Charles University in Prague
Faculty of Mathematics and Physics
Department of Software Engineering

Technical Report 2011/2

Štěpán Šindelář, Filip Zavoral

# Design Patterns Support in Development Environments

# Contents


**1 Introduction** **3**
1.1 Patterns4Net . . . 5
1.2 Report structure. . . . 5

**2 Design patterns support** **6**
2.1 Patterns formalization and verification . . . 6
2.2 The role of programming language . . . 9
2.2.1 New features . . . 10
2.2.2 New design problems . . . 10
2.2.3 Future trends . . . 12
2.3 Tools support . . . 12
2.3.1 Patterns4Net . . . 13

**3 Pattern Enforcer** **15**
3.1 Design patterns documentation . . . 15
3.1.1 Terminology . . . 15
3.1.2 Documentation of pattern instances . . . 15
3.1.3 Pattern instances representation . . . 16
3.2 Patterns structural constraints specification . . . 17
3.2.1 Domain specific languages . . . 18
3.2.2 The API for pattern constraints specification . . . 19
3.2.3 Built-in patterns . . . 20
3.3 Usage . . . 22
3.3.1 Unit tests . . . 24
3.3.2 Specification of custom pattern . . . 25
3.4 Architecture . . . 29
3.4.1 CIL processing . . . 29
3.4.2 Patterns representation and discovery . . . 30
3.4.3 CIL analysis . . . 32
3.4.4 Pattern Enforcer Design . . . 34
3.4.5 Development infrastructure . . . 35
3.5 Comparison . . . 37
3.5.1 FxCop and Gendarme tools. . . . 37
3.5.2 Pattern Enforcing Compiler (PEC) for Java . . . 38
3.5.3 Other tools . . . 39

**4 Architecture Explorer** **40**
4.1 Features . . . 40
4.1.1 Abstraction levels . . . 40
4.1.2 Further meta-information . . . 41
4.2 User Interface . . . 42
4.3 Architecture . . . 43
4.3.1 User interface logic . . . 44
4.3.2 Diagram Classes Design . . . 45
4.4 Related Work . . . 46

4.4.1 Pattern recognition tools. . . . 46
4.4.2 UML reverse engineering. . . . 47

**5 Graphviz4Net** **48**
5.1 Public API . . . . 48
5.1.1 Graph representation . . . . 49
5.1.2 Layout builder . . . . 50
5.2 Architecture . . . . 50
5.2.1 DOT parsing. . . . . 50
5.2.2 WPF Support . . . . 50

**6 Conclusion** **52**

**Bibliography** **53**

# 1. Introduction

The concept of a design pattern as a reusable solution to a recurring problem was first introduced by Christopher Alexander in the field of architecture ([1]). His book gives design patterns such as *Outdoor Room*, or *Arcades* to architects. Although firstly used in the domain of architecture, over last two decades, design patterns have gained popularity also in computer science, especially in object-oriented design and programming.

*This report is about design patterns in object-oriented design and programming and in the following text the term "pattern" or "design pattern" refers to these kinds of patterns.*

In general [2], a design pattern consists of

- a name to provide a common vocabulary,
- a description of a problem and it's context,
- a proved and widely-accepted solution to this problem,
- the consequences of applying the pattern.

A design pattern provides a solution that cannot be implemented in a generic library or a framework. The abstract ideas behind a pattern are implemented again and again but in each concrete case a little bit differently. If we take the *Composite* pattern as an example, the problem it solves is to let clients treat individual objects and compositions of objects uniformly. The solution of the *Composite* pattern suggests to create a *Composite* class that composes children components and delegates it's operations to these children components. Note that the pattern's solution, in this case, does not say how exactly the composite operation must be implemented. The operation getWidth may return the width of the largest component, or it may return the mean of all widths. But one thing that should be fulfilled is that the composite operation uses it's children components to do it's work and this fact should be transparent to the clients. To make the example complete, let us mention one of the consequences. The *Composite* pattern makes it easier to add new kinds of components.

The main aim of patterns in object-oriented design is to make the design reusable and flexible. This is very important because changes in the functional requirements of software during the development, or requests for new features in an already developed software are quite usual these days. The mentioned consequence of the *Composite* pattern could be an evidence for this aspiration of design patterns.

In programming a typical mistake is to spend a fair amount of time by solving something that has already been solved by someone else. Patterns, among reusable libraries, frameworks and others, address and partly solves this problem. Another advantage of patterns is the common language, or common vocabulary. It makes the communication between developers much more effective when all of them understand what the *Visitor* pattern is. Even if two developers understand the complex logic behind this pattern, it would take them some time to find out that they both mean the same concept if they didn't know the common name for this pattern.

Since the first notable publication about patterns in the field of object-oriented design by so-called Gang of Four [2], there has been a great number of books about patterns each focusing on a different kind of patterns. For example, so called business patterns described in [3], or the enterprise patterns from [4]. The principles discussed in [5] might be as well considered as patterns, although on a higher level of abstraction than the original design patterns. We could continue to enumerate more of them. In this report we mainly focus on the design patterns as described in [2], where the authors define a design pattern as a

> description of communicating objects and classes that are customized to solve a general design problem in a particular context.

One of the disadvantages of design patterns is that they bring a new complexity into the design. This complexity is caused by an introduction of new classes and interfaces in order to provide better flexibility and reusability. Developers often don't have enough time to create a documentation for their classes and so the mapping between classes and a design patterns is lost. The other members of the development team can only study the source code, or reverse-engineered diagrams, but neither of these emphasize the design patterns structure, which would provide more abstract view and thus tackle some of the complexity.

Even if the code documentation includes information about implemented patterns, an incorrect understanding of some design patterns by one of the development team members may slow down the development process or even lead to an introduction of software bugs in the system. For instance, when one part of the system expects the objects of a specific type to be immutable, but a developer unaware of what the immutability means changes this behavior. In this case a formal verification might help.

While tools for a formal verification and tools for tackling the complexity of design patterns exist, they were mainly developed as research prototypes and, except for few of them, they didn't get enough attention from the industry. Moreover, most of these tools target the Java platform, but only few target the .NET platform.

Some of the reasons why the industry is not adopting design patterns verification tools may be too much mathematical formalism involved in their usage. For a definition of new patterns, the knowledge of formal logic is usually required. Tools for tackling the complexity of design patterns are mostly based on an automatic recognition of design patterns, whose advantage is that it does not require additional work from developers and can be used for legacy systems, but it's disadvantage is that it cannot correctly recognize all the design patterns, since the differences between some of them are only semantical (the *Bridge* and the *Adapter* patterns) and some patterns, such as the *Command* pattern, are too much abstract to be recognized only from the source code [6].

The problems described in the previous paragraphs are addressed by the Patterns4Net project, whose presentation is the main aim of this report. Besides this, we also provide a brief overview of existing approaches for design patterns formalization, which is needed for formal verification and tool support, and we give a few examples of existing tools that provide support for design patterns.

## 1.1 Patterns4Net

Experienced developers who use design patterns make usually this intention explicit by some kind of documentation. For instance, leaving a note "this class is immutable" in an API documentation may prevent the other developers in a team from making the class mutable, or the fact that another class implements the *Composite* pattern may direct the developer to implement a new operation by delegating it to a collection of components, which should be present in the *Composite* class.

An information about implemented pattern can also be helpful when a new developer in the team tries to understand the overall architecture of the software project. Some design patterns usually represent an infrastructural detail rather than a domain specific code. On the other hand, if we also consider the patterns used in the Domain-driven-design approach, these are mainly represented by domain specific classes, which are important for an overall picture of the architecture.

Unfortunately documentation in natural language is not understandable for software, but some kind of standardized documentation of design patterns implementation would be. The main conception behind the Patterns4Net is that developers will annotate their code using .NET attributes mechanism and the Patterns4Net will provide tools that will take advantage of this documentation and will support the development process.

Patterns4Net provides two main tools. Pattern Enforcer verifies some of the structural aspects of selected design patterns implementation and Architecture Explorer generates interactive UML-like class diagrams from .NET assemblies. This tool uses the information about design patterns implementations to generate more abstract and high-level diagrams than standard UML reverse engineering tools.

## 1.2 Report structure.

In the following chapter we discuss design patterns support in development process in general. For precise patterns support and reasoning about patterns, it is crucial to have a formal definitions of patterns and so we explore patterns formalization techniques in the section 2.1. The role of programming languages in the design patterns implementation is discussed in the section 2.2. Reasons for tools support in the design patterns based development and limitations of existing tools form the content of the section 2.3

Third and fourth chapter focus on concrete tools developed as a part of Patterns4Net. Namely Pattern Enforcer in third chapter and Architecture Explorer in fourth chapter. In both cases we firstly present features of the selected tool, then a few use-cases, architecture of the tool and finally comparison or related work.

Description of the Graphviz4Net a graph visualization tool for .NET, which was developed for Architecture Explorer, is provided in the fifth chapter.

In the conclusion we summarize the report and suggest future work.

# 2. Design patterns support

Even though design patterns cannot be completely implemented as reusable libraries, there is room for some automation which can be handled by the software to overcome some of the disadvantages of design patterns. Software tools may enhance the implementation of design patterns on the source code level, for example, by code generation or refactoring. On the higher level of abstraction, during the modeling of class diagrams in UML, tools may direct a designer to introduce suitable patterns in the design. Verification of patterns implementation on either the source code level or the higher level object design (like UML class diagrams) could be useful for discovering software bugs and could prevent from communication errors, when, for instance, one of the team members is used to use a little bit different variation of some design pattern than the others in the team.

## 2.1 Patterns formalization and verification

Design patterns used to be described only in an informal manner in natural language using graphical diagrams, usually complemented with code examples. This representation, useful for human beings, is not suitable for a rigorous reasoning (e.g., for formal verification) and obstructs any automation tools support. The need for a formal specification of design patterns is obvious. In this section we discuss the patterns formalization techniques. It is important to note that formalization of patterns is not intended to replace the informally written pattern catalogs, which are ideal for learning purposes.

A pattern in object-oriented design consists of several elements. In the introduction we mention a description, a solution and consequences. These parts could also be broken down into smaller pieces. The solution part can decomposed to a structural aspect and a behavioral aspect. In this section we focus on the structural aspect of the solution part.

The solution part of a design pattern always contains some degree of flexibility. In the introduction we provide an example of the *Composite* pattern and we explain that the composite operation getWidth may be implemented as the mean of all widths or the width of the largest children component. This kind of flexibility is what makes the *Composite* pattern a pattern and not an adept for an aspect or a generated class using meta programming [1]. Some authors assume that the *Composite* pattern should always have the methods for adding and removing components [6]. The "children related operations" are indeed mentioned in [2], but does it mean that a *Composite* class must always be mutable (allow to change it's children collection)? Does it mean that immutable quasi *Composite* class that does not allow adding or removing children after it's creation does not implement the *Composite* pattern, even though it clearly solves the problem solved by the *Composite* pattern and it does it in very similar way? These questions might be another evidence for the need for a precise pattern formalization. But the approach should balance the degree of formalization and the degree of flexibility.

[1] Aspects and meta programming with connection to patterns are discussed in section 2.2

We think that during the process of actual formalization of concrete patterns the informal description should not be translated literally, because otherwise, the formal verification would be unnecessarily strict and thus would go against the flexibility developers expect from design patterns.

Another thing to note is that some patterns are different in the problem part, but their solution parts are almost the same. The *Bridge* and the *Adapter* patterns differ only in the intent: the *Bridge* is used during the design phase, but the *Adapter* is used to wire up already existing classes.

### Structural formalization

Most of the design patterns solutions involve more cooperating classes or objects. The term participant or the term role is used frequently. In the *Composite* pattern solution we have a *Composite* class, *Leaf* objects and a *Component*, the base interface for *Composite* and *Leafs*. This implies that if we have a set of real classes and we choose one to play the role of the *Component* and one to play the role of the *Composite*, the *Component* class must inherit from *Composite* class (or implement *Composite* interface in the languages such as C#), otherwise it is not a correct implementation of the *Composite* pattern. This is a simple example of a structural aspect of the *Composite* pattern solution, whose formalization could be rather straightforward. The same holds for the fact that in a valid implementation of this pattern the *Composite* class should aggregate a collection or a list of *Components*.

The *Composite* pattern solution also guides us to implement the operations on *Composite* class by delegation to the *Components*. This is more complicated to formalize since a delegation to the *Components* can have several different forms. The special case might be a situation when the composite operation returns a cached value, which is refreshed after each addition or removal of a child. Besides this very special case we could say that the composite operations should iterate over the *Components* collection. Could we also say that a composite operation should always call the corresponding operation on each of the *Components*? It all depends on the degree of flexibility we want to have in our formalization. Most of the approaches presented in [6] are relatively strict. On the other hand in Patterns4Net we went for more flexible formal specifications of patterns solutions and we verify only the core aspects, which should be fulfilled almost always. Thus the users of Patterns4Net can still take the advantage of some flexibility in design patterns implementations.

