18	-	00:27:40
Factory Method	00:03:03	00:15:56	00:00:02
Iterator	00:00:07	03:54:12	00:14:19
Mediator	00:48:03	04:19:07	00:18:10
Prototype	01:24:55	00:14:21	00:25:49
Proxy	00:00:07	04:41:47	00:27:10
Singleton	00:00:02	00:00:00	00:13:17
State	04:09:20	-	00:20:22
Strategy	04:09:20	-	00:20:22
Template Method	00:14:46	-	00:13:27
Visitor	00:00:07	00:06:56	00:20:45

Table 9. Mozilla times

5.3 Memory requirements

In this section we will introduce and compare the memory usage of the three compared design pattern miner tools. We have measured the memory requirements of every design pattern mining procedure, but we show our results summarized in one table because we have found them very similar.

The memory measurement method in the examined systems was accomplished in the following way:

- *Columbus.* We have extended the tool, so that it reports statistics about its memory usage.

- *Maisa.* Maisa did not report the memory usage in its statistics, so we measured it by simply monitoring its peak memory usage on the task manager.

Subject system	Columbus	Maisa	CrocoPat
DC++	00:00:03	00:00:00	00:00:03
WinMerge	00:00:08	00:00:03	00:00:11
Jikes	00:00:11	00:00:06	00:00:12
Mozilla	00:05:33	00:01:32	00:03:12

Table 10. Initialization times

- *CrocoPat.* CrocoPat's memory usage is constant and can be set as a command line parameter. Therefore, we created a script that executed CrocoPat iteratively from 1 megabyte reserved memory up to 200 megabytes for every pattern mining process. We took the smallest possible value so that the pattern mining process still completed successfully.

Our experiment proved that the memory usage strongly depended on the size of the analyzed projects and it was independent from the searched design patterns. This was true for every pattern miner tool as it can be seen in Table 11.

Subject system	Columbus	Maisa	CrocoPat
DC++	37 (19)	10-11	2-3
WinMerge	71 (32)	13-14	10-11
Jikes	51 (26)	13-17	10-14
Mozilla	866 (330)	60-71	125-175

Table 11. Memory requirements in megabytes

In the case of Columbus the reserved memory was very large compared to the other tools. This is due to the fact that Columbus is a general reverse engineering framework and design pattern detection is only one of its many features. For this reason it uses an ASG representation, which contains all information about the source code (including detailed facts about statements and expressions not needed for design pattern detection) for all kinds of tasks. Right now, for technical reasons, the design pattern miner plug-in of Columbus does not work without the ASG (although it does not have to use it), but we wish to fix this in the future. Therefore, we measured the memory needed by Columbus also without the ASG and showed these numbers in parentheses in Table 11.

Note, in the case of CrocoPat and Maisa the reserved memory was smaller because their input contained only the information about the source code necessary for pattern detection.

After examining Table 11 we can conclude that in the aspect of memory requirement Maisa's performance was the best.

6 Conclusion and Future Work

In this paper we have presented a comparison of three design pattern miner tools: Columbus, Maisa and CrocoPat. We have compared them regarding patterns hits, speed and memory consumption. We have guaranteed the common input for the tools by analyzing the source code with the front end of Columbus and by creating plug-ins for producing the required files for the tools. This way, as a "side effect" of this work, we have extended our Columbus Reverse Engineering Framework with plug-ins for Maisa and CrocoPat. We conclude that the fastest tool is CrocoPat, and Maisa requires the least memory, while Columbus is an all-in-one solution for design pattern detection from C++ source code with comparable performance to the other two specialized tools.

Originally, Gamma et. al. [12] defined the design patterns to develop object-oriented applications in forward engineering. Therefore, pattern definitions were informal to make them easier to use in different languages and contexts. Consequently, the design pattern miner tools have a big problem in common, which is how to define a given design pattern. In this paper we have shown that the tools found different design pattern instances in common inputs mostly because of their different pattern definitions. Hence, a formal description of design patterns is very desirable.

In the future we plan to create a design pattern catalog for reverse engineers, where every design pattern will have a strict and formal description. With this new collection of design patterns the presented drawbacks in Section 5.1 (different definitions of design patterns, precision of pattern descriptions and differences in methods) can be avoided.

References

[1] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato. A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition. In *Proceedings of the 15th Australian Software Engineering Conference (ASWEC'05)*, pages 677–691. IEEE Computer Society, Feb. 2005.

[2] Z. Balanyi and R. Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, Sept. 2003.

[3] D. Beyer and C. Lewerentz. CrocoPat: Efficient pattern analysis in object-oriented programs. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003)*, pages 294–295. IEEE Computer Society, 2003.

[4] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. In *Transactions on Software Engineering (TSE'05)*, pages 137–149. IEEE Computer Society, Feb. 2005.

[5] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *Transactions on Computers*, pages 677–691. IEEE Computer Society, Feb. 1986.

[6] G. Costagliola, A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi. Design Pattern Recovery by Visual Language Parsing. In *Proceedings of the 9th Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 102–111. IEEE Computer Society, Mar. 2005.

[7] DC++ Project.
`http://sourceforge.net/projects/dcplusplus/`

[8] R. Ferenc, Á. Beszédes, L. Fülöp, and J. Lelle. Design Pattern Mining Enhanced by Machine Learning. In *Proceedings of the 21th International Conference on Software Maintenance (ICSM 2005)*, pages 295–304. IEEE Computer Society, 2005.

[9] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[10] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[11] The FUJABA Homepage.
`http://www.cs.uni-paderborn.de/cs/fujaba/`

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

[13] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design patterns identification. In *Proceedings of IJCAI Workshop on Modelling and Solving Problems with Constraints*, pages 57–64, Aug. 2001.

[14] Y.-G. Guéhéneuc, K. Mens, and R. Wuyts. A Comparative Framework for Design Recovery Tools. In *Proceedings of the 10th Conference on Software Maintenance and Reengineering(CSMR'06)*, pages 123–134. IEEE Computer Society, Mar. 2006.

[15] IBM Jikes Project. `http://jikes.sourceforge.net/`

[16] O. Kaczor, Y.-G. Guéhéneuc, and S. Hamel. Efficient Identification of Design Patterns with Bit-vector Algorithm. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 175–184. IEEE Computer Society, 2006.

[17] A. K. Mackworth. The logic of constraint satisfaction. *Artif. Intell.*, 58(1-3):3–20, 1992.

[18] The Mozilla Homepage. `http://www.mozilla.org/`

[19] H. A. Müller, K. Wong, and S. R. Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.

[20] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. Verkamo. Software Metrics by Architectural Pattern Mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress).*, pages 325–332, 2000.

[21] The Ptidej Homepage.
`http://ptidej.iro.umontreal.ca/`

[22] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, pages 230–238, Washington, DC, USA, 1999. IEEE Computer Society.

[23] WinMerge Project.
`http://sourceforge.net/projects/winmerge/`