Mining Design Patterns from Existing Projects Using Static and Run-Time Analysis

Michal Dobiš and Ľubomír Majtás

Faculty of Informatics and Information Technologies,
Slovak University of Technology, Bratislava, Slovakia
`dobism@zmail.sk, majtas@fiit.stuba.sk`

Abstract. Software design patterns are documented best practice solutions that can be applied to recurring problems. The information about used patterns and their placement in the system can be crucial when trying to add a new feature without degradation of its internal quality. In this paper a new method for recognition of given patterns in object-oriented software is presented. It is based on static and semidynamic analysis of intermediate code, which is precised by the run-time analysis. It utilizes own XML based language for the pattern description and the graph theory based approach for the final search. The proof of concept is provided by the tool searching for the patterns in .Net framework intermediate language and presenting the results using common UML-like diagrams, text and tree views.

Keywords: Design patterns, reverse engineering.

1 Introduction

It is not easy to develop a good design of a complex system. Experienced designer of object oriented software often rely on usage of design patterns that are known as good solutions to recurring problems. A design pattern can impact scalability of a system and defines the way of modification of the architecture without decreasing its quality. Patterns are based on experience of many software developers and they are abstractions of effective, reliable and robust solutions to repeating problems. A pattern describes a problem, its context and supplies some alternative solutions, each of them consisting of collaborating objects, their methods and services.

Introducing patterns to software design provides one more benefit. While class in the object oriented paradigm makes it easier to understand the code by encapsulating the functionality and the data, an instance of a design pattern identifies the basic idea behind the relations between classes. It is an abstract term that can help to understand design decisions made by the original developer. But how to find instances of design patterns in the implemented system, when they represent an idea rather than just an implementation template? A large system has usually long lifetime and original architect of a particular component can leave the team before the system life cycle ends. Modification of the internal structure of a software component is a common need during the maintenance or when developing it for a long time. Therefore the precise documentation of design decisions by describing used design patterns becomes necessary. Unfortunately, such documentation is mostly obsolete or absent at all.

There can be plenty of design pattern instances in a large system, and each pattern can have many implementation variants. Moreover, a code fragment can play role in more than one pattern. It is unlikely to expect a human being to be able to orientate in a large mesh of connected classes forming a system. Thus, automatic analysis and search have to be done. We focused our effort on analysis of intermediate code produced by compilers of modern object oriented programming languages like C#. This code is also directly executable so its runtime behavior can be inspected as well. With the proposed method and the implemented tool, the valid pattern instances can be found more precisely, so the developers can become familiar with the analyzed system easier and have all the information necessary to make the correct design decisions while modifying it. Besides that, the library of patterns is kept open and new ones can be added using a special editor or by directly editing XML files.

2 Related Work

There are some tools that support reverse engineering with design pattern support, but this feature is far from perfect and there is still research needed to be done. There are three main questions to answer when developing a solid design patterns mining tool.

First question that comes out is ”What to find?” It represents the challenge of defining a formal language for the design pattern description that would allow dynamic editing of the catalogue. For humans the most suitable is natural language as it is used in works like the GoF’s catalogue [7]. It is good for description of the pattern’s variability, but is insufficient for the computer processing. UML-based ”Design Pattern Modeling Language” [9] is visual language consisting of 3 models (specification, instantiation and class diagram) that uses extensions capability of UML but authors propose just the forward engineering way of model transformations. Mathematical logic with its power is introduced in ”LanguagE for Patterns’ Uniform Specification” (LePUS) [5]. It, however, is difficult to capture the leitmotif of pattern using LePUS - even GoF patterns are not fully written in it. Progressive way of pattern description applicable in the context of pattern mining utilizes XML - ”Design Pattern Markup Language” [2] defines basic requirements on each class, operation and attribute so the group of them can be considered a pattern instance.

The answer to the question ”Where to search?” starts with creation of system model. There are three basic ways how to get it automatically. First, static analysis of the source code comes in mind. It is straightforward, but the drawback is that it would require development of a new parser for each language analyzed system could use. Thus, lower level language would be useful - that is intermediate language like Java byte-code (JBC) or Common Intermediate Language (CIL).

