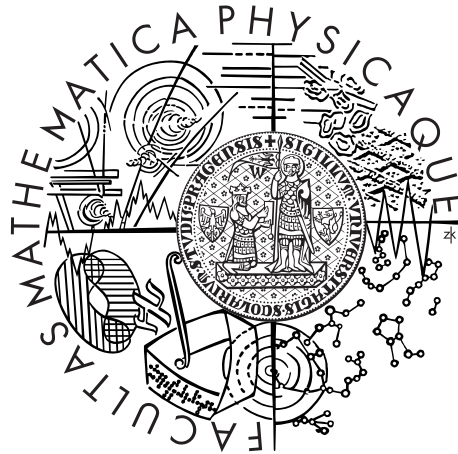


Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Štěpán Šindelář

Design Patterns Support in Development Environments

The Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Filip Zavoral, Ph.D.

Study programme: Computer Science

Specialization: Programming

Prague 2011

Poděkování TBD.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In date

signature

Název práce: Podpora návrhových vzorů ve vývojových prostředích

Autor: Štěpán Šindelář

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

Abstrakt: Flexibility, kterou návrhové vzory poskytují, je většinou dosaženo pomocí tvorby nových tříd. Členové vývojového týmu často nemají dost času pro tvorbu klasické textové dokumentace, a tedy vazba mezi třídami a návrhovým vzorem, který implementují, je ztracena. Nesprávné pochopení podstaty návrhového vzoru může způsobit komunikační chyby mezi programátory nebo dokonce chyby v softwaru. V této práci je představena sada nástrojů nazvaná Patters4Net určená pro platformu .NET. Programátoři mohou za pomoci Patters4Net označit návrhové vzory ve svém kódu pomocí speciálních atributů. Tato dokumentace může být následně využita nástrojem Pattern Enforcer, který verifikuje správnou implementaci některých vzorů. Nástroj Architecture Explorer používá dokumentaci o implementovaných vzorech ke generování diagramů podobných UML třídním diagramům, která ale zohledňuje vazbu mezi třídami a implementovanými vzory.

Klíčová slova: návrhové vzory, verifikace, vizualizace.

Title: Design Patterns Support in Development Environments

Author: Štěpán Šindelář

Department: The Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.

Abstract: The flexibility provided by design patterns is usually achieved by introducing new classes into the desing and developers often don't have enough time to create a textual documentation for them, therefore the mapping between classes and design patterns is lost. Incorrect understanding of a specific design patter can produce communication errors, or even software bugs. In this thesis we present the Patterns4Net project that targets the .NET platform. With Patterns4Net developers can annotate their classes using special attributes that documents the usage of design patterns in a standardized way. This documentation is then used by Pattern Enforcer, a tool that verifies correctenes of design patterns implementation, and Architecture Explorer, which uses information about implemented design patterns to generate UML-like class diagrams that emphasize the connection between design patterns and concrete classes.

Keywords: design patterns, verification, vizualization.

Contents

1	Introduction	3
1.1	Patterns4Net	5
1.2	Thesis structure – should be updated to fit the new structure. . .	6
2	Design patterns support	7
2.1	Patterns formalization and verification	7
2.2	The role of programming language	10
2.3	Tools support	13
2.3.1	Patterns4Net	15
3	Pattern Enforcer	17
3.1	Features	17
3.2	Usage	21
3.3	Architecture	26
3.3.1	Overall architecture	26
3.3.2	CIL processing	29
3.3.3	Patterns representation and discovery	32
3.3.4	Pattern Enforcer Design	34
3.4	Comparison	35
4	Architecture Explorer	39
4.1	Features	39
4.2	User Interface	41
4.3	Architecture	43
4.3.1	Model-View-ViewModel	43
4.3.2	Diagram Classes Design	44
4.4	Related Work	45
5	Graphviz4Net	49
5.1	Public API	50
5.2	Architecture	51
6	Conclusion	53
	Bibliography	54

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 thesis 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, a design pattern consists of

- a name to provide a common vocabulary,
- a description of a problem and it's context,
- a proven and widely-accepted solution to this problem,
- the consequences of applying the pattern ([2]).

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 ([2]) 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 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 different kind of patterns. For example, so called business patterns described in [3], or enterprise patterns from [4]. Principles discussed in [5] might be as well considered as patterns, although on higher level of abstraction than original design patterns. We could continue to enumerate more of them. In this thesis we mainly focus on design patterns as described in [2], where the authors define the 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 new complexity into the design. This complexity is caused by 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 design patterns is lost. 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, incorrect understanding of some design patterns by one of the development team members may slow down the development process or even lead to introduction of software bugs in the system. For instance, when one part of the system expects objects of a specific type to be immutable, but a developer unaware of what immutability means changes this behaviour. In this case formal verification might help.