In the previous two paragraphs we rather informally describe how the formalization of patterns structural aspects could work. To make the approach of formalization complete we need some instrument to precisely capture the rules which should be fulfilled by the structure of a correctly implemented pattern. The existing formalization techniques are usually based on mathematical formalisms. For example, the Balanced pattern specification language (BPSL, [7]) leverages the first-order logic, because the relations between pattern roles can be easily expressed as predicates. In BSPL a pattern is specified using the first-order language called $S_{BSPL}$, where variable and constant symbols represent classes, typed variables and methods, sets of these are designated C, V and M. $S_{BSPL}$ provides predicates (BSPL authors use the term relation) such as $Invocation(m_1, m_2)$

where $m_1, m_2 \in M$, which evaluates to true iff[2] method $m_1$ invokes method $m_2$. The structural specification of the *Observer* pattern in $S_{BSPL}$ is given in the figure 1. The English names of the predicates are self-describing.

$$
\begin{aligned}
&\exists subject, concrete_subject, observer, concrete_observer \in C;\\
&\exists subject_state, observer_state \in V;\\
&\exists attach, detach, notify, get_state, set_state, update \in M:\\
&Defined_in(subject_state, concrete_subject) \land\\
&Defined_in(observer_state, concrete_observer) \land\\
&Defined_in(attach, subject) \land Defined_in(detach, subject) \land\\
&Defined_in(notify, subject) \land Defined_in(set_state, concrete_subject) \land\\
&Defined_in(get_state, concrete_subject) \land Defined_in(update, observer) \land\\
&Reference_to_one(concrete_observer, concrete_subject) \land\\
&Reference_to_many(subject, observer) \land\\
&Inheritance(concrete_subject, subject) \land Inheritance(concrete_observer, observer) \land\\
&Invocation(set_state, notify) \land Invocation(notify, update) \land\\
&Invocation(update, get_state) \land Argument(observer, attach) \land\\
&Argument(observer, detach) \land Argument(subject, update)
\end{aligned}
$$

Figure 1: The structural specification of the *Observer* pattern in $S_{BSPL}$

To employ such formalization in a practical use for a verification or a recognition, we need to evaluate the predicates according to a source code or another representation of an object oriented program. An interesting proposal is discussed by the authors of SPINE [8]. They suggest to use Prolog language. We can represent the constraints for pattern structure as Prolog rules and those rules that depend on source code analysis (e.g., *Invocation*) can be added to the Prolog program database using `assert` or removed using `retract`. The SPINE language they present is based on Prolog and it comes with HEDGEHOG which is a proof engine that parses Java programs, adds the corresponding rules to the database and then is able to answer questions such as standard Prolog program; for example, whether specific class implements the *Singleton* pattern or whether a class that implements the *Composite* pattern exists in the database. The figure 2 shows structural specification of a variant of the *Singleton* pattern in SPINE and a Java class that implements the *Singleton* pattern according to this specification.

A promising approach might be to express patterns as stereotypes in UML and use the Object Constraint Language (OCL, [9]) to express the stereotype constraints. UML, as a part of the Model Driven Architecture (MDA, [10]), is widely used technology and so the OCL, also part of the MDA, might become popular and widely used as well in the future.

Another approach leverages the semantic Web technologies [11]. Design patterns can be defined as RDF documents instantiating a vocabulary based on the Web ontology language (OWL). This approach also promotes the usage of design

[2] if and only if

```
realises('PublicSingleton',[C]) :-
  exists(constructorsOf(C),true),
  forAll(constructorsOf(C), Cn.isPrivate(Cn)),
  exists(fieldsOf(C),F.and([
    isStatic(F),
    isPublic(F),
    isFinal(F),
    typeOf(F,C),
    nonNull(F)
  ])).

public class PublicSingleton {
  public static final PublicSingleton
    instance = new PublicSingleton();
  private PublicSingleton() {}
}
```

Figure 2: The *Singleton* pattern in SPINE and Java.

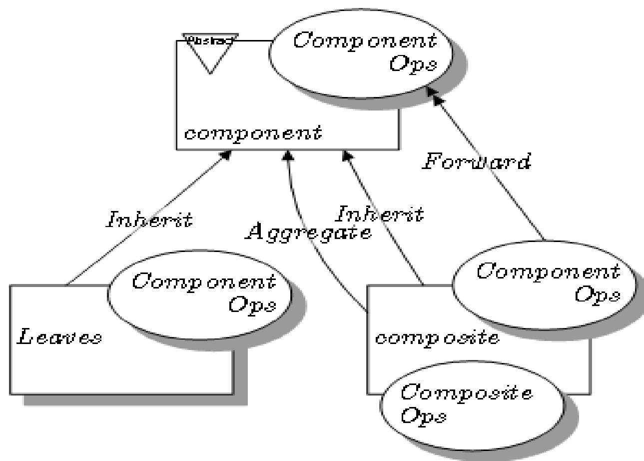



Figure 3: Specification of the *Composite* pattern in LePUS3.

patterns as a knowledge shared among software developers. Lastly the LePUS3 [12], which is one of the most accepted and well known approaches for design patterns formalization, provides graphical notation for expressing the structural aspects of design patterns. The figure 3 shows specification of the *Composite* pattern in the LePUS3. It can be seen that graphical notation provides more lucid (in comparison to textual forms) form of specifying the structural aspects.

## 2.2 The role of programming language

The choice of a programming language determines what can and what cannot be implemented easily. In [2] the authors assume Smalltalk/C++-level language features. If they assumed procedural languages, they might introduce patterns such as "Inheritance" or "Encapsulation". But there are also important differences between object-oriented languages. For example Groovy [13] supports the multiple dispatch, which lessens the need for the *Visitor* pattern.

### 2.2.1 New features

Since the first publication of [2], the mainstream programming languages went through an evolution. For example, lambda expressions are supported in C# since version 3.0, new versions of PHP, Java and new C++ specification all include lambda functions. This is four of the top five most popular languages [14], therefore we can say that nowadays lambda functions can be considered as an essential feature. Let us investigate what lambda functions may bring for design patterns. In most design patterns solutions a polymorphism is used to "inject" some logic that will be implemented later on and can be swapped for another. This brings flexibility to the design.

The *Template method* pattern might be a good example. It defines a skeleton of an algorithm and one or more steps of the algorithm might be altered in the subclasses by overriding virtual methods. When we have lambda functions we may use them to achieve the similar flexibility. The new version of the *Template method* pattern with lambda functions does not invoke virtual methods, but the lambda functions that are passed to it as parameters, or that are given to the object as constructor parameters. This alternative imlementation does not require to create a new class each time we want to implement a new set of steps that alternate the algorithm. On the other hand we have to provide the lambda functions and if they are long enough we may still end up with refactoring them to methods and these methods to a new class. A good compromise might be to implement the *Template method* as usual, but provide a subclass that invokes lambda functions, which it gets as constructor parameters. The method ForEach( Action) from the List class from the .NET Base Class Library might be considered as a simple example of such *Template method* with lambda function.

Another well known example of features that ease the *Observer* design pattern implementation are delegates and events in C#. They are supported in C# because the *Observer* pattern is suitable for event driven applications, which are developed in the .NET quiet often. Java and Swing, on the other side, use the *Command* pattern to invoke some code in response to a GUI event. Implementing each event response as a separate class (command) is not so tedious in Java as it would be in C#, because Java supports anonymous classes, which are missing from C#.

Some modern languages support advanced and innovative features in the field of object oriented programming, which could ease the implementation of design patterns even more. Such features include aspects or meta-programming. Special tools can also enable these in C# or Java. The implementation of some of the design patterns in ApectJ, aspect oriented extension for Java, is presented in [15]. [16] presents Ruby's advanced features (e.g., meta programming) that simplifies the implementation of design patterns.

### 2.2.2 New design problems

The new features of programming languages bring us also new design problems. An example might be the pattern from the .NET that we call *Flexible Generic Interface*, which leverages .NET feature of an explicit interface implementation to overcome the problem of implementing non-generic interfaces (e.g., IEnumerable) and their generic counterparts (e.g., IEnumerable<T>) in a one class.

```
interface IGraph
{
  IEnumerable Vertices { get; }
  IEdge { get; set; }
}
```

Figure 4: Non-generic IGraph interface

```
interface IGraph<T>
{
  IEnumerable<T> Vertices { get; }
  IEdge<T> { get; set; }
}
```

Figure 5: Generic IGraph interface

Imagine that we implement a graph library in .NET. We have the IGraph interface aggregating edges and vertices and we want it to be flexible, so we don't restrict the vertices to be objects of any specific type. This way users can create a graph containing integers and strings at the same time. The example of such interface is in the figure 4.

Now users can add whatever they want as a vertex, but some users may want to work with just a one type of vertices. In the case of the non-generic IGraph they would have to cast all the objects they get from the IGraph interface. An unnecessary casting was one of the reasons for introducing the generics into the .NET runtime. The generic version of the IGraph interface might look like in the figure 5.

If we had these two interfaces separately, then we would have to implement each graph algorithm in two versions – one for the generic, and the other for the non-generic version of the IGraph interface. This is far from ideal, so what we do is to let the generic interface implement the non-generic one. Because the names of the members are the same, we have to use the new keyword as in the figure 6.

How shall we now implement the new interface? That is where the explicit interface implementation comes to play. The generic methods are implemented as usual, but the non-generic methods are implemented using this feature. We can see a fragment of the implementation in figure 7. A consequence of this implementation is that we have to explicitly cast the class to non-generic IGraph if we want to work with the non-generic versions of the methods, but normally we don't want to do so, unless we pass our graph as a parameter to an algorithm

```
interface IGraph<T> : IGraph
{
  new IEnumerable<T> Vertices { get; }
  new IEdge<T> { get; set; }
}
```

Figure 6: Generic IGraph interface implements the non-generic one.

```
class Graph<T> : IGraph<T> {
  private IList<T> vertices = new List<T>();
  public IEnumerable<T> Vertices {
    get { return this.vertices; }
  }
  IEnumerable IGraph.Vertices {
    get { return (IEnumerable)this.vertices; }
  }
  // ...
}
```

Figure 7: Fragment of implementation of the generic IGraph interface.

which works only with the non-generic version of the interface, but in such case case, the cast is done automatically.

### 2.2.3 Future trends

The mainstream programming languages are usually not adopting new features because their authors want to ease the implementation of design patterns, but they normally adopt features that help to solve more general problems such as lambda functions. These features might make some design patterns obsolete or almost disappear, for instance, the *Observer* pattern in .NET, but they also might bring new challenges. The authors of the mainstream programming languages, usually corporations or standardization committees, don't want to add features that will be only useful in some rare situations, because then the language would be overcomplicated. Some patterns (e.g., the *Flyweight* pattern) are too complex, abstract or not so widely used, so that direct support for their implementation in languages such as Java or C# is not likely in the near future.

## 2.3 Tools support

Some of the complexity of implementing and maintaining design patterns is caused by an introduction of new classes and new methods into the design. During the development, the connection between the classes and the concrete design patterns roles might be lost and the system that used to be well designed, perfectly lucid and easily extensible becomes the exact opposite. Design patterns might provide an abstraction that helps to develop large software systems. For example, when a specific class hierarchy does not change often, but the operations over these classes are being constantly added or removed, then the *Visitor* pattern is suitable. Developers who already know the *Visitor* pattern don't need to study how the double dispatch in this pattern is implemented. If they want to add a new operation and they have the information that these classes can be visited by an instance of the *Visitor*, the task becomes easy. However, if the intent to use the *Visitor* pattern is not clear, some developers might start adding new operations directly to the classes that can be visited and the system becomes inconsistent. One could conclude that, if the system had been designed without

the *Visitor* pattern from the first moment, it could paradoxically be better.

The reasons for possible misunderstanding when design patterns are involved in the code might be either a complete lack of a documentation, or either an inaccuracy of a textual documentation in natural language. One possibility to overcome these problems might be a standardized documentation of design patterns instances. For the Java platform there is a project called JPatterns ([17]), which provides the annotations to mark patterns in Java code. At the moment it only provides a javadoc documentation for the annotations and the annotations itself. We are not aware of any similar approach for the .NET platform.

The standardized documentation won't prevent developers from violating the principles of implemented design patterns, although it could help a lot with this problem. To take even more advantage of the documentation, a verification tool that would enforce some aspects of design patterns could be implemented. Such tool might, for example, prevent a developer from direct communication with an object of specific type, when this communication should in fact be done through the *Mediator* object.

Moreover, during the process of implementing a pattern, the abstract idea behind the pattern is broken down into several classes or methods. With a standardized documentation we could reconstruct it back a thus provide a more abstract view on the software system.

While tools for formal verification and tools for reconstructing abstract design patterns from a set of concrete classes exist, most of them do not leverage standardized documentation of design patterns that is located directly in the code and direct location of documentation in the source code may motivate developers to keep it up to date. Moreover, except for few of these tools, they didn't get enough attention from the industry and most of them target the Java platform, but only few target the .NET platform.

Some of the reasons why the industry is not adopting the design patterns verification tools may be too much mathematical formalism involved in their usage. For definition of new patterns, knowledge of formal logic is usually required.

Tools that reconstruct abstract design patterns from a set of concrete classes are mostly based on an automatic recognition of design patterns, whose advantage is that it does not require additional work from developers and can be used for legacy systems, but it's disadvantage is that it cannot correctly recognize all the design patterns, since some of them don't differ in the solution part, or are too much abstract to berecognized only from the source code.

### 2.3.1 Patterns4Net

In order to evaluate the ideas stated in the previous paragraphs, we implemented a prototype project called Patterns4Net, which is a set of tools that support the development of an object oriented software on the .NET platform. These tools take advantage of a special documentation about patterns solution participants (in the following text referred as "patterns meta-data"), which is usually expressed using custom .NET attributes provided by Patterns4Net (in the following text referred as "Patterns4Net attributes"), but this mechanism is extensible and the patterns meta-data may be discovered using, for example, naming conventions, or anything else that can be inferred from CIL meta-data. There are predefined pat-

terns in the standard distribution of Patterns4Net, but users can add their own patterns. Patterns4Net consists of Pattern Enforcer and Architecture Explorer tools.

Pattern Enforcer checks marked pattern implementations in .NET assemblies against constraints written in C# using special API. Users can add constraints for their custom patterns or even just idioms or simple conventions such as "all methods in domain classes should invoke Logger.Log method".

Architecture Explorer leverages the patterns documentation to generate UML-like class diagrams that support a notion of zooming in and out which adds or removes details from the diagram. Such way a developer can have a general overview of the architecture or he can zoom to a specific class and see all related classes. The decision whether class should be displayed in the general overview or whether it should be displayed only in the highest zoom is based on the patterns roles it implements. Some patterns represents rather an infrastructural detail, on the other hand, for instance, patterns from [5] or from [4] are usually represented by domain specific classes.

# 3. Pattern Enforcer

Pattern Enforcer is a tool that verifies selected structural aspects of design patterns. There are several built-in patterns, but users can also add their custom patterns using the special API. Pattern Enforcer needs to know which classes are supposed to implement which pattern in order to enforce its correct implementation. For this purpose the Patterns4Net special documentation for pattern solution participants is used.

This chapter starts with the explanation of Patterns4Net – the common infrastructure for Pattern Enforcer and Architecture Explorer. Patterns4Net handles discovering of the patterns meta-data and representation of design patterns instances. Then the description of Pattern Enforcer tool itself follows.