All the languages of the .Net family (C#. Visual Basic, managed C++, J#, DotLisp, etc.) are translated to CIL when compiled. There exists also a transformation between the JBC and CIL using IKVM.NET [6]. Some behavioral analysis can be done directly from the statically written code - good examples are tools like Pinot (Pattern INference and recOvery Tool) [11] or Hedgehog [3] identify blocks of code and data flow between them to gain the simple image of the idea behind the code and its dynamic nature. More precise perspective of a system can be gain by dynamic analysis realizable through profiling API (e.g. CLR Profiler uses it). Running the system and monitoring its behavior might not be the most reliable method, but it is the only way how to gain some very useful information about it (e.g. Wendehals uses this, but his tool is not really automatic [14]).

Last of the three questions is "How to search?" There has been some works utilizing a graph theory to search the patterns in graph model of analyzed system, while mostly just static information were taken in account [2]. Similar to it, distance between graphs of an analyzed system and a pattern (difference in the similarity of graphs) was introduced in the literature [13]. A quick method of pattern recognition, useful mostly to speed up the search process in combination with something else, comes from the utilization of object oriented software metrics known from the algorithms for looking for "bad smells" in the code [1]. Finally, rules and logic programming come alive in "System for Pattern Query and Recognition" [12] in cooperation with already mentioned language LePUS with all its benefits and drawbacks.

3 Our Approach

The main reason for design pattern mining is making the system understandable and easier for modification. The problem of many mining approaches presented before is the relatively high false acceptance ratio, they often identify elements that just look like pattern instance, but they are certainly not. Therefore we decided to focus on preciseness of the searching method that would try to identify only correct pattern instances that meet all constrains a man could define on a pattern. The collaborating classes, methods and fields are marked as instance of particular design pattern when they really fit the design pattern definition; so the software engineer can gain benefits from using it.

The first information that a software developer learns about a design pattern is mostly its structure. This static information is very important but is useless without the knowledge of the idea represented by pattern. What we want from our products - systems and programs - is the behavior rather than the structure and that is what specifies pattern most accurately.

To provide reliable results we decided to take three different types of analysis that need to come to an agreement whether the examined elements complies with pattern constrains or not: structural, semidynamic and dynamic (run-time) analysis.

3.1 Structural Analysis

The first and the easiest phase of the precise system analysis is the recognition of all *types*, their *operations* and *data* (attributes, local variables, method parameters). The search algorithm is based on graph theory, so we form the three types of vertex from these elements. Following information are recognized for each of them:

– Unique identifier of each element.
– Recognition whether it is static.
– Identification of parent - the element in which the current one is defined inside.

Type

The *Type* vertex represents class, abstract class, interface, delegate, enumeration or generic parameter. It contains identification of element type (Boolean values indicating whether it is a class, interface, etc.). Special type of edge identifies the types the element derives from - this list contains the base type and all the interfaces implemented by it. Finally the child elements (inner classes, operations, attributes, etc.) are recognized forming new vertices connected to the current one with ”parent-identification” edge.

Operation

The business logic of applications is defined inside the methods and constructors. For these types of elements we recognize the following information:

– Identification whether the operation is virtual. If so, the overridden virtual or abstract element is searched.
– Recognition whether it is abstract.
– Distinction between the constructors and common methods.
– Returned type.
– Parameters and local variables of the operation.

Data

Attributes, parameters and local variables are represented as the vertices of *type* data having the original scope stored as the vertex attribute. Each data field has its type (identified by an edge connected to the vertex standing for the type). The type of collections is not important when looking for design pattern instances. The important information is the type of elements in the collection in combination with additional information, that the reference (the data field) is not only one instance, but it stays for a group of elements of the recognized type.

Event

Some modern object oriented languages define language level implementation of design pattern *Observer*. In .Net, the definition of events is similar to common class property - it has its type (delegate) and name. The difference is that during the compilation two new methods on the class containing the event are defined. These are used to attach and detach listeners (observers).

From the design pattern mining point of view we can say, that the events are single-purpose. The definition of an event and references to it exactly define the *Observer* pattern instance that is implemented correctly. It is clear that when looking for this kind of pattern implementation, we can mark each occurance of the key-word. Thus, we store this information in the system model, but do not discuss this in the paper bellow.

3.2 Semidynamic Analysis

Application behavior is represented by methods, invocations between them and object creations. Some information can be gathered already from static representation of an analyzed system - the most important information is the *call* from one operation to another one.