While tools for 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 industry is not adopting 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 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 its disadvantage is that it cannot correctly recognize all the design patterns, since 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 previous paragraphs are addressed by the Patterns4Net project, whose presentation is the main aim of this thesis. 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 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 Domain-driven-design approach, these are mainly represented by domain specific classes, which are important for 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 Patterns4Net is that developers will annotate their code using .NET attributes mechanism and 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 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 Thesis structure – should be updated to fit the new structure.

In the following chapter entitled Design patterns support (2) we discuss possible kinds of design patterns support in development process. 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 section 2.1. Various approaches to ease the implementation of patterns on source code level are discussed in section 2.2. Tools for design phase support (e.g., special UML extensions) are presented in section ???. Since one of the main features implemented as a part of Patterns4Net tool set is the source code verification, we examine it more closely in the section ???.

In the third chapter we focus on features of Patterns4Net tool set (??). Section ?? provides detailed description of Patterns4Net functionality and in section ?? we show basic usage scenarios. In section ?? titled A case study we provide a walk-through of larger and complete example of usage of Patterns4Net. In this section the advantages of support of patterns are presented on real world example. Last section ?? provides a comparison of Patterns4Net to similar tools mainly for Java platform.

Architecture of the software is analysed in fourth chapter (??).

In the conclusion we summarize the thesis 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 software to overcome some of the disadvantages of design patterns. Software tools may enhance implementation of design patterns on the source code level, for example, by code generation or refactoring. On 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 source code level or 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 pattern than the others.

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 rigorous reasoning (e.g., for formal verification) and encumbers any automation tools support. The need for a formal specification of design patterns is obvious. In this section we discuss patterns formalization techniques. It is important to note that formalization of patterns is not intended to replace 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 description, solution and consequences. These parts could also be broken down into smaller pieces. The solution part can be decomposed to structural aspect and 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 mean of all widths or 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 programming¹. Some authors assume that the *Composite* pattern should always have methods for adding and removing components ([6]). "Children related operations" are indeed mentioned in [2], but does it mean that a *Composite* class must always be mutable (allow to change its children collection)? Does it mean that immutable quasi *Composite* class that

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

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 precise pattern formalization. But the approach should balance the degree of formalization and the degree of flexibility. We think that during the process of actual formalization of concrete patterns the informal description (e.g., in [2]) 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. *Bridge* and *Adapter* patterns differ only in the intent: *Bridge* is used during the design phase, but *Adapter* is used to wire up already existing classes.

Structural formalization

Most of the design patterns solutions involve more cooperating classes or objects. In [2] authors use the term participant for each kind of these classes and objects, term role is used as well in literature ([6]). In the *Composite* pattern solution we have a *Composite* class, *Leaf* objects and the *Component* the base interface for *Composite* and *Leafs*. This implies that if we have some 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 languages like C#), otherwise it is not correct implementation of the *Composite* pattern. This is simple example of structural aspect of the *Composite* pattern solution, whose formalization could be rather straightforward. The same holds for the fact that in valid implementation of this pattern the *Composite* class should aggregate a collection or list of *Components*.

The *Composite* pattern solution also guides us to implement operations on *Composite* class by delegation to the *Components*. This is more complicated to formalize since the delegation to the *Components* could have several different forms. The special case might be a situation when the composite operation returns cached value, which is refreshed after each addition or removal of a child. Besides this very special case we could say that composite operations should iterate the *Components* collection. Could we also say that the composite operation should call 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 advantage of some flexibility in design patterns implementations.

In previous two paragraphs we rather informally described 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 correctly implemented pattern. Existing formalization techniques are usually based on mathematical formalisms. For example, the Balanced pattern specification language (BPSL, [7]) leverages first-order logic, because relations between pattern roles can be easily expressed as predicates. In BSPL a pattern is specified using 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) like $Invocation(m_1, m_2)$ where $m_1, m_2 \in M$, which evaluates to true iff² method m_1 invokes method m_2 . The structural specification of the *Observer* pattern in S_{BSPL} is given in figure 2.1. English names of the predicates are self-describing.

```

 $\exists$ subject, concrete_subject, observer, concrete_observer  $\in C$ ;
subject_state, observer_state  $\in V$ ;
attach, detach, notify, get_state, set_state, update  $\in M$  :
Defined_in(subject_state, concrete_subject)  $\wedge$  Defined_in(observer_state, concrete_observer)  $\wedge$ 
Defined_in(attach, subject)  $\wedge$  Defined_in(detach, subject)  $\wedge$ 
Defined_in(notify, subject)  $\wedge$  Defined_in(set_state, concrete_subject)  $\wedge$ 
Defined_in(get_state, concrete_subject)  $\wedge$  Defined_in(update, observer)  $\wedge$ 
Reference_to_one(concrete_observer, concrete_subject)  $\wedge$ 
Reference_to_many(subject, observer)  $\wedge$  Inheritance(concrete_subject, subject)  $\wedge$ 
Inheritance(concrete_observer, observer)  $\wedge$  Invocation(set_state, notify)  $\wedge$ 
Invocation(notify, update)  $\wedge$  Invocation(update, get_state)  $\wedge$ 
Argument(observer, attach) Argument(observer, detach)  $\wedge$ 
Argument(subject, update)
```