## 3.1 Design patterns documentation

### 3.1.1 Terminology

Design pattern is an abstract entity, which, among other things, primarily describes a solution to a recurring problem. The description of the solution usually contains a certain degree of flexibility. If such design pattern is implemented by a developer, he transforms the abstract ideas behind the pattern into a real source code. For example, participants of the *Composite* pattern are transformed into the concrete classes in a real source code, or a method for the creation of a *Product* object, described in the *Factory Method* pattern, is implemented by a concrete method. When the participants of a certain pattern are implemented by the concrete elements in a source code, we will say that these elements form an instance of the pattern. For better illustration, an instance of the *Composite* pattern is given in the figure 8. From the structural point of view the *Composite* pattern has two roles: the *Composite* class (in this particular instance represented by the WidgetComposition class) and the *Component* interface (in this case, the IWidget interface), which should by implemented by the *Composite* class.

```
public class WidgetComposition : IWidget
{
  private IList<IWidget> children;
  public int Width {
    get { return children.Sum(x => x.Width); }
  }
}
```

Figure 8: Example of the *Composite* pattern instance.

### 3.1.2 Documentation of pattern instances

If we consider the example from the figure 8, Pattern Enforcer doesn't know that the WidgetComposition class should implement the *Composite* pattern, and therefore Pattern Enforcer doesn't know that it should enforce the structural aspects

of the correct implementation of the *Composite* pattern on the WidgetComposition class. For this purpose we need to create a mapping between concrete elements in a source code and the pattern participants they are supposed to implement. We call this mapping as a design patterns participants mapping.

Since Patterns4Net and therefore Pattern Enforcer works with .NET assemblies the design patterns participants mapping can be created from any data included in .NET assemblies. It can be constructed from naming conventions, for example, classes whose name starts with "Null" can be said to be implementation of the *Null Object* pattern; or the mapping can be constructed from special meta-data added into the .NET assembly by it's author in order to explicitly document his intentions to implement such and such pattern. We call this kind of meta-data as patterns meta-data.

Patterns4Net provides an extensible mechanism for construction of design patterns participants mapping from .NET assemblies data. At the moment it supports pattern meta-data expressed as .NET attributes. Class that plays the main role in particular design pattern implementation is decorated with a special attribute and references to classes that implement another roles in the pattern solution, if there are any, can be inserted as arguments for this attribute. For better illustration, a code example is provided in the figure 9. Here the WidgetComposition class is decorated with the Composite attribute, which also allows us to provide a *Component* type as a constructor parameter. Explicit specification of a *Component* type is required when a *Composite* class implements more than one interface, otherwise the *Component* type can be inferred automatically.

```
using Patterns4Net.Attributes;
[Composite(typeof(IWidget))]
public class WidgetComposition : IWidget, ICloneable
{
  private IList<IWidget> children;
  public int Width {
    get { return children.Sum(x => x.Width); }
  }
  // ...
}
```

Figure 9: An example of attributes driven documentation of pattern instances.

### 3.1.3 Pattern instances representation

In order to use the design patterns participants mapping, Patterns4Net needs to have data structures that represent the mapping. For this purpose standard C# classes are used and instances of these classes represent the instances of design patterns. An object that represents a pattern instance provides a name of the pattern, and the references to the elements that participate in this pattern instance. For example, the object that represents the *Composite* pattern instance from the figure 9 returns the string "Composite" as the pattern name and it provides references to the WidgetComposition as the *Composite* class role and to

the IWidget interface as the *Component* interface. Each pattern has a specific set of participants and therefore for each pattern, there is a specific class.

The classes for the pattern representation provide the name of the pattern and the references to the pattern instance participants as standard .NET properties. The references to code elements (that is classes, interfaces, methods, etc.) are represented by instances of Mono Cecil's classes, which are similar to the System .Type type from the standard library (e.g. TypeReference, [18]). Mono Cecil's types are used, because we use Mono Cecil for parsing of .NET assemblies[1]. The figure 10 demonstrates an example of the *Composite* pattern definition.

```
public class Composite : IPattern
{
  public TypeDefinition Composite { get; set; }
  public TypeReference Component { get; set; }
  // The Name is required by IPatter interface
  public string Name {
    get { return "Composite"; }
  }
}
```

Figure 10: The *Composite* pattern definition for Patterns4Net.

## 3.2 Patterns structural constraints specification

There are two possibilities to capture the structural constraints of a particular pattern that should be verified by Pattern Enforcer. The constraints can be hard-coded in the Patter Enforcer itself, or they can be located in external files and expressed in a special language, which would ease the addition of constraints for new design patterns.

We used a compromise approach in Pattern Enforcer. We developed a special C# API for the specification of the structural aspects of design patterns, therefore the specification itself is expressed in a standard C# (or any other .NET language) code, but the author of the specification is provided with a set of classes and methods that ease this task. The code that expresses the specification can be then loaded into Pattern Enforcer at runtime using the standard .NET mechanisms designated for this purposes. When we made this decision, we had considered several important consequences: the authors of the specification will be able to use the provided API or, if the API is not sufficient for their purposes, they can take the advantage of the full power of C#. We didn't have to develop a parser for a special language; and, since the users of Pattern Enforcer are .NET programmers, they will learn the C# API with less effort than a new syntax of a special language.

In the rest of this section we describe the API for the patterns structural constraints specification in more detail. Because this API can be considered as an example of so called Domain Specific Language (DSL, [19]) and because it also

[1] Reasons why we have chosen Mono Cecil and more detailed information about it are presented in the subsection 3.4.1

uses a technique called Fluent API, we discuss these two concepts in the following subsection.

### 3.2.1 Domain specific languages

Domain Specific Language (DSL) is a computer programming language of limited expressiveness focused on a particular domain. There are two types of DSLs: internal and external. The external DSLs are completely new languages with their own custom syntax, while the internal DSLs are embedded into existing general purpose language such as C#, Java or Ruby by providing specific public API. When developing an embedded DSL, a programmer do not have to create a parser for his DSL, but he can be limited by the syntax of the "hosting" language.

In the connection with the embedded DSLs, the term type-safe DSL is often used. Type-safe DSLs use constructs that can be verified by a compiler rather than strings with a special internal syntax that can be verified only during the runtime or by an additional tool. For example, NHibernate ORM framework ([20]) has such API for a definition of objects to database schema mapping. Instead of expressing the names of properties as strings, NHibernate exploits the C#'s feature of lambda expressions for this purpose, and thus the existence of the properties used in the mapping is verified by the C# compiler. For a better idea of this approach, figure 11 shows a short example of the NHibernate DSL usage in C#. Note that all text in this figure forms a perfectly valid C# code, although it may seem as a special language.

```
var mapper = new ModelMapper();
mapper.Class<RegisteredUser>(mapping =>
{
    mapping.Id(x => x.Id, map => map.Column("MyClassId"));
    mapping.Property(x => x.Username, map => map.Length(150));
});
```

Figure 11: Example of type safe DSL embedded into the C# language.

The usage of type safe DSLs also enables integrated development environments support. Namely intellisense support can make the development more effective and can bring a better experience for developers who don't know the DSL syntax yet, because they can see all the possibilities in the intellisense window together with their API documentation. On the other hand, after every change, the code has to be recompiled and the assembly must be deployed, which is not always possible. An xml based configuration or an external DSL might provide more flexible solution in such case.

Embedded DSLs usually leverage a technique called Fluent API, which means that a method returns an object on which a user is expected to invoke another method. This chaining of methods can make the API more self describing, because methods names and their arguments names can be then read almost as an English sentence. An example of the Fluent API from jMock, a mock object library for Java [21], is show in figure 12.

```
mock.expects(once())
  .method("m")
  .with(stringContains("hello"));
```

Figure 12: An example of methods chaining in Fluent API.

### 3.2.2 The API for pattern constraints specification

Because we have a strongly typed representation of design patterns instances, we can build a type safe DSL for their constraints specification, where we will use lambda expressions in a similar way as the authors of NHibernate.

In our conception, a constraint is any boolean function that takes a pattern instance as a parameter and returns a boolean value, which indicates whether the pattern instance conforms to the constraint or not. However, Pattern Enforcer provides a DSL to make the specification of these constraints easier than that. The key part is that it enables to specify the constraints as lambda functions. We call such function a "check".

A check may be performed on the whole pattern instance, then the parameter of the lambda function will be the object representing the pattern. These checks may verify the relations between roles, for example, that the *Composite* class implements the *Component* interface. Users can also set up checks only for a specific role of a pattern instance. In such case, the Pattern Enforcer API provides a method to select the specific property of the pattern instance object with a lambda function the same way NHibernate uses lambda functions for selecting properties. After the property is selected, the user can create a check only for the value of the selected property (that is for a particular role). Finally the user can also select specific methods of the selected role to provide a check for each of them. The selection of these methods is also done using a lambda filter function.

To summarize it all up: users can select a subject of the check, using lambda functions, and then they can enter the check itself again as a lambda function, which takes the subject of the check as a parameter. For a better idea, an example is shown in figure 13.

```
// we want to work with the Composite role
this.Type(composite => composite.Composite)
// we want to check all its non-private methods
.Methods(method => method.IsPublic || method.IsProtected)
// on each of them, we perform the following check
.Check((composite, method) =>
      method.CallsAnyOf(pattern.Component),
  (composite, method) => "An error in " + method.Name));
```

Figure 13: An example of constraints configuration in Pattern Enforcer.

A check expression might be anything, which enables wide range of possibilities for experienced users, but Pattern Enforcer provides an easy to use extensions to underlying Mono Cecil's API. CallsAnyOf is an example of such extension, which returns true iff the method invokes a member of given class. Basically

these extensions are designed to enable straightforward specification of most of the predicates presented in the section 2.1.

### 3.2.3 Built-in patterns

As we claim in the section 2.1 constraints for the built-in patterns were chosen rather less restrictively than in the other tools of this type. The aim was to enforce those aspects that are strongly significant to given pattern and the implementation without them cannot be clearly called as an implementation of this pattern. For example, the *Factory Method* pattern, whose main participant is the *Factory Method* itself, would make no sense if the actual *Factory Method* was void. On the other hand, to enforce that the method's body contains only a constructor invocation and a return statement, seems to us as an inappropriate restriction, because the developer might want to prepare some data structures before returning the *Product* of the *Factory Method*. A list of patterns supported by Pattern Enforcer can be found in the appendix **??**.

The relatively unrestrictive API for patterns constraints specification allows us to provide more advanced verification than only verification of structural aspects. This is the case of the *Immutable* pattern. The verification of it's implementation checks that the *Immutable* class does not allow to change the internal state of it's instance once it is available to the "outside world". What does this bring us? A simple immutability in C# can be enforced by specifying all the class's fields as readonly, but this disables the creator of the class to provide a *Simple Factory Method* that would do some changes to the *Immutable* class instance before the method returns it to the "outside world". Also auto-implemented properties, which bring a notable simplification of implementation of simple properties, cannot be specified as readonly.

We will illustrate the process of choosing the structural constraints that should be verified by Pattern Enforcer on the example of the *Template Method* pattern and the *Adapter* pattern.

#### Template Method

The main role of the *Template Method* pattern is a *template method*, which defines the skeleton of an algorithm. The *template method* invokes one or more virtual methods, which are expected to implement certain steps of the algorithm. Because these methods are virtual, one can override them in a sub-class and thus alter some steps of the algorithm without the need to write the whole algorithm from scratch.

The core of the *Template Method* pattern are invocations of virtual methods that can alter the algorithm. From a first look, one could say we should enforce that the *template method* invokes at least one virtual method. However, a *template method* that invokes another non-virtual method that then invokes another virtual method can be considered as an implementation of the *Template Method* pattern as well, because it also allows us to alter the algorithm in sub-classes. We can recursively check all methods that are invoked from our *template method*, but it would be unsystematic. Instead, a simple observation can help: non-virtual

methods that invoke virtual methods are usually also implementation of the *Template Method* pattern. So the conclusion is that a *template method* should invoke at least one virtual method or at least one another *template method*.

It is considered a good practice with the *Template Method* pattern to declare the *template method* as non-virtual (sealed) and so we enforce this too.

The specification of constraints for the *Template Method* pattern is shown in figure 14 (we stripped out the full error messages from it). As a first step we check that the type that declares the *template method* is not sealed and therefore it can be sub-classed. If this is fulfilled, we check that the template method calls at least one virtual method or another template method.

```
this.Type(pattern => pattern.TargetMethod.DeclaringType)
  .Check(type => type.IsSealed == false,
         (pattern, type) => "...error message...");

// check that template method invokes at least
// one virtual method or another template method:
this.If(pattern => !pattern.TargetMethod.DeclaringType.
    IsSealed)
  .Method(pattern => pattern.TargetMethod)
  .Check(
    method =>
      method.GetMethodCalls().Any(
        call =>
          call.TargetObject != null &&
          call.TargetObject.IsThisParameter &&
          (IsTemplateMethod(call.Method) ||
           call.Method.Resolve().IsOverriddeable())),
    (pattern, method) => "...error message...");
```

Figure 14: The specification of the built-in *Template Method* pattern.

### Adapter

The *Adapter* pattern solves the problem of two incompatible interfaces: the *Target* interface and the *Adaptee* interface. It suggest to create an *Adapter* class that uses the *Adaptee* interface and adapts it to the *Target* interface. The most usual implementation of this pattern is done by implementing the *Target* interface by the *Adapter* class, which holds an instance of the *Adaptee* interface as a private field and delegates operations to it.

Only in rare cases the *Adapter* class just delegates all it's operations to the *Adaptee*. Usually it is needed to convert either parameters values or a return method value. The *Target* interface might not have the same number of methods as the *Adaptee*, therefore some *Target*'s methods might be delegated to more than one *Adaptee*'s method; or a *Target*'s method might not have a corresponding method in the *Adaptee*, so it is performed by the *Adapter* class without any help from the *Adaptee*.

It would be very difficult or almost impossible to capture some useful constraints about methods delegation in this case. The *Adapter* pattern is one of the patterns whose solution contains a certain degree of flexibility. One possibility is to enforce that the *Adapter* class delegates at least one operation to the *Adaptee*, but when the *Adapter* class has a field of the *Adaptee* type and does not use it at all, the programmer will be anyway warned by the compiler. Therefore the only two constrains that seems to us as useful, but still does allow the flexibility in the implementation of the *Adapter* pattern, are: the *Adapter* class should have at least one field of the *Adaptee* type, and the *Adapter* class should implement or inherit the *Target* interface.