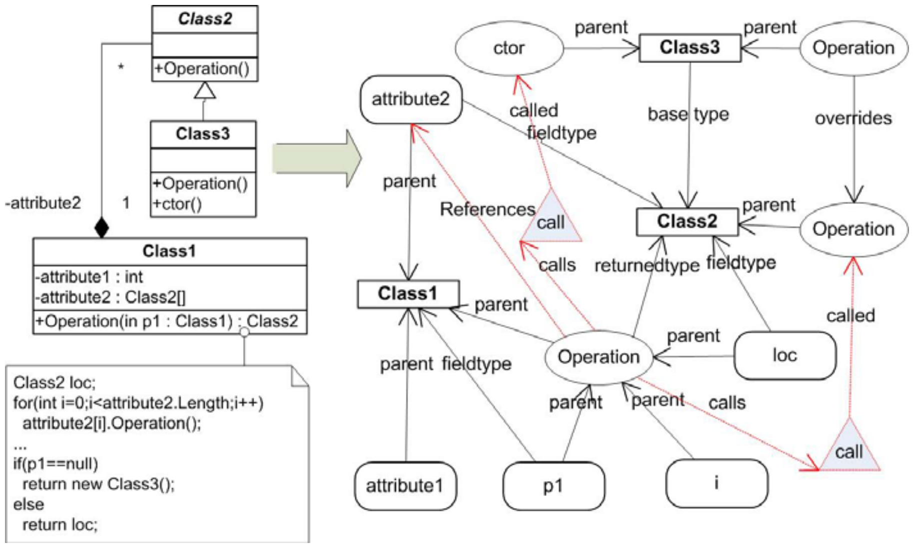
The necessity of the information about the calls was introduced in many recent works [2] [3] [11] concerning the design patterns mining. However, just a few of them inspected the whole body of operations in more detail. We find the placement of a method call (whether inside a condition, cycle or directly in a method) equally important as the basic recognition of the call. Most of the design patterns define the condition, that the particular call should be invoked just when an if-clause is evaluated as true (e.g. *Singleton*, *Proxy*) or it needs to be realized inside a cycle (e.g. *Composite*). It, however, should not forbid it - the inclusion inside a condition can be done due to many other reasons not affecting the design pattern instance. For some design patterns is also important the count of calls to a single operation placed inside the current operation (e.g. *Adapter*).

During the semidynamic analysis there we also analyze the class attributes. The access to a member variable from an operation brings some very useful information enhancing the image we can gain about the behavior from the statically written system. If a method plays the role of operation *GetInstance* in pattern *Singleton*, it is insufficient to have just the structural information, it needs to have a reference to the static field *instance* and modify it during the lazy initialization.

The Fig. 1 presents the example results of the static and semidynamic analysis.

3.3 Dynamic Analysis

The only time, we can really observe the behavior of a system, is the run time. When we execute the analyzed system we can see, what is really happening inside the system. To do so, we decided to enrich the model (graph) with four simple information collected automatically.



```
Class2 loc;
for(int i=0;i<attribute2.Length;i++)
  attribute2[i].Operation();
...
if(p1==null)
  return new Class3();
else
  return loc;
```

Fig. 1. Graph of the system structure enhanced with semidynamic information

Count of Instances

Several design patterns say how many instances a particular class should have during one system execution. The simplest example should be the *Singleton* pattern - it is obvious that there needs to be exactly one instance of the class. The suitability and the goal we want to achieve by counting instances of concrete types is obvious. We decided to increase the number also for class A when an instance of class B is created whereas B derives from A. The reason for doing this is the fact that patterns usually do not forbid to derive from classes playing the roles in pattern instance. Thus, there might be a class that derives from *Singleton* but the method *GetInstace* still has to return the same object - *Singleton* needs to have just one instance no matter how many subclasses of it are defined in the system.

Count of the Operation Invokes

In the previous section we have already introduced the importance of the information how many times an operation was invoked. The *Prototype* example illustrates the comparison of the count of operation *Clone* invokes to the count of instances of a particular type. More common however, is the comparison to count of invokes of calls done from a particular operation (useful when searching for e.g. *Composite* or *Proxy*). Thanks to this information it is easy to verify whether the placement of a call inside a loop really increases, resp. the placement inside a condition decreases, the count of call executions. Using this information we can check also whether e.g. *Composite* really does its job and groups more components forming a tree structure.

Less common is to compare count of invokes of two separate operations. The reason is that mostly we do not know whether the target operation really gets executed - there might be lazy binding and the pattern definition would be useless. We could increase the count of invokes not only for the operation really called, but also for the overridden one. It might seem similar to the situation we have with the classes (previous chapter) but there are some differences. First of all, it is sometimes useful to distinguish exactly the called operation and modifying the count of overridden method would make it a little bit harder. The most important difference, however, is that we have another useful information when considering the operation invokes - we can identify the source and desired target by monitoring invokes of the calls from one operation to another.

Count of the Calls Invokes

The call from an operation is mostly polymorphic - its target is a virtual method that can but (in case it is not abstract) does not have to be overridden in a subclass. As described in previous section, this information can serve as verification of the real usage of a loop or a condition during the system run time. A good example is the Proxy pattern, where we can observe whether the condition really leads to lower calls to the *RealSubject* and is not just a result of coding standards or an accident.

Count of Changes in Attributes

Some patterns require tracking the value-changes in data variables. It might be interesting to monitor changes in all of them including local variables and method parameters, but we do not find this really useful and it would lead to performance problems due to many logging. What we find useful and sufficient is to count, how many times a value in a class attribute changes.

The importance of tracking the changes can be seen in several design patterns -even when considering the very simple pattern *Singleton* we can propose a restriction that value stored in the static field *instance* should change just once. More changes would cause the program to loose the reference to the only instance, thus the pattern instance would behave incorrect.

The most important, however, is the value to identify patterns *Strategy* and *State*. They have been discussed many times in literature so far [2]. When trying to recognize these patterns in a system automatically, most of the approaches presented earlier simply sets the instance as "Strategy or State" - since they were using just the static analysis they were unable to distinguish one from other. The approaches that took the challenge and proposed the ability to distinguish them used some additional information, restriction that do not fit always and were added by them [3] [11]. Very common is the definition that a concrete state needs to know about its successor (one concrete state has reference to another concrete state) or the context simply creates the concrete states. This is usually true and is a very good assumption. We have identified different assumption that allows us to recognize these patterns: states should change, whereas strategies should not. This means that we need to know whether the attribute in *Context* class (its reference to abstract state or strategy) changes or not. We can say that the *Strategy* pattern says about algorithm a context can be configured with. Strategy encapsulates behavior used by a system, its part or a single class and it should not change during the client (*Context*) life time. Thus, the restriction we placed is, that there can not exist any class that contains a parameter of type *Strategy* (reference to abstract strategy) that changes too many times - the highest possible count of changes in the field referencing from the *Context* to the *Strategy* is the count of instances of the *Context* class. Having the *Strategy* clear, we know how to find a *State* - when all the other requirements (structural, semidynamic, other dynamic restrictions) are met, there needs to be a *Context* class that has an attribute of type *State* that was changed (the value stored in the variable was changed) more times than the *Context* class was instantiated.

3.4 Design Pattern Model and the Search

We have introduced several constraints that can be used to exact recognition of design pattern instances. Because of these new constraints, we had to develop a new pattern description that includes improved structural and behavioral information. As we mentioned earlier, the analyzed system is represented as a graph with four kinds of vertices: *type*, *operation*, *data* and *call*. Each type of vertex has its own attributes (as they are described in chapters 3.1, 3.2 and 3.3) and can be connected with the other vertices using typed edges. The pattern is a rule that describes constraints on a subgraph of this graph - its vertices including values of the attributes, types and edges.

Patterns usually define roles that might (or should be) played by more system elements - e.g. they define a *concrete strategy*, but expect that there will be more of them in a single instance of the pattern. It means that the multiplicity of items playing role *concrete strategy* should be at least 2. The multiplicity is crucial for all patterns (defining the minimal and maximal value of it for each pattern element) - it is clear that there should not be a public constructor available in the class *Singleton*, thus its multiplicity needs to be exactly 0. There needs to be at least one private constructor, so multiplicity of this element has the minimum at 1. The multiplicity is even more interesting when we look for an *Abstract Factory* instances - when we take the definition of pattern very precisely, we can propose a constraint on count of abstract products demanding it to be the same as the count of methods of the factory that create objects of different types. The minimal and maximal value is used also for all the information gathered using the dynamic analysis and can be set as constant number or as a reference to another numeric value evaluated on the pattern instance.