Another special case appears when the implementation of the *Adapter* pattern does not adapt an interface of a class, but, for example, a procedural interface or an interface to a device. In such case we cannot enforce even existence of the field of the correct type. Similarly when the *Target* interface is not standard .NET interface, we cannot enforce that the *Adapter* class implements it. Therefore, we allow users to omit the *Target* or *Adaptee* types. If they do so, nothing is enforced, but the documentation of patterns participants is still useful for Architecture Explorer.

The final specification of constraints for the *Adapter* pattern is given in the figure 15.

```
   .Check(pattern => pattern.WrapperType.Implements(pattern.
       Target),
            pattern => "...error...");

this.If(pattern => pattern.WrappedType != null)
   .Check(HasFieldOfType,
            pattern => "...error...");

// ...
static bool HasFieldOfType(Adapter pattern)
{
   return pattern.WrapperType.HasFields &&
      pattern.WrapperType.Fields.Any(x =>
           x.FieldType.IsEqual(pattern.WrappedType));
}
```

Figure 15: The specification of the built-in *Adapter* pattern.

## 3.3 Usage

If a user wants to take advantage of Pattern Enforcer, one possible way to achieve it is to decorate his types with pattern attributes. For this, it is required to add a reference to the *Patterns4Net.Attributes.dll* assembly in the project. This assembly contains only the attributes definitions, thus it's footprint should be minimal. It is built for .NET version 2.0, so Pattern Enforcer can be basically used in projects built for older versions of the .NET. When the reference is added, the

types can be decorated with patterns attributes from the namespace Patterns4Net .Attributes.

```
using Patterns4Net.Attributes;
[Composite(typeof(IWidget))]
public class WidgetComposition : IWidget, ICloneable
{
  private IList<IWidget> children;
  int IWidget.Width {
    get { return 10; }
  }
  // ...
}
```

Figure 16: An example of an annotated implementation of the *Composite* pattern.

The figure 16 contains an annotated implementation of the *Composite* pattern, which is not valid, because the getter method of the Width property is not using the children collection.

Pattern Enforcer can be run outside the Visual Studio or inside the Visual Studio. When the Visual Studio project is built, there should be a resulting assembly in the output folder (usually {project folder}\bin\Debug). Say it's name is *EnforcerExample.dll*. Then if the *pattern-enforcer.exe* is run from the command line supplied with a path to *EnforcerExample.dll* as an argument, it should produce the output shown in the figure 17. Pattern Enforcer supports three output formats: plain text, xml and special format for Visual Studio. The output format could be specified with a command line option. When Pattern Enforcer is run without arguments, it displays the help.

Besides the direct execution of *pattern-enforcer.exe*, Pattern Enforcer can be integrated more tightly into the build process in Visual Studio. Visual Studio project files are basically MSBuild scripts, so the only thing a user has to do is to add a reference to Pattern Enforcer MSBuild task and invoke it in the After-Build target, which, as its name indicates, gets always executed after the source code is built. To enable this integration, it is needed to to open the project file *EnforcerExample.csproj* in any text editor, find the xml root element Project and just below it, insert a UsingTask tag, where the location of the *PatternEnforcer.MSBuildTask.dll* assembly should be specified. Next, the AfterBuild target should be located (it should be commented out and placed near the end of the

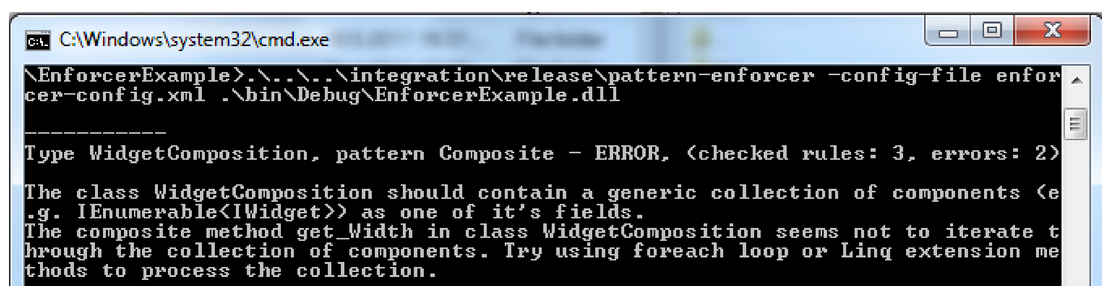



Figure 17: The output of Pattern Enforcer for the WidgetComposition class.

file), it should be uncommented, and an invocation of the Pattern Enforcer task should be inserted as it's child element. The figure 18 shows the xml code fragment and also a screenshot of Visual Studio displaying the warnings.

```
<Project ToolsVersion="4.0" DefaultTargets="..." xmlns="...">
  <!-- Includes the PatternEnforcerTask -->
  <UsingTask TaskName="PatternEnforcerTask"
     AssemblyFile=".\a\path\to\PatternEnforcer.MSBuildTask.dll"
        />

  <!-- ... -->
  <PropertyGroup>
    <PatternAssemblies>
      $(OutputPath)/$(AssemblyName).dll;
    </PatternAssemblies>
  </PropertyGroup>
  <Target Name="AfterBuild">
    <PatternEnforcerTask
      ToolPath="..\a\path\to\pattern-enforcer-executable\"
      ConfigFile=".\enforcer-config.xml"
      ShowErrorsAsWarnings="true"
      InputAssemblies="\$(PatternAssemblies)">
    </PatternEnforcerTask>
  </Target>
</Project>
```

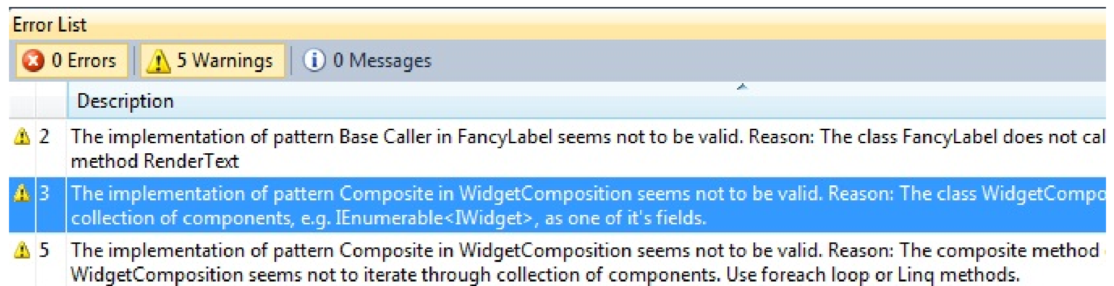



Figure 18: Integration of Pattern Enforcer and Visual Studio 2010.

Checking by Pattern Enforcer can be turned off for a specific element by annotating it with a special attribute PatternEnforcerIgnoreAttribute, which has a string property Justification, where developers should provide a description why they have disabled the checks on this class or method.

### 3.3.1 Unit tests

The second possible way of taking advantage of Pattern Enforcer does not require to annotate classes with pattern attributes. Instead the relation between a concrete pattern and it's roles is constructed by hand in an automatized test. Pattern Enforcer provides the PatternEnforcerContext class whose instance represents an assembly loaded into memory and prepared for execution of Pattern

Enforcer checks. It is recommended to set up this object in the test fixture[2] set up method, which is a method that gets executed only once before any test from the test fixture is executed. The PatternEnforcerContext provides a method AssertThat, which has one generic parameter. This method returns an object that provides methods with names Is{PatternName}, which perform the check of conformance to given pattern. The type selected as the generic argument of the call to AssertThat is used as main role of the pattern, other additional required information, if needed, are supplied as parameters of the Is{PatternName} method. The figure 19 shows an example of such test fixture using the NUnit framework ([22]).

```
public class WidgetCompositionTests : IWidget, ICloneable
{
  private PatternEnforcerContext patternEnforcer;
  [FixtureSetUp]
  public void SetUpFixture() {
    this.patternEnforcer =
      PatterEnforcerContext.Create("EnforcerExample.dll");
  }
  [Test]
  public void WidgetComposition_Is_Composite() {
    this.patternEnforcer
      .AssertThat<WidgetComposition>()
      .IsComposite(typeof(IWidget));
  }
}
```

Figure 19: Example of an automatized test that invokes Pattern Enforcer.

### 3.3.2 Specification of custom pattern

There are two possibilities to define a pattern and constraints that will be enforced on it's implementation. The first one is more complex, but provides better flexibility, and thus is used internally by Pattern Enforcer. The second one is more simple and is designed to provide an easier instrument to create user-defined patterns. The first approach is described in the section 3.4.2. Here we show how to use the second one.

We describe an example of an implementation of a simple custom pattern from a user perspective. We will call our new pattern as the *Base Caller*. It has two roles: the Target class and the Target's base class. The constraint we specify for this pattern is that the Target class is required to invoke the corresponding base methods in the bodies of overridden methods. We describe how to create a class that represents our pattern and attribute for it's documentation, then we specify the constraints for our new pattern and then we show how to load our new pattern into Pattern Enforcer.

[2]This term is used by the NUnit framework ([22]), some of other xUnit frameworks also use the term "test suite" instead of a test fixture.

### Representation of custom pattern

A custom pattern is represented by a class that implements the IPattern interface and it is also recommended to implement the IPatternAttribute interface, which is just a marker for pattern attributes. This class is used for representation of the pattern and at the same time as an attribute for annotating the pattern instances in code, therefore it should inherit from the Attribute class from standard .NET library.

The IPattern interface requires just a getter of the property named Name and a getter of the AbstractionLevel property, which is used by Architecture Explorer (for now, null can be used as a default value). The value of the Name property should be a human readable name of the pattern, which may contain any characters including spaces. The IPattern interface does not require any other properties, but the creator of the pattern should add other properties for representing the pattern participants, in this case the Target and it's base class. Because these properties should contain references to other types, they will be of type TypeDefinition. The implementation is shown in the figure 20.

```
public class BaseCaller : Attribute, IPattern,
   IPatternAttribute
{
  string IPattern.Name {
    get { return "Base Caller"; }
  }
  public TypeDefinition TargetType { get; set; }
  public TypeDefinition BaseType { get; set; }
}
```

Figure 20: The implementation of custom user-defined pattern.

### Implementation of the ***Base Caller*** attribute

During the processing of patterns attributes, Patterns4Net needs to reconstruct the BaseCaller attribute instance from CIL meta-data. The meta-data does not contain an instance of the attribute, instead it contains only values of constructor arguments used for its instantiation in the original source code and names and the values of the properties that were assigned. For example, meta-data for the standard .NET Obsolete attribute as it is used in the figure 21 contains: zero constructor arguments, because the parameterless constructor of the Obsolete attribute is used; and one property with name Message and it's value.

For the purpose of the reconstruction of pattern attributes from CIL meta-data, classes that implement both the pattern and it's attribute are required to define a constructor with one parameter of type IDictionary<string, object> [3]. The pattern attribute class should be able to reconstruct it's instance from this dictionary, which provides the CIL meta-data in the following format:

[3] More technical reasons that lead to this decision are given in section 3.4

```
[Obsolete(Message="A constant value")]
public class AnnotatedClass
{
}
```

Figure 21: A code example to illustrate CIL metadata for attributes.

- Constructor Arguments – indexed by the number of the position. For example, the first argument, if any, will be under the index "0".
- Attribute's target – a TypeReference instance that contains a reference to the type that was decorated with this attribute. This value is available under the index "–Target".
- Assigned properties – the remaining entries of the dictionary are name-value pairs representing the properties. If the original property is of type System.Type, than it's actual value in the dictionary will be TypeReference from Mono Cecil referring to the same type.

The implementation of such constructor for the BaseCaller class is shown in the figure 22. (Note: an instance of the TypeReference class from Mono Cecil can be converted to the corresponding TypeDefinition instance using method Resolve() as in the example.)

```
public class BaseCaller : Attribute, IPattern,
   IPatternAttribute
{
  public BaseCaller(IDictionary<string, object> values) {
    var targetRef = (TypeDefinition)values["-Target"];
    this.TargetType = targetRef.Resolve();
    this.BaseType = this.TargetType.BaseType.Resolve();
  }
  // ... as before
}
```

Figure 22: The implementation of a special constructor required by Patterns4Net.

### Constraints specification

The pattern attribute, as declared in figure 22, can be used for annotation of classes that implement our *Base Caller* pattern. However, to verify that such class invokes corresponding base methods in overridden methods bodies, the last two things are needed. The first one is to implement the IPatternCheckerProvider interface, defined in the assembly *PatternEnfocer.Core.dll*. This interface contains one method GetChecker, which should return a constraints checker for the pattern. The last thing needed is to create the checker itself. The FluentPatternChecker class, which implements the DSL described in the subsection 3.2.2, is intended to be the base class for pattern checkers, although a minimal pattern checker

has only to implement the IPatternChecker interface. The implementation of a checker for the *Base Caller* pattern is similar to the specification of the built-in patterns, which is described in the subsection 3.2.3. For completeness of the example, figure 23 shows the final implementation of the *Base Caller* pattern.

```
{
  // ... same as before
  public IPatternChecker GetChecker() {
    return new Checker();
  }

  private class Checker : FluentPatternChecker<BaseCaller> {
    public Checker() {
      this.Type(pattern => pattern.TargetType)
           .Methods(method => method.OverridesBaseMethod())
           .Check((pattern, method) =>
                     method.GetMethodCalls() != null &&
                     method.GetMethodCalls().Any(call =>
                        call.HasTargetObject &&
                        call.TargetObject.IsThisParameter &&
                        call.Method.DeclaringType.IsEqual(
                pattern.TargetBase) &&
                        call.Method.Name == method.Name),
                 (pattern, method) =>
                      string.Format(
                        "Method {0} does not invoke the base
                            method.",
                        method.Name));
    }
  }
}
```

Figure 23: Definition of the checker for the *Base Caller* pattern.