4 The Tool

To prove our concept we have developed a complex reverse engineering tool that is able to analyze any system built on the .Net platform and search the instances of design patterns defined in an editable catalogue (see Fig. 2 for the screenshot

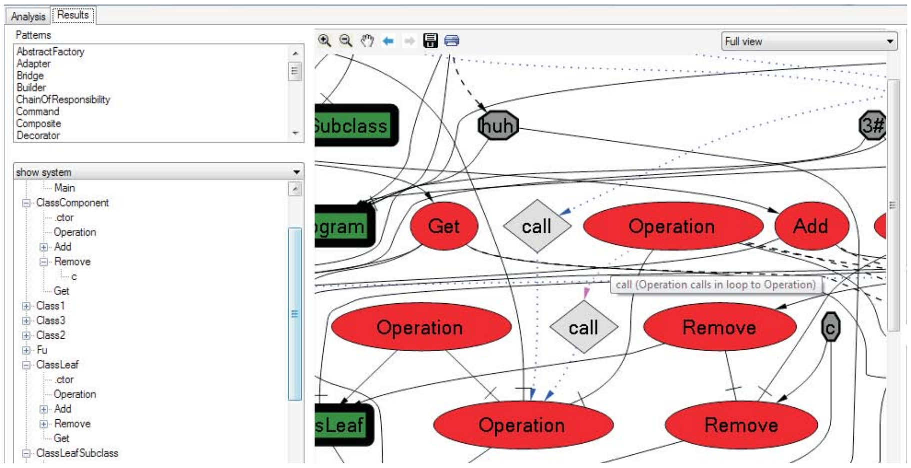


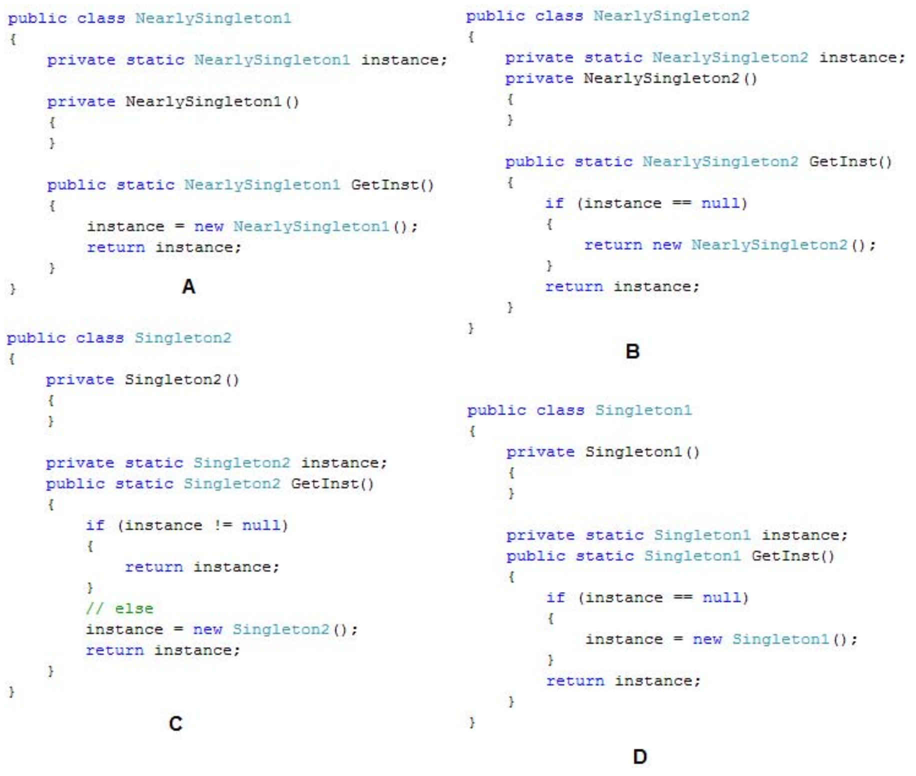
Fig. 2. The results dialog window of the tool

of the tool). The analysis process starts with selection of assemblies (DLL and EXE files) and specifying namespaces that should be included. The content of the assemblies is written in Common Intermediate Language, that is produced by every compiler of all the languages of the .Net family. CIL can be read in various ways. The simplest one is the usage of the reflection that is commonly available in the .Net environment and is sufficient for the needs of structural analysis. The content of the operations that is investigated by the rest of the analysis process is available as plain CIL. We find here all the references to attributes (data fields), method calls and recognize whether they are positioned inside a condition, loop or directly in the method body. Finally, the dynamic analysis is done using .Net profiling API that allows us to inject logging instructions into every operation of the analyzed system. This is done before Just-In-Time compilation (from CIL to machine code) of each method starts.