### Addition of the *Base Caller* to Pattern Enforcer

Finally Pattern Enforcer has to be informed that it should load the assembly that contain the custom pattern definition and search it for custom patterns definitions. For this purpose, it is required to provide the assembly location in a Pattern Enforcer configuration file. The configuration is an xml file (the pattern−enforcer−config.xsd file with definition of it's structure is supplied with Patterns4Net and included in the appendix **??**). The location of a configuration file is provided to Pattern Enforcer as a command line option, or the parameter of the MSBuild task.

## 3.4 Architecture

In this section the architecture and the implementation of the common Patterns4Net infrastructure and Pattern Enforcer is discussed. We start with CIL parsing, because the instruments we use for this task influence the rest of the system. Then we describe design patterns representation and discovery architecture in more detail than at the beginning of this chapter. Our solution to CIL analysis and overall Pattern Enforcer design are also presented. In the last subsection, we provide basic information about the development infrastructure we used for the development of Patterns4Net.

### 3.4.1 CIL processing

We have two basic options to process the source code of a .NET application or a library. The original textual source code can be parsed and represented as an abstract syntax tree (AST), or we can parse .NET assembly and use the Common Intermediate Language (CIL).

When the original source code is parsed and represented as an AST it is much easier to reconstruct higher level information such as actual parameters for a method invocation. On the other hand, available parsers not always support all of the most current language features and parsing of a source code of a specific language might restrict us to support the only one language. Some parsers are capable of parsing more source languages into the same AST structure, but the resulting AST is still different for some language specific constructs.

The other option, which we have chosen, is to analyze the intermediate language, in case of the .NET it is the Common Intermediate Language (CIL). The structure of CIL is more stable than, for example, the syntax of C#. The latest version of CIL standard [23] from 2010 has the same instruction set as the previous version from 2006. The version from 2010 only extends semantics and verification rules for some of the instructions. Another advantage is that intermediate language is produced by all the compilers for .NET, thus Patterns4Net can be theoretically used also for Visual Basic.NET, IronRuby, IronPython and others, although we have tested it only on C#. One of the disadvantages of this approach is that the CIL is stack based lower level language and the reconstruction of some constructs, such as actual parameters for a method invocation, requires special effort.

#### Library for CIL parsing

There are three popular, publicly available libraries that could be used to parse .NET assemblies and get meta-data about types and CIL code of the methods. First option is to use the reflection API that is available as a part of the .NET base libraries. Second option is the Microsoft Common Compiler Infrastructure (CCI, [24]), which is developed in Microsoft Research. Last option is Mono Cecil [18], which is developed as a part of the Mono open-source project.

Standard .NET Reflection API treats assemblies as a code, not as a raw data, which has two important consequences: the code loaded through the .NET

Reflection API can be executed; and, because the code can be executed, the run-time must check access rights and might throw Code Access Security exception. Assemblies loaded into an AppDomain (a .NET object similar to an process in a operating system) cannot be unloaded, which means that after a long session with Architecture Explorer, when user have loaded many different assemblies, the process memory usage will be unnecessary high. Finally the .NET Reflection API does not distinguish between a type definition and a type reference, which is an entry in assembly meta-data referring to a type located in another assembly. If we used the standard Reflection API, there would be one notable advantage. In the public API of Patterns4Net, in some cases, we allow to use the .NET Reflection data structures in order to make the usage of the Patterns4Net API easier for developers used to use the .NET Reflection. However, because we internally use another library, we have to do a translation of the .NET Reflection data structures.

The other two libraries (CCI and Mono Cecil) process .NET assemblies as just a binary data, hence they do not support loading the assemblies into an AppDomain and execution of the loaded code. On the other hand they are claimed by their authors to be faster than the standard Reflection API, however, we are not aware of any serious benchmarks. Public API and features of CCI and Mono Cecil seem to be similar, except CCI provides AST over the intermediate language, which Mono Cecil doesn't provide[4]. However, the AST generated by the CCI is more complex than we would need, therefore, for our purposes, the advantage of generated AST would be lessen by extra work for it's processing. Both of these two libraries have a long list of advanced software that use them. In case of Mono Cecil it is, for example, db4o (object database for java and .NET) or Mono C# compiler. On the other side, FxCop (a bug-finding tool) or Code Contracts are both based on CCI.

Our previous experiences with Mono Cecil have resolved the choice between Mono Cecil and CCI in favor of Mono Cecil. This choice does not only influence the code that does the CIL analysis, but also other code, because we use specific Mono Cecil's data structures (e.g., TypeReference) in the whole Patterns4Net project.

### 3.4.2 Patterns representation and discovery

Patterns representation is described in the section 3.1. Here we just remind that a pattern instance is represented as an object that provides references to the participants of this pattern instance. Mono Cecils structures are used for types and methods identification.

The discovery of patterns meta-data is implemented as a flexible mechanism. There is a central class, which aggregates several objects and each of them provides a strategy for creation of the pattern participants mapping based on CIL metadata. The class hierarchy is show in the figure 24.

There are two built-in strategies for the pattern participants mapping discovery. Both are based on pattern meta-data (additional information added to a .NET assembly by it's author in order to document patterns he has implemented). In both cases these meta-data are expressed as .NET attributes provided

[4]There is project Cecil Decompiler, but it is not in production ready quality.

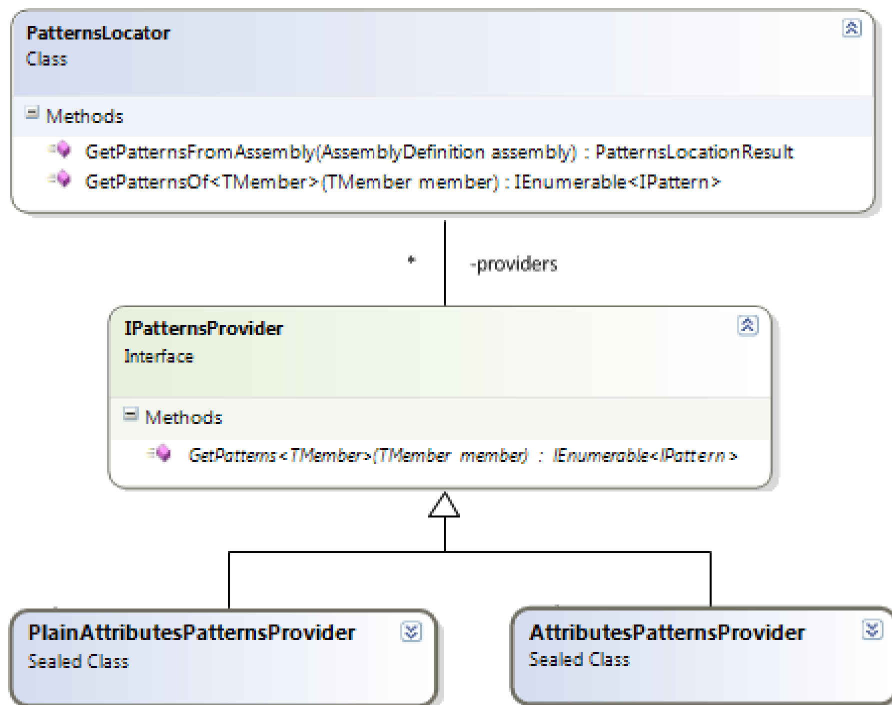



Figure 24: Hierarchy of classes that are used for discovery of patterns meta-data.

by Patterns4Net. These two strategies differ only in way they reconstruct the pattern participants mapping from attributes. Before we describe them in more detail, we discuss challenges connected with .NET attributes.

### Attributes based discovery

When a class is annotated with an attribute, this information is projected into the meta-data located in the .NET assembly. These CIL meta-data are then parsed by Mono Cecil as a raw data, so it does not contain real instances of the classes that represent attributes, rather, Mono Cecil provides information about the type of the attribute, the values provided as it's constructor parameters and provided named arguments. If a user of Mono Cecil wants to reconstruct an identical instance of the attribute, he has to do it by hand.

There are two basic options to solve the task of attributes reconstruction. The pattern attributes should always implement a constructor with name-value dictionary as a parameter. The constructor is expected to reconstruct the attribute instance from this dictionary and it is used to create attributes instances according to CIL meta-data. (This approach is also used in the MEF.) The other option is not to reconstruct the attribute instance at all, but for each attribute provide another strategy class that will create the pattern instance based on the attribute's CIL meta-data, but without instantiating the attribute.

The reconstruction of an attribute using special constructor provided by the attribute is implemented in the AttributesPatternsProvider strategy class. It provides higher cohesion, because we don't have to create another strategy class for each attribute. The direct construction of pattern instances from CIL meta-data is implemented in the PlainAttributesPatternProvider strategy class.

#### Attributes and pattern instances representation

Since we wanted the mechanism of the discovery of the pattern participants mapping to be flexible, our representation of pattern instances must be independent of whether attributes or something else is used for their documentation. Therefore the classes that represent patterns, shouldn't have to represent the attributes for their documentation. There should be possibility decouple these two things.

Because the PlainAttributesPatternProvider strategy class constructs directly the design pattern instances, without constructing the attributes instances, there doesn't have to be any coupling between the pattern class and the attribute class. However, the AttributesPatternProvider class reconstructs the attribute instance and in order to take advantage of this instance we allow such attributes to provide a method which creates the pattern instance. In this case an additional strategy class for the construction of a pattern instance from the CIL meta-data is not needed, the strategy is implemented as method of the attribute, but it also means that the pattern and it's attribute are coupled. The approach implemented by the PlainAttributesPatternProvider class is used internally by Pattern Enforcer and the second one is intended to be used for implementation of user-defined custom patterns, because is it easier to use.

### 3.4.3 CIL analysis

Mono Cecil provides only data parsed from .NET assemblies, it does not provide anything more. From CIL meta-data we can, for example, determine for a given class what type is it's base type, or which interfaces it implements. But Cecil itself does not provide a method that would give us a list of types that implement given interface, because this information cannot be inferred directly from it's meta-data. For such purposes there is the Mono Cecil Rocks project, which contains a few extension methods for the Cecil's classes, but it does not have all we wanted to support in Patterns4Net, so we also implemented our custom set of extension methods designed for CIL analysis and patterns structure constrains specification.

For example, one of the extensions we wanted to provide was uniform API for getting information about methods overrides. In CIL, according to ECMA CIL specification [23], there is an attribute "overrides" in the meta-data of every method, which is a list of methods that this method overrides. But this attribute is used only in specific cases (e.g., explicit interface implementation) and normally it is left empty, because overridden methods are determined by conventions (which are also described in the ECMA CIL specification).

#### Methods invocation analysis

For the purposes of the discovery of relationships in Architecture Explorer and methods invocations in Pattern Enforcer, we needed classes that would help us with analysis of CIL. We don't need to analyze conditional statements – we just want to know whether a method M1 on a field F is invoked in body of a method M2, even in a dead branch of code.

Method calls in CIL are done by several instructions, for example . callvirt . CIL does not distinguish between instance methods and static methods. Instance methods has the instance as a first parameter, which is normally added by a compiler. Each of these instructions has a method reference as an operand, so the only difficulty is to analyze with which actual parameters the method was invoked.

The CIL virtual machine is a stack based machine, which means that all arguments for operations are taken from the evaluation stack and results are pushed onto the stack. Usually instructions pop all their arguments from the stack and push results onto the top. Stack behavior of each instruction is documented in the ECMA CIL specification, however, Cecil provides this information through the enumeration StackBehaviour.

The CIL analysis is done by simulating the evaluation stack. In a loop we iterate over all instructions in the method body. For each instruction we determine how many items it pops from the stack and we determine, which items it pushes onto the stack. The stack is represented as a collection of instances of the StackItem class. Each StackItem has a reference to the instruction that resulted in pushing this item onto the stack, and with this basic information the StackItem can provide some more additional information such as whether it represents a field pushed onto the stack (if so, then which field), or a parameter aso. The result of this analysis is a collection of the StackState class instances – n-th of them represents the state of the stack after the execution of n-th instruction in the method body. State of the stack is represented as a collection of StackItem instances. From the signature of the method we know how many parameters it has (we will designate it as $m$) and whether it is an instance method or a static method. To get the actual parameters of a specific call instruction (say it's $n$-th instruction), we just need to take $m$ (or $m + 1$ for instance methods, which have implicit first parameter) items from the top of the $n - 1$-th StackState.

The last question may be whether this correctly simulates the stack if we do not take the control flow instructions into account (only their stack behavior). The answer is provided by ECMA CIL specification, which reads

> Regardless of the control flow that allows execution to arrive there, each slot on the stack shall have the same data type at any given point within the method body.

### CIL instructions sequences matching

In order to check some more specific constraints such as the specification for the *Singleton* pattern implementation, we need to check whether a method body contains a specific CIL instructions sequence.

The aim here was to be able to match sequence which, for example, contains anything at the beginning and then it contains a sequence of instructions that represents an if with a specific condition. For this purpose the matching process is directed by one object that delegates its work to several strategy objects that do the actual matching. In our example, we would have a strategy that would match any instruction and a strategy that would match the instructions sequence that represents an if.

The main class for CIL instructions sequences matching is the CILPatternsMatcher . It aggregates a collection of instances of the InstructionMatcher abstract class, which represents an instructions sequence. Interface of the InstructionMatcher class is shown in figure 25. The Matches method is called in a loop provided with current instruction. If the method returns false, than the CIL instructions does not match the expected sequence and the whole process ends with a negative result. Otherwise property Found is checked and if true, then the next InstructionMatcher is used in the next iteration, if it was the last InstructionMatcher , then process ends with success. In the next iteration current instruction is set to the one returned by last call of Match. A pseudo code is given in the figure 26, variable matchers represents an array of the InstructionMatcher class instances.

```
public abstract class InstructionMatcher
{
  public virtual bool Found { get; protected set; }
  public abstract bool Matches(
    Instruction instruction,
    out Instruction next);
  public virtual void Reset() { ... }
}
```

Figure 25: The InstructionMatcher abstract class interface.