Before we could search for something, we needed to define it. Therefore we have created an editable catalogue of design patterns stored in an XML-based language that we have created from original Design Pattern Markup Language (DPML) [2] enhancing it by the features we added (we call it *DPMLd*). The search for design patterns instances itself is based on graph theory and utilizes GrGen.Net [8]. GrGen.Net allows a developer to define his own model of graph that contains typed vertices, each type of vertex having its own attributes and connected with other vertices using typed edges. Upon the defined graph, there can be a graph-rewrite-rule created. The rule consists of conditional and changing part and can be used to specify conditions on subgraphs we want to search for. The tool we created transforms the graph of analyzed system into the GrGen.Net form and produces the graph-rewrite-rules from specification of design patterns. Finally, it collects the found subgraphs - basic fragments of candidates of design pattern instance (e.g. group of vertices that represent an instance of State with only one concrete state). These fragments are grouped together if they have common ”group-defining” elements - this is for example the abstract state class, or the template method. Which elements are the ”group-defining” is defined by the pattern and it can be any type, operation or data. The last step of the search is done when subgraphs are grouped into the candidates. It consists of multiplicity checks verifying the correct count of elements in the pattern instance.

5 Results

The Fig. 3 illustrates the benefits of the individual steps of the analysis we proposed in this article. It contains wrong and good implementations of the simplest of the GoF design patterns - the *Singleton*.



```
public class NearlySingleton1
{
    private static NearlySingleton1 instance;

    private NearlySingleton1()
    {
    }

    public static NearlySingleton1 GetInst()
    {
        instance = new NearlySingleton1();
        return instance;
    }
}
```

```
public class NearlySingleton2
{
    private static NearlySingleton2 instance;
    private NearlySingleton2()
    {
    }

    public static NearlySingleton2 GetInst()
    {
        if (instance == null)
        {
            return new NearlySingleton2();
        }
        return instance;
    }
}
```

```
public class Singleton2
{
    private Singleton2()
    {
    }

    private static Singleton2 instance;
    public static Singleton2 GetInst()
    {
        if (instance != null)
        {
            return instance;
        }
        // else
        instance = new Singleton2();
        return instance;
    }
}
```

```
public class Singleton1
{
    private Singleton1()
    {
    }

    private static Singleton1 instance;
    public static Singleton1 GetInst()
    {
        if (instance == null)
        {
            instance = new Singleton1();
        }
        return instance;
    }
}
```

Fig. 3. Experiments with singleton pattern

Structural analysis is the first what comes in mind when we talk about mining of design patterns from existing projects. It checks many conditions - when considering the *Singleton*, it checks whether there is no public constructor and at least one private one in a class that contains static attribute of its own type and a static method returning the type. However, all these conditions are met already in the case of class *NearlySingleton1* (part A in the figure). Even more, the method *GetInstance* has reference to the instance data field and calls the private constructor. The semidynamic analysis checks all this and adds another feature - it requires the call to be placed inside a condition (if-clause). It might seem enough, but as we can see the *NearlySingleton2* (part B in the figure) fits also these constraints and still is not a correct instance of the singleton pattern. Finally, the dynamic analysis adds some something that ensures the correctness - it checks the count of instances of the class (to be more exact it checks also the count of value changes in the *instance* field and count of method and call invokes) and verifies that the *Singleton1* and *Singleton2* are the valid *Singleton* pattern instances in the system. Please note, that the call to the private constructor in the *Singleton2* is not placed in a condition as it is written in the code. However, we can say there is a hidden (suppressed) else block because the precious if-block ends with the return statement (the tool recognizes this correctly).