1: *currentInstruction* ← first instruction of the method's body.
2: $currentMatcherIdx \leftarrow 0$
3: **loop**
4: $\quad matcher \leftarrow matchers[currentMatcherIds]$
5: $\quad match \leftarrow matcher.Match(currentInstruction, out\ next)$
6: $\quad$ **if** *not match* **then**
7: $\quad\quad$ **return** $false$
8: $\quad$ **end if**
9: $\quad$ **if** $matcher.Found$ **then**
10: $\quad\quad$ **if** $++currentMatcherIdx == matchers.Length$ **then**
11: $\quad\quad\quad$ **return** *true*
12: $\quad\quad$ **end if**
13: $\quad$ **end if**
14: $\quad currentInstruction \leftarrow next$
15: **end loop**

Figure 26: Pseudo code of CIL instructions patterns matching.

### 3.4.4 Pattern Enforcer Design

The basic interface in the Pattern Enforcer design is the generic IChecker<T> interface, which defines one method Check(T). This method returns the result of the check encapsulated in an instance of the CheckerResult class. Concrete checkers have the generic parameter T set to the TypeDefinition class, the MethodDefinition class or the IPattern interface.

An important class is the FluentPatternChecker. It is a base class of most of the pattern checkers, because it provides the structural constraints specification API. The API is formed by protected methods of the FluentPatternChecker class, which should be invoked in the constructor of a derived class in order to specify the constraints. Because we wanted to provide a Fluent interface (chaining of method calls), we have to remember the parameters of chained methods, so the last method in the chain can access all of them and perform the desired action. For this purpose each chained method returns special objects that have

- a reference to the parent object (an object on which the method was invoked) and
- the values of the arguments the method was invoked with.

For the first method in a chain the parent object is the checker instance itself, for others, it is a object returned by their ancestor method in the chain. As the method calls are chaining the arguments are collected and the last method in the chain, which is the first method that wil actually do something, has all the arguments of previous methods available to do it's work. The example of such chaining of methods is given in the figure 27.

```
this.Type(pattern => pattern.PatternRoleType)
  .Methods(method => method.IsPublic || method.IsProtected)
  .Check((pattern, method) => method.Name.EndsWith("42"),
    (pattern, method) => "Method has not a valid name");
```

Figure 27: An example of methods chaining.

The whole Pattern Enforcer is encapsulated as the PatternEnforcer class. It requires a PatternsLocator instance and an array of IPatternCheckersLocator instances.

### 3.4.5 Development infrastructure

In this section about Pattern Enforcer architecture, we also describe common infrastructure used by both Pattern Enforcer and Architecture Explorer tools. Patterns4Net tools are developed in the .NET platform version 4.0, mostly in the C# 4.0 language. Xml technologies are also used. All xml formats have their corresponding xsd file.

#### Visual Studio solution layout

In the figure 28 we can see the layout of the Visual Studio solution used for Patterns4Net development. When we refer to the projects from this solution in the following text, we will omit the *Patterns4Net* prefix.

The projects that start with "PaternEnforcer" are related to Pattern Enforcer tool. The classes that provide the core functionality of Pattern Enforcer and the classes that form the unit-testing public API are located in the *PatternEnforcer.Core* project. The output of the *PatternEnforcer.Cmd* project is a command line interface for Pattern Enforcer and the project *PatternEnforcer.MSBuildTask*

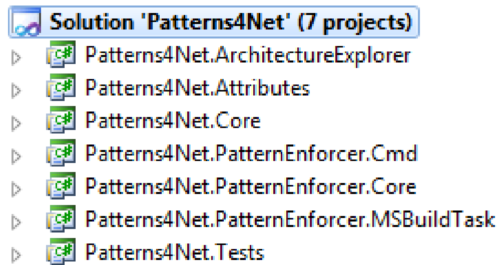



Figure 28: The layout of Visual Studio solution.

is the implementation of the task for the MSBuild engine. The core functionality and the unit-testing API of Pattern Enforcer are decoupled from the command line interface and the MSBuild task into separate project, and thus separate assembly, because the Pattern Enforcer core functionality is also used in the *ArchitectureExplorer* project. The resulting assembly with the Pattern Enforcer core functionality is also meant to be referenced by users in their unit-testing projects and if it had an .exe suffix, although perfectly valid assembly that can be referenced, an unusual suffix might confuse some users.

The *ArchitectureExplorer* project contains the code of Architecture Explorer GUI tool. The GUI is done in Windows Presentation Foundation (WPF) framework. In this project, besides C# classes, also XAML[5] files are included.

Automated tests are used during the development of Patterns4Net. These tests are located in the *Tests* project. This project aggregates tests for the classes from all the other projects, because we don't need to separate the tests into several projects and a lower number of projects speeds up the build process.

The classes that deal with the discovery of patterns meta-data and the patterns representation are located in the *Core* project. Although we use only Patterns4Net attributes at the moment, process of discovering patterns meta-data for classes and methods is extensible.

Finally, the *Patterns* project contains only the classes that represent the Patterns4Net attributes. These classes inherit from System.Attribute class and they are very simple, hence we don't necessarily need to take advantage of the advanced features of C# 4.0. This enables us to set the target framework version to 2.0, which means that the resulting assembly can be referenced and used by older .NET projects as well.

### General principles

**Automated tests.** Every software should be tested. Besides manual testing, usually a time consuming task, there is also the possibility to automate some tests, which means that their execution is controlled by a software and the software reports eventual errors. Execution of such tests lasts in seconds, so they can be executed quite often. Some of the code of Patterns4Net is tested this way. For automated tests the NUnit framework [22] is used.

[5]Extensible Application Markup Language

**Extensibility.** For better support of extensibility the Managed Extensibility Framework (MEF, [25]) is used. Most of the classes in Patterns4Net get their dependencies as constructor arguments so the composition of all the objects by hand would be tedious. Instead the composition capabilities of the MEF are employed. A type whose instance should be used anywhere the interface IFace is expected must be decorated with attribute [Export(typeof(IFace))], the dependencies of such type will be resolved by MEF recursively. Because assemblies with custom patterns loaded by Pattern Enforcer are also added to the MEF composition process, users can add custom plugins, for example, custom implementation of IPatternsProvider. However, this experimental feature was not extensively tested and is not further documented.

**Code Contracts.** Most of the classes define their contracts using Microsoft Code Contracts [26]. Contracts help us to find the issues earlier. An exception is typically thrown during the pre-conditions check, which is the real cause of the problem, rather than later on the source code line that expects valid input parameters. Code Contracts also serve as a complement to the API documentation.

**Design Patterns.** Patterns4Net can be considered as a first example of it's own usage, because Patterns4Net code is annotated with patterns attributes.

## 3.5 Comparison

There are several existing tools that provide verification of design patterns implementation. The most similar approach to Pattern Enforcer is the Pattern Enforcing Compiler (PEC) for Java.

### 3.5.1 FxCop and Gendarme tools.

It may not be obvious, but Pattern Enforcer is similar to static analysis bug-hunting tools such as FxCop [27] or Gendarme [28]. These tools search the source code for the idioms that are generally considered as bad. For example strings should be, in most cases, compared using string.CompareOrdinal, but not using == operator. There are two main differences between Pattern Enforcer and these tools

- Pattern Enforcer checks only the code that is annotated,
- Pattern Enforcer checks structural aspects and code idioms, but Gendarme and FxCop check only code idioms.
- Gendarme and FxCop are looking for bad idioms, but Pattern Enforcer verifies that expected idiom is present.

Gendarme is open-source tool that is meant to be an alternative to FxCop. It uses Mono Cecil for CIL analysis. If we look at Gendarme's source code, it has a similar structure to Pattern Enforcer's code. It has also "checker" classes, that perform checks on a code element, which might be, for example, Cecil's TypeDefinition.

### 3.5.2 Pattern Enforcing Compiler (PEC) for Java

Because the Pattern Enforcing Compiler (PEC) for Java [29] has been the main source of an inspiration for this work, we discuss it a little bit more in detail.

PEC for Java is an extended Java compiler that formalizes patterns. Developers can use standard Java syntax to annotate their classes as an implementation of specific design pattern. The PEC then checks whether the classes actually implement the specified patterns.

**Annotation.** For annotation of patterns instances, PEC uses so called marker interfaces. The authors have chosen this technique, because implemented interfaces are listed in generated API documentation and so an integration with an API documentation didn't require any additional work to be done. On contrary, interfaces can only be used for annotation of classes, but not methods, and even when interfaces can have arguments – generic arguments –, these can capture only a limited number of additional information. The authors of PEC admit these weaknesses of interfaces as a technique for the annotation of patterns and in [29], they propose to introduce the standard java annotations, similar to .NET attributes, in PEC. However, we are not aware of any updated version of PEC that uses standard Java annotations.

**Pattern Enforcement.** PEC uses static analysis and it also enforces the rules dynamically by inserting assertions into the resulting program, which we don't support in our Pattern Enforcer. Dynamical enforcement provides more accurate results, because, for example, uniqueness of the one *Singleton* instance cannot be proved statically, but one simple assertion in it's constructor is enough to enforce it dynamically. The disadvantage of dynamical enforcement is that it slows down the resulting program and the program still has to be manually tested in order to discover possible bugs.

**Code generation.** PEC provides also code generation capabilities. For example, a body of a void method in a *Composite* class can be generated by PEC – it just creates a loop over all components and on each of them it invokes the corresponding method. However, an implementation of the *Composite* pattern is usually not so straightforward, so these capabilities turn to be not so useful.

**Patterns specification.** Two APIs for patterns specification are supported in PEC. It is the standard Java reflection API and Javaasist, which is similar to Mono Cecil. In Pattern Enforcer we support the standard .NET reflection only partly. In PEC a method that checks a pattern implementation must be a static method with specific signature declared in a marker interface of the pattern. This introduces coupling between a pattern annotation and a pattern enforcement, which we tried to avoid in Patterns4Net. PEC uses exceptions to to signal the errors during the check of pattern implementation. This means that usually when a first violation is found, an exception is thrown and the verification process does not continue. In our Pattern Enforcer we use a return value of the special type CheckerResult as the result of the check and this object can aggregate more errors.

**Integration with development environment.** The authors of PEC claim that it is an extended compiler, which means that Java source code is compiled only with PEC, although PEC internally uses javac. This provides a seamless integration with the Java environment, but at the same moment PEC becomes an essential requirement for successful build. Our Pattern Enforcer is standalone tool, which can be easily taken out from the build process. PEC does not provide any other usage possibilities, but our Pattern Enforcer has the MSBuild task and the unit-testing API.

### 3.5.3 Other tools

CoffeeStrainer [30] is a tool that is somewhere between static analysis bug-hunting tools whose objects of interest are idioms, smaller pieces of code, and pattern enforcement tools. Unlike other static analysis bug-hunting tools CoffeeStrainer enforces rules that result from particular design decisions, for this it provides means for custom rules specification. CoffeeStrainer targets the Java platform.

Pattern-Lint [31] can check conformance to variety of design principles from coding style rules to design patterns. Pattern-Lint targets C++ and has been successfully evaluated during development of a multimedia operating system.

Most of the approaches described in [6] are connected with some prototype tool that enforces the specification represented according to the formalization approach. However, most of them are not publicly available and all of them target either Java or C++ languages. The most interesting tools from this book include the HEDGEHOG engine, which we also discuss in the section 2.1, and tools that come with LePUS3, which we also present in the same section.

To close this section, we can conclude that we are not aware of any design patterns verification tool for the .NET platform. Pattern Enforcer is, among all of these tools, also extraordinary with it's special C# API for structural constraints specification, because most of the other approaches uses special language for patterns formalization, or, in case of the Pattern Enforcing Compiler for Java, they don't provide special means for structural constraints specification at all.

# 4. Architecture Explorer

Architecture Explorer provides UML-like class diagrams generated from .NET assemblies. Instead of one large diagram with lots of unnecessary infrastructural classes, it uses the information about implemented design patterns to create more diagrams with different levels of abstraction. User can browse the diagrams in interactive graphical user interface.

Similarly to UML class diagrams, diagrams generated by Architecture Explorer can contain classes, interfaces and structures, which can be grouped into packages. But packages, in the case of Architecture Explorer, cannot be hierarchical, instead Architecture Explorer provides a concept of a *Layer*, which is a container for packages. Standard types of UML relationship are supported in Architecture Explorer. For elements that implement a design pattern Architecture Explorer provides documentation of the design pattern displayed in a side panel.

Data for diagrams construction are reverse engineered from .NET assemblies. In some cases Architecture Explorer uses special meta-data, for example Patterns4Net documentation of design patterns or special attributes that allows users to document other aspects such as types of relationships. Therefore we can say that Architecture Explorer performs human aided reverse engineering. Important to note is that all the documentation, which helps Architecture Explorer in reverse engineering, is located directly in the source code.

## 4.1 Features

### 4.1.1 Abstraction levels

Information about implemented patterns allows the Architecture Explorer to provide more levels of abstraction. The top level, for example, shows only high level, domain specific classes that are important for understanding of the overall architecture. However, if user "zooms" to a particular class, all related classes, even infrastructural, are displayed. The rules that defines which elements will be displayed and which not, were chosen according to our opinion on what should be considered as important and what is rather infrastructural. However, mechanism for definition of these rules is easy to change. In the following subsection we describe these rules, but before that we introduce the related terminology.

Classes that implement higher level patterns such as *Entity*, *AggregateRoot* or *ValueObject* known from Domain-Driven-Design approach are level 0 classes. Classes that implement an infrastructural pattern such as *Null Object* or *Helper Class* are level 2 classes. Other classes belong to level 1. Users can define their own patterns and assign them into any level. Levels of built-in patterns are defined in xml configuration file and can be changed by users as well.

Now we can define four levels of abstraction that Architecture Explorer provides. Some of the rules are applied only, if there are enough elements of given level, because otherwise the diagram will be unnecessarily half-empty.

- Layers level – all layers are displayed each one contains it's packages. If there is a class in a layer A which is in relation with a class in a layer B,

then the relation is displayed between the layers A and B. If there are more relations of the same type between A and B, only one is displayed.

- Layer level – if the layer contains at least as many classes of level zero as packages, then these classes are displayed. Otherwise first level classes from the layer are displayed. Classes have labels with their names, but members are not displayed. Packages are displayed as rectangles and contain corresponding classes.

- Package level – classes from the package are displayed.

  - If the package contains at least on class of level zero, then classes of level zero and one are displayed.
  - If the package does not contain any class of level zero, then classes of level one and two are displayed.

- Class level – selected class is displayed with all it's methods and properties. All classes from any package or any layer that are in any relationship with this class are displayed. Classes are gathered in rectangles that represent packages.

### 4.1.2 Further meta-information

Architecture Explorer displays relations that are part of some pattern implementation, and it also displays standard relations from object oriented design. These are inheritance, association, aggregation, composition and uses. Architecture Explorer reverse engineer these relations using source code analysis, but difference between some of them might be just semantical. For the purpose of the differentiation of these relationships Patterns4Net provides, besides attributes for patterns participants annotation, also attributes for annotating relations. Rules for reverse engineering of relations are summarized in the following listing.

- When a class A has a field of type B without any annotation, then the association from A to B is constructed. (The construction of cardinality of associations and other relationships is described below.)

- When a class A has a field of type B annotated with attribute Composition, Aggregation or Uses, then the relationship of composition, aggregation or uses from A to B is constructed.

- When at least one of class's A methods invokes the constructor of class B, has a parameter of type B or invokes a static method or property from B, then uses relation from A to B is constructed.

Uses relation from A to B is not added when the composition or aggregation relation from A to B was discovered. Default cardinality is one-to-one. If the field of a class A is of a type which is assignable to IEnumerable<B> then the type B will be used in relation, and the cardinality of the relation will be set according to the rules below.

- If the field is annotated with ManyToMany attribute then the relation's cardinality will be many-to-many.

- If the field is not annotated with ManyToMany attribute, but there is also field of type IEnumerable<A> in B, then the relation's cardinality will be many-to-many.
- Otherwise the cardinality will be one-to-many.

Architecture Explorer is not capable of discovering the uses relation if the constructor or static member invocation is hidden in reflection API calls. Uses relation is constructed even if the constructor or static member invocation is in the dead branch of code, which means that the constructor or static member will actually never be invoked. Architecture Explorer also does not check whether method's parameters are actually used inside the method's body.

## 4.2 User Interface

User interface of Architecture explorer consists of the content area in the center of the window, where the diagram is displayed, a toolbox on the top and various dockable panels. The toolbox contains buttons that serves to control the program and the panels display additional information about the diagram.

Architecture Explorer displays only one diagram at once. If an assembly is loaded, it's content is added into the current diagram. Therefore, if two assemblies are consequently loaded, the diagram will contain all elements from both of them. A user can browse through the diagram and let the Architecture Explorer to display him various levels of abstraction.

Basic input for Architecture Explorer is an assembly or set of assemblies to analyze. Instead of a assembly file, users can also choose Visual Studio C# project files or Visual Studio solutions, in this case the tool will extract information about assemblies location from these files.

Dockable side panels contain additional information. There are five of them

- Diagram Browser displays all diagram elements in the treeview.
- Pattern Documentation displays descriptions of all patterns where the currently selected class plays the main role.
- Properties panel displays information about current element. These information may also contain an API documentation if available.
- Output window displays warning and informal messages for user. These messages are generated during the loading of an assembly or when Pattern Enforcer is running.
- Errors window contains a gird that displays errors from Pattern Enforcer.

Architecture Explorer can display API documentation generated from source code. Source of this documentation is an xml file produced by C# compiler. Architecture Explorer expects the documentation file to have a default name and to be located in the same folder as the assembly.

Patterns documentation is loaded from the patterns.xml file, whose format is described in the patterns.xsd file. Patterns in the patterns.xml file are identified by

the full name of the class that represents the pattern. For pattern documentation an xml dialect based on standard .NET API documentation format is used. A documentation for a custom pattern can be added by editing this xml file.

When Pattern Enforcer is invoked from the Architecture Explorer, errors are displayed in special side panel and when a user clicks on the error, the diagram will zoom to class, where the error occurred.

Normally *Layers* correspond to each .NET assembly when reverse engineered, but users can define *Layers* on their own using assembly attribute as in the figure 29. First level namespaces in a layer are then reverse engineered as packages.

```
using Patterns4Net.Attributes;
[assembly:Layer("Layer Name", "Namespace")]
// safer definition using reflection,
// a namespace of MyType will be used.
[assembly:Layer("Layer Name", typeof(MyType))]
```

Figure 29: Definition of *Layer* using assembly attributes.

#### Definition of custom pattern

The definition of a custom pattern is described in section 3.3. The IPattern interface, which is required to be implemented by a class that represents a pattern, has the readonly AbstractionLevel property. Value of this property is used by Arcitecture Explorer to decide at which abstraction level it will display classes that play the main role in this pattern.

The class that represents a pattern might have properties that represent the pattern roles. To direct Architecture Explorer to emphasize relationships between the main role of the pattern and the other roles, the properties representing the roles might be annotated with the PatternRoleAttribute attribute. This attribute allows to define the type of the relation (composition, aggregation, ...), the abstraction level of the relation, a cardinality, and a name. If a default value is provided for any of these properties, Architecture Explorer tries to infer the value from the source code.

To inform Architecture Explorer about assemblies that contain custom patterns a xml configuration file can be used. This xml file is an extension of the format used for Pattern Enforcer only and thus can be used for both tools.

## 4.3 Architecture

Architecture Explorer is developed in Windows Presentation Foundation (WPF). There are two reasons for choice of WPF. We wanted to use so called *Model-View-ViewModel* [32] pattern and implementation of this pattern is easier in WPF than in Windows Forms. The second reason involves possible future work on Architecture Explorer. WPF applications can be, with some effort, ported to Silverlight, which can run in a Web browser and is supported also on other platforms than Windows.

A large portion of Architecture Explorer functionality is generation of "nice looking" graphs. For this purpose the *Graphviz* ([33]) tool is used, but it's adoption to WPF is not as easy as it might seem at first look, so it resulted in the introduction of a separate project called Graphviz4Net, which is discussed in the following chapter.

### 4.3.1 User interface logic

#### Model-View-ViewModel

The *Model-View-ViewModel* pattern is a variation of well known *Model-View-Controller* pattern. The *ViewModel* is an object that supplies data to be displayed in the *View* as values of regular properties and it provides actions that could be invoked from the *View* (e.g., by clicking on a button) as regular methods. The *ViewModel* encapsulates all the user interface logic, but it does not handle displaying the data and therefore it could be an instance of a plain C# class. The connection between the *ViewModel* and the *View*, which is WPF specific user control, is not handled by the objects themselves but is driven by powerful data-binding features of WPF.

To ease the implementation of the *Model-View-ViewModel* pattern even more, Caliburn.Micro framework [34] is used. It is capable of applying the *ViewModel* to *View* binding, *ViewModel* data to *View* visual elements binding and actions binding only according to naming conventions. For example, for the ShellViewModel class there is the ShellView WPF control and their binding is handled by Caliburn.Micro.

The figure 30 shows the layout of the graphical user interface of Architecture Explorer. The *ViewModel* classes are located approximately in the same place, where they will be displayed by their corresponding *View* WPF controls. The whole window is represented by the ShellViewModel class, which aggregates all the other *ViewModel* objects.

#### Communication between *ViewModel*s

The *ViewModel* objects communicate either directly, or through events. Events are represented by C# classes. When an object wants to publish an event it invokes the Publish method on the EventAggreator object, which is a singleton. This method has one argument, which is an object that represents the event and the arguments of the event. If an object wants to be notified when an event of certain type T is published, it has to implement the IHandle<T> interface and register itself to the EventAggreator object. Events mechanism is used for handling selection of current element and navigation in diagram, because these actions might be invoked from several panels and might result into an update of several GUI elements.

#### Documentation processing

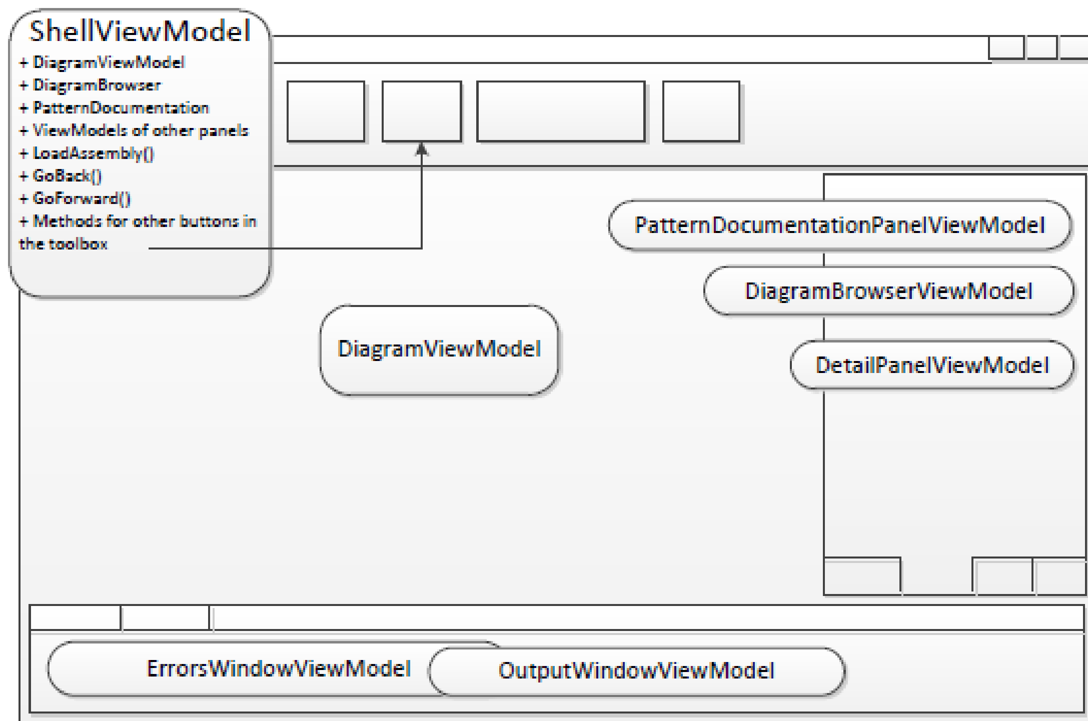



Figure 30: Decomposition of the user interface into several ViewModel classes.

An interesting aspect of the implementation might be displaying of API documentation from source code and patterns documentation from special xml file. The format of patterns documentation was chosen to be compatible with the standard format of the API documentation. The fact that in both cases the format is an xml dialect allows us to create a XSL template that transforms the xml representation into the XAML, which is then parsed and displayed by WPF.

### 4.3.2 Diagram Classes Design

A diagram is represented by a hierarchy of classes that is depicted in the figure 31. Each of these classes inherits from the base class DiagramElement, which means that each instance of these classes have a reference to it's parent object. The top level Diagram object returns a reference to itself as the value of this property.

Each type of relationships is represented as a separate class, because, in WPF, the graphical templates for objects are selected according to the object's type. This way we can have several separate templates for each type of the relationship.

Because the diagram class structure is not likely to change often, the *Visitor* pattern is used to add new operations of the diagram elements. The base class for all *Visitors* is the DiagramVisitor class. The *Visitor*'s traversing algorithm is implemented in the diagram elements Visit method.

When an assembly should be loaded into the diagram, an instance of CecilDiagramLoader class loads all the layers, packages, types, methods and properties, but it does not add any additional information such as implemented patterns or relations between them. These additional information should be added by instances of IDiagramUpdater interface, whose method UpdateDiagram is always invoked when a new assembly is added to the diagram. Most of the classes that implement the

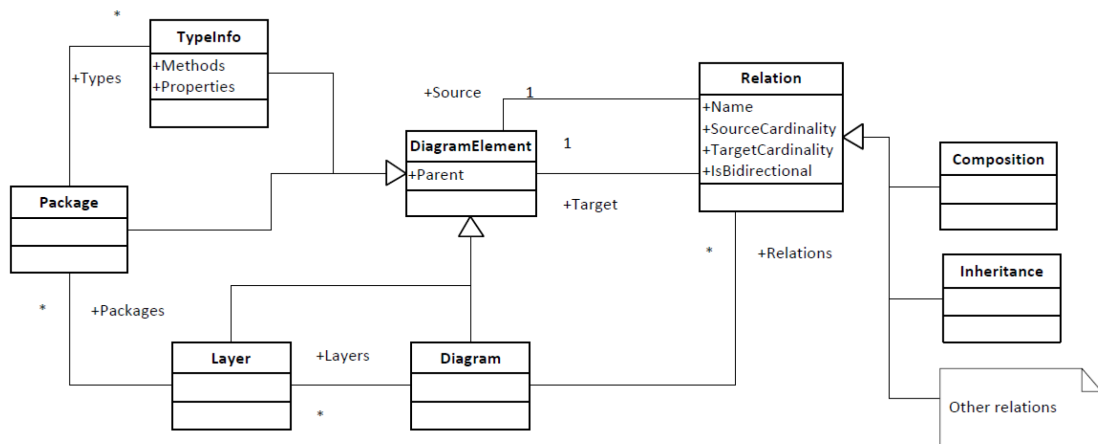



Figure 31: Hierarchy of classes that represent a diagram.

IDiagramUpdater interface are also diagram visitors and the implementation of the UpdateDiagram method is just call to the diagram's Visit method supplying itself as an argument. These "diagram updaters" implement discovery of relations and implemented design patterns.

All the objects are, as in the case of Pattern Enforcer, composed together using the Managed Extensibility Framework (MEF).

## 4.4 Related Work

According to [35] only a few approaches to reverse engineering that use additional information provided by developers exist. We are not aware of any reverse engineering tool that explicitly supports design patterns and use additional information provided by human beings. Tools that support UML standard might be, however, extended with stereotypes that could express implemented design patterns.

To provide more views on the same system each of them with different level of abstraction is the main idea behind the Model Driven Development (MDA). In contrast to Architecture Explorer, MDA does not only address design patterns but also platform independence, transformation from higher abstract models to more specific models or source code, and other issues. MDA is a standard maintained by Object Management Group and this standard has to be implemented by concrete tools.

### 4.4.1 Pattern recognition tools.

A tool presented in [36] provides design patterns instances recognition based on static and dynamic analysis. It might be interesting in the context of Patterns4Net, because it one of the few design patterns tools that targets the .NET platform. Authors also process the intermediate language, but they use standard .NET reflection.