We have filled the catalogue of the implemented tool with all GoF design patterns having one group of implementation variants for each pattern. This means that all common implementation variants are found, but some correct pattern instances that rely on transitive relations might stay undiscovered. The resolution to this is by adjusting the level of accuracy of the pattern definition or by adding the definitions of the additional variants (all using the GUI of the tool; automatic transitive relations are to be added during future work). The *Observer* and *Iterator* patterns are also included. We, however, decided not to search for the language-level implementation of them - this would be just the keyword based search for *event* or *foreach.* We defined these two as they are described in the GoF catalogue and search just in the client code. As expected, none of the projects we tested was "reinventing the wheel" (they used the language level features instead of coding it at their own). Thus, we do not show these results in the tables bellow.

We have checked our tool on the code of our tool itself (*PatternFinder*). We chose our own tool for examination because it is an untrivial piece of code and we know about occurrence of all instances in it. The tool was able to find all the pattern instances correctly (except Mediator - it did not find it since it has colleagues made from system types that were skipped during the analysis process). It was also able to find some pattern instances we included unintentionally.

The other tests were made on both - the free (mostly open source) and the commercial projects on the .Net platform (since we do not need sources, we need just the DLL and EXE). The Table 1 presents results of the examination executed on the different projects. Please note that we looked just for a single variant of each design pattern (resp. small group of variants covered using single pattern description in the DPMLd pattern language we proposed). The variants can be modified or some new one can be added to the tool using the editor it contains or by writing it as the XML.

The table columns marked with S are based on static (structural and semi-dynamic) analysis and the D columns contain results that came from complete

Table 1. Experiments on existing projects

Pattern	PatternFinder		Paint.Net		SharpDevelop		NUnit	
	S	D	S	D	S	D	S	D
Abstract Factory	0 / 0 / 0	0 / 0 / 0	14 / 1 / 0	7 / 1 / 0	6 / 2 / 0	2 / 1 / 0	14 / 1 / 0	4 / 1 / 0
Adapter	389 / 25 / 7	152 / 16 / 7	1585 / 73 / 24	451 / 30 / 5	13 568 / 285 / 71	1 249 / 100 / 39	247 / 28 / 11	30 / 12 / 7
Bridge	0 / 0 / 0	0 / 0 / 0	188 / 2 / 2	0 / 0 / 0	3040 / 7 / 0	193 / 3 / 0	96 / 1 / 1	27 / 1 / 1
Builder	9 / 1 / 1	6 / 1 / 1	20 / 1 / 1	12 / 1 / 1	624 / 14 / 14	66 / 3 / 3	27 / 1 / 1	0 / 0 / 0
Chain of Respon.	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	3 / 1 / 1	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Command	54 / 1 / 0	20 / 1 / 0	43 / 3 / 0	4 / 1 / 0	7054 / 16 / 3	2 565 / 5 / 1	398 / 4 / 1	147 / 4 / 1
Composite	96 / 1 / 1	48 / 1 / 1	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Decorator	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Facade	1 / 1 / 0	0 / 0 / 0	16 / 5 / 0	0 / 0 / 0	3 / 1 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Factory method	9 / 1 / 1	4 / 1 / 1	25 / 1 / 1	1 / 1 / 1	24 / 2 / 2	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Flyweight	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Interpret	0 / 0 / 0	0 / 0 / 0	12 672 / 2 / 2	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Mediator	23 / 1 / 1	14 / 1 / 1	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Memento	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Prototype	0 / 0 / 0	0 / 0 / 0	8 / 2 / 2	2 / 1 / 1	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Proxy	17 / 3 / 3	17 / 3 / 3	33 / 3 / 3	0 / 0 / 0	15 / 4 / 4	1 / 1 / 1	1 / 1 / 1	0 / 0 / 0
Singleton	6 / 2 / 2	6 / 2 / 2	0 / 0 / 0	0 / 0 / 0	7 / 2 / 2	7 / 2 / 2	0 / 0 / 0	0 / 0 / 0
State	42 / 1 / 0	0 / 0 / 0	325 / 18 / 12	110 / 2 / 1	1528 / 43 / 25	428 / 23 / 14	771 / 13 / 6	108 / 5 / 4
Strategy	394 / 11 / 11	237 / 9 / 9	591 / 43 / 23	171 / 16 / 11	7924 / 127 / 47	1476 / 75 / 26	161 / 18 / 11	212 / 18 / 10
Template Method	14 / 2 / 1	14 / 2 / 1	87 / 14 / 2	41 / 5 / 1	17 / 5 / 0	0 / 0 / 0	7 / 3 / 0	4 / 1 / 0
Visitor	82 / 1 / 1	51 / 1 / 1	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0

static and dynamic analysis Each table cell contains three numbers formatted like x / y / z, where:
x - count of all subgraphs (smallest fragments that meet the basic requirements).
y - count of all candidates that arrised from grouping of the subgraphs.
z - count of the found pattern instances.

6 Conclusions and Future Work

In the previous sections we have introduced a method for searching instances of design patterns in an intermediate code and a tool that is a proof-of-concept. We have focused on preciseness of search results, therefore we have proposed a method based on three different types of analysis. All of them need to agree before they mark the pattern instance, so they together provide interesting results with low false acceptance ratio.

The implemented tool provides identification of pattern instances in the intermediate language of the .Net platform. Choosing intermediate language as an input to the analysis removes the necessity of access to the source code. As a result we are able to check any software developed for the .Net platform, including COTS products provided by third parties. The search process is driven by a pattern specification which is written in an XML-based language. The tool provides a GUI editor for the language, what allows users to modify pattern descriptions or specify other patterns. Through modification of a pattern description a user can even configure the level of search preciseness.

In the future, we would like to improve our specification language to allow searching more variants of design patterns. An interesting approach in this area has been introduced by Nagl [10]. He uses feature modeling [4] for specification of design pattern variability. Another improvement can be gained by allowing the transitive relationships which are sometimes parts of the software design (for example indirect inheritance, multiple code delegation, etc.).

Acknowledgement. This work was partially supported by the project of APVV (Slovak Research and Development Agency), No. APVV-0391-06 ”Semantic Composition of Web and Grid Services” and the Scientific Grant Agency of Republic of Slovakia, grant No. VG1/3102/06.

References

1. Antoniol, G., Fiutem, R., Cristoforetti, L.: Using Metrics to Identify Design Patterns in Object-Oriented Software. In: Proceedings of the 5th International Symposium on Software Metrics, pp. 23–34. IEEE Computer Society, Washington DC (1998)
2. Balanyi, Z., Ferenc, R.: Mining Design Patterns from C++ Source Code. In: Proceedings of the International Conference on Software Maintenance, pp. 305–314. IEEE Computer Society, Washington DC (2003)

3. Blewitt, A., et al.: Automatic Verification of Design Patterns in Java. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM, New York (2005)
4. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. Software Process Improvement and Practice, Special Issue on Software Variability: Process and Management 10(2), 143–169 (2005)
5. Eden, A.H.: Precise Specification of Design Patterns and Tool Support in Their Application. PhD thesis, University of Tel Aviv (1999)
6. Frijters, J.: IKVM.NET Home Page, `http://www.ikvm.net/index.html` (accessed May 5, 2007)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley professional computing series (1995) ISBN 0-201-63361-2
8. Geiss, R., et al.: GrGen.NET, `www.grgen.net` (accessed November 4th, 2007)
9. Mapelsden, D., Hosking, J., Grundy, J.: Design Pattern Modelling and Instantiation using DPML. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, pp. 3–11. CRPIT Press, Sydney (2002)
10. Nágl, M. (supervised by Filkorn, R.): Catalog of software knowledge with variability modeling, Master thesis, Slovak university of technology, Faculty of informatics and software engineering (2008)
11. Shi, N., Olsson, R.A.: Reverse Engineering of Design Patterns from Java Source Code. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, Washington (2006)
12. Smith, McC, J., Stotts, D.: SPQR: Flexible Automated Design Pattern Extraction From Source Code. In: 18th IEEE International Conference on Automated Software Engineering, pp. 215–225. IEEE Computer Society, Washington DC (2003)
13. Tsantalis, N., et al.: Design Pattern Detection Using Similarity Scoring. IEEE Transactions on Software Engineering 32(11), 896–909 (2006)
14. Wendehals, L.: Improving design pattern instance recognition by dynamic analysis. In: Proceedings of the ICSE 2003 Workshop on Dynamic Analysis, pp. 29–32. IEEE Computer Society, Portland (2003)