The idea that information about implemented design patterns might help to provide several views on the same system but with different level of abstraction is also discussed in [37]. The authors propose an Eclipse plug-in called MARPLE

(Metrics and Architecture Reconstruction Plug-in for Eclipse), which could automatically recognize design patterns in Java code and then display special diagrams. The authors of MARPLE also plan to take advantage of Graphviz – a graph visualization tool.

### 4.4.2 UML reverse engineering.

The software called UMLGraph ([38]) provides automated drawing of UML diagrams extracted from java source code. It uses Graphviz for visualization and call graph analysis for discovery of relationships in similar way we do in Architecture Explorer. UMLGraph uses Graphviz directly to generate SVG images. On contrary, in Architecture Explorer we process the output of Graphviz and convert it to WPF controls in order to provide interactivity in the user interface.

# 5. Graphviz4Net

Graphviz [33] is an open-source graph visualization tool, which we use in Patters4Net to create class diagrams. Graphviz is implemented as typical UNIX filter [39]. For graphs representation, Graphviz uses special language called DOT. An example of DOT file is given in figure **??**. Graphviz expects a graph in the DOT language on the input, it generates the layout for given graph and then it renders it in a selected image format on standard output, or it can print the same graph in the DOT language on the standard output, but with attributes that provide information about the generated layout. Output format can be set up by command line option. Examples of the output are shown in the figure 33.

```
digraph G {
  node [shape=rect];
  node1 [label="A", width=2, height=1];
  node2 [label="B"];
  node3; node5;
  node1 -> node2 [label="Edge from A to B"];
  node3 -> node1; node3 -> node5;
  node1 -> node5; node5 -> node2;
}
```

Figure 32: An example of DOT file.

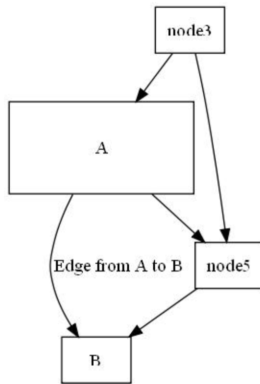
To employ Graphviz in visualization of class diagrams, we have to convert our internal representation into the DOT language, then we have to parse the output of Graphviz and finally do some coordinates transformation and scaling to convert the layout in Graphviz representation to the WPF coordinates system.

During the development, it turned out that this process can be modularized and we can segregate an independent library that provides .NET interface to the Graphviz filter and means to use the layout information generated by Graphviz for generating layouts in WPF or other GUI framework. Such library might be helpful for other projects than Patterns4Net. This resulted into the separate library called Graphviz4Net.

With Graphviz4Net users can define a graph and then display it into WPF application, or provide custom *Layout Builder* for other GUI framework (e.g., Windows Forms). Graphviz (and thus Graphviz4Net) is capable of rendering graph clusters, curved edges with labels and arrows on both sides (arrows can have also labels) and much more. With built-in WPF *Layout Builder*, graph node can any WPF control and even edges and labels rendering can be customized, although not as much as rendering of nodes.

## 5.1 Public API

Public API that Graphviz4Net provides can be divided up into two parts: graphs representation (input for Graphviz) and layout processing (output of Graphviz).




```
digraph G {
  node [label="\N", shape=rect];
  graph [bb="0,0,199,294"];
  node1 [label=A, width=2, height=1, pos="72,184"];
  node2 [label=B, width="0.75", height="0.5", pos="68,18"];
  node3 [width="0.75", height="0.5", pos="141,276"];
  node5 [width="0.75", height="0.5", pos="172,92"];
  node1 -> node2 [label="Edge from A to B", pos="..."];
  node3 -> node1 [pos="e,99.022,220.03 ..."];
  node3 -> node5 [pos="e,169.66,110.29 ..."];
  node1 -> node5 [pos="e,152.4,110.03 ..."];
  node5 -> node2 [pos="e,93.539,36.172 ..."];
}
```

Figure 33: Graphviz output.

### 5.1.1 Graph representation

Conversion of graphs into the input format of Graphviz works with interfaces IGraph, ISubGraph and IEdge, nodes may be of any type. However, for convenient use Graphviz4Net offers generic versions of these interfaces and classes that implement them. So the user of Graphviz4Net may: implement his own structures, he just have to make them implement interfaces mentioned above; or he may use the predefined generic classes.

A graph aggregates list of it's nodes and sub-graphs, which aggregate list of their own nodes. Edges are aggregated by a graph object, but not by subgraphs, because edges may cross sub-graph boundaries.

User may add custom attributes to the resulting DOT graph representation. The only thing which is needed for this is that the element (node, edge or subgraph) implements the IAttributed interface, which defines one property Attributes – a name-value collection of DOT attributes. Default graph structures supplied with Graphviz4Net implement this interface and have properties for setting and getting the usual DOT attributes such as Label. These properties provide type-safe access to the Attributes collection, which can also be modified by hand in non type-safe manner to set up less usual DOT attributes.

### 5.1.2 Layout builder

When the graph is processed by Graphviz and the output is parsed by Graphviz4Net, we need to convert the layout data to actual elements on the screen or in the generated picture. For this purpose the *Builder* pattern is employed. Graphviz4Net takes care of parsing the output, but when it has a piece of layout information for example "the position of the node with id 2 is [34, 55]", it passes this piece of information to the appropriate method of the *Layout Builder* and this method may then create an element on the screen or anything else.

Next to the graphical elements building, the *Layout Builder* is also responsible for suppling the sizes of the graph nodes, so that Graphviz can produce precise layout where nodes and edges do not overlap.

Graphviz4Net has one built-in *Layout Builder* for WPF applications (we suppose that it could build Siverlight layouts as well, however it hasn't been tested yet). Users even don't have to directly use this *Layout Builder*, the whole process of layouting is encapsulated in the GraphLayout WPF control. The only thing needed is to set up the dependency property Graph and provide data templates for nodes types.

## 5.2 Architecture

### 5.2.1 DOT parsing.

One of the tasks Graphviz4Net has to deal with is parsing the Graphviz output, which is a text in the DOT language. We developed a parser based on ANTLR parser generator ([40]) that is able to parse most of the the DOT language constructs that Graphviz produces as an output (it is a subset of the full DOT language, because we know that some DOT constructs, e.g., comments are never produced by the Graphviz).

Graphviz provides also plain-text output format, which is line oriented language suitable for parsing. However, we found out that this format does not support some features of Graphviz that we wanted to support in our library (e.g., node clusters).

### 5.2.2 WPF Support

Graphviz generates layout information in format where lengths are in inches, coordinates are in points (1/72 of an inch) and refer to the center of the element, the origin $[0, 0]$ is in the bottom left corner, coordinate values increase up and to the right and curved edges are represented as B-spline points. All these pieces have to be adopted to the WPF formats where e. g. positions can refer to one of the corners of the element, but not to it's center. There are two possible approaches for this adoption: convert all the values; or make use of render transformations in WPF to overcome the problem of different coordinate systems, but other values would still have to be converted. In Graphviz4Net we went with the first option, because the render transformations might slow down the application and there is not much difference between the two approaches in the amount of work.

The main work of the WPF *Layout Builder* is to adopt the values from Graphviz to WPF format. It gets a Canvas instance as a constructor parameter and it places all the elements into this Canvas using Canvas dependency properties Top and Left. The decision which WPF elements should be used for the graphical representation of each of the elements in the graph is leaved to an *Abstract Factory* object, which is also a parameter of the constructor of the WPF *Layout Builder*. In the default implementation of the factory for nodes we just create ContentPresenter with Content set to the node type. This enables users of Graphviz4Net to define a data template for each type of a node (remember that nodes may be of any type, so a data structure with complex information or just simple string for label may be used).

The WPF *Layout Builder* should also provide the sizes of the nodes for Graphviz. For this purpose it uses the WPF layout system. Every FrameworkElement has a method Measure(availableSize), in which the FrameworkElement should determine it's size requirements by using an availableSize parameter. For the availableSize we use double.Infinity and thus allow the element to set up any size. Desired size of the element is then accessible via property DesiredSize and the transformed value of this property is given to Graphviz.

Graphviz4Net provides also a WPF control that encapsulates this logic. The control uses standard WPF mechanisms of dependency properties and templates.

# 6. Conclusion

The aim of this work was to explore existing approaches for design patterns support in development environments and to present the Patterns4Net project. Most of the existing tools for support of design patterns enabled development target Java platform or C++, but Patterns4Net provides this form of support for the .NET platform. With Patterns4Net users can explicitly document their intent to implement a particular design pattern. Pattern Enforcer, part of the Patterns4Net project, is able to verify some of the structural aspects of selected design patterns. The second tool included in Patterns4Net is Architecture Explorer, which provides interactive UML-like class diagrams that emphasize implemented design patterns.

Patterns4Net might enhance the development process of complex design patterns oriented systems that are created by a larger team, because it helps to discover communication errors and violations of design patterns implementations earlier and it provides visual tool to tackle some of the design complexity that is caused by design patterns usage.

During the development and testing of Architecture Explorer, it turned out that rules for hiding and displaying various elements in the diagram in order to provide better abstraction are crucial for the appropriate user experience. These rules should be reevaluated after more extensive testing on real projects. Architecture Explorer user interface could be also enhanced to provide more additional information, for example, for every relationship it could show a panel with detailed information on which code fragments were lead to establishing this relationship during the reverse engineering phase.

Some of the more general rules from Pattern Enforcer, such as immutability check, could be extracted from it's source and proposed to open-source community as additional rules for well-established open-source project Gendarme, which is an extensible rule-based tool used to find problematic code in .NET assemblies.

Software systems are getting larger and more complex and this trend will continue. Changes in requirements are usual and reusability is important. Design patterns provide widely accepted approach for tackling the complexity of large systems and with tools such as Patterns4Net we can get even more advantages from their usage.

# Bibliography

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A pattern language.* Oxford Univ. Pr., 1977.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Addison-wesley Reading, MA, 1995.

[3] P. Hruby, J. Kiehn, and C. Scheller, *Model-driven design using business patterns.* Springer-Verlag, 2006.

[4] M. Fowler, *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[5] E. Evans, *Domain-driven design: tackling complexity in the heart of software.* Longman, 2004.

[6] T. Taibi, *Design patterns formalization techniques.* Igi Global, 2007.

[7] T. Taibi, *Design patterns formalization techniques*, ch. An Integrated Approach to Design Patterns Formalization. Igi Global, 2007.

[8] A. Blewitt, *Design patterns formalization techniques*, ch. Spine: Language for Pattern Verification. Igi Global, 2007.

[9] J. Warmer and A. Kleppe, "The object constraint language: precise modeling with uml," *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA*, p. 112, 1998.

[10] A. Kleppe, J. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[11] J. Dietrich and C. Elgar, *Design patterns formalization techniques*, ch. An Ontology Based Representation of Software Design Patterns. Igi Global, 2007.

[12] J. Nicholson, E. Gasparis, A. Eden, and R. Kazman, "Automated Verification of Design Patterns with LePUS3," in *Methods Symposium*, p. 76, Citeseer, 2009.

[13] "Groovy: an agile dynamic language for the java platform." `http://groovy.codehaus.org/`, May 2011.

[14] "Tiobe programming community index for april 2011." `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, may 2011.

[15] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *ACM Sigplan Notices*, vol. 37, pp. 161–173, ACM, 2002.

[16] T. Østerlie, "Ruby," *Linux Journal*, vol. 2002, no. 95, p. 4, 2002.

[17] "Jpatterns: Java design patterns." http://www.jpatterns.org/, May 2011.

[18] "Cecil – mono." http://www.mono-project.com/Cecil, May 2011.

[19] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[20] "Nhibernate forge." http://nhforge.org, May 2011.

[21] "jmock - an expressive mock object library for java." http://www.jmock.org/, May 2011.

[22] "Nunit - home." http://www.nunit.org/, Aug. 2010.

[23] T. Ecma, "Tg3. common language infrastructure (cli). standard ecma-335," 2010.

[24] "Common compiler infrastructure: Metadata api." http://ccimetadata.codeplex.com/, May 2011.

[25] "Managed extensibility framework." http://mef.codeplex.com/, Aug. 2010.

[26] "Contracts - microsoft research." http://research.microsoft.com/en-us/projects/contracts/, Aug. 2010.

[27] "Fxcop." http://msdn.microsoft.com/en-us/library/bb429476.aspx, Aug. 2010.

[28] "Gendarme – mono." http://www.mono-project.com/Gendarme, May 2011.

[29] H. Lovatt, A. Sloane, and D. Verity, "A pattern enforcing compiler (pec) for java: A practical way to formally specify patterns," 2007.

[30] B. Bokowski, "Coffeestrainer: statically-checked constraints on the definition and use of types in java," in *Software Engineering—ESEC/FSE'99*, pp. 355–374, Springer, 1999.

[31] M. Sefika, A. Sane, and R. Campbell, "Monitoring compliance of a software system with its high-level design models," in *Proceedings of the 18th international conference on Software engineering*, pp. 387–396, IEEE Computer Society, 1996.

[32] J. Smith, "Wpf apps with the model-view-viewmodel design pattern." http://msdn.microsoft.com/en-us/magazine/dd419663.aspx, Aug. 2010.

[33] "Graphviz - graph visualization software." http://www.graphviz.org/, Aug. 2010.

[34] "Caliburn micro: A micro-framework for wpf, silverlight and wp7." http://caliburnmicro.codeplex.com/, Aug. 2010.

[35] A. Br
"uhlmann, T. Gîrba, O. Greevy, and O. Nierstrasz, "Enriching reverse engineering with annotations," *Model Driven Engineering Languages and Systems*, pp. 660–674, 2010.

[36] L. Majtas, "Contribution to the creation and recognition of the design patterns instances," *Information Sciences and Technologies Bulletin of the ACM Slovakia*, vol. 3, pp. 84–92, march 2011.

[37] F. Arcelli Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Inf. Sci.*, vol. 181, pp. 1306–1324, April 2011.

[38] "Automated drawing of uml diagrams." `http://www.umlgraph.org/`, May 2011.

[39] "Wikipedia: Filter (unix)." `http://en.wikipedia.org/wiki/Filter_(Unix)`, May 2011.

[40] "Antlr parser generator." `http://www.antlr.org/`, May 2011.