Figure 2.1: Structural specification of the *Observer* pattern in S_{BSPL}

To employ such formalization in practical use for verification or recognition, we need to evaluate the predicates according to source code or other representation of object oriented program. Interesting proposal is discussed by authors of SPINE ([8]). They suggest to use Prolog language. We can represent constraints for pattern structure as Prolog rules and those rules that depend on source code analysis (like *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 corresponding rules to the database and then is able to answer questions like 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. Figure 2.2 shows structural specification of the variant of the *Singleton* pattern in SPINE and a Java class that implements the *Singleton* pattern.

A promising approach might be to express patterns as stereotypes in UML and use 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 OCL, also part of the MDA, might become popular and widely

²if 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 2.2: The *Singleton* pattern in SPINE and Java.

used as well in the future. Another approach leverages semantic Web technologies ([11]). Design patterns can be defined as RDF documents instantiating a vocabulary based on the Web ontology language (OWL). This approach promotes the usage of design patterns as a knowledge shared among software developers. Lastly LePUS3 ([12]), which is one of the most accepted and well known approaches, provides graphical notation for expressing the structural aspects of design patterns. Figure 2.3 shows specification of the *Composite* pattern in 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 programming language determines what can and what cannot be implemented easily. In [2] authors assume Smalltalk/C++-level language features. If they assumed procedural languages, they might introduce patterns like "Inheritance" or "Encapsulation". But there are also important differences between object-oriented languages. For example Groovy supports multiple dispatch, which lessens the need for the *Visitor* pattern ([2]).

New features. Since first publication of [2], 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 4 of the top 5 most popular languages ([14]), therefore we can say that nowadays lambda functions can be considered as an essential feature. Let us investigate what lambdas may bring to design patterns. In most design patterns solutions 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.

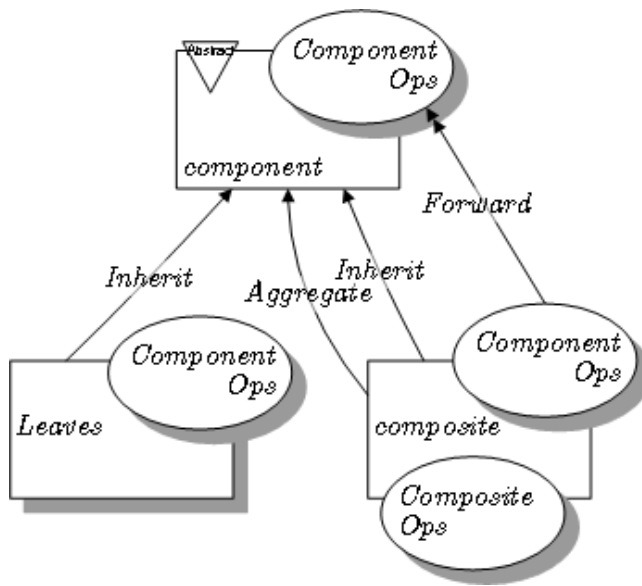


Figure 2.3: Specification of the *Composite* pattern in LePUS3 ([13]).

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 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 lambda functions that are passed to it as parameters, or that are given to the object as constructor parameters. This alternative does not require creation of 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 a 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 it gets as constructor parameters. Method `ForEach(Action)` of the class `List` from .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 .NET. They are supported in .NET because the *Observer* pattern is suitable for event driven applications, which are developed in .NET quite 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 enable these also in C# or Java. Implementation of some of the design patterns in ApectJ, aspect oriented extension for Java, is presented in [15]. An

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

Figure 2.4: Non-generic IGraph interface

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

Figure 2.5: Generic IGraph interface

article [16] presents Ruby’s advanced features, for instance meta programming, that simplifies the implementation of design patterns.

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

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 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 figure 2.4.

Now users can add whatever they want as a vertex, but some users may want to work with just 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. Unnecessary casting was one of the reasons for introducing generics into the .NET runtime. Generic version of the `IGraph` interface might look like in figure 2.5.

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 same, we have to use `new` keyword as in figure 2.6.

How shall we now implement the new interface? That is where 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

```

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

```

Figure 2.6: 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 2.7: Fragment of implementation of the generic IGraph interface.

see a fragment of the implementation in figure 2.7. A consequence of this implementation is that we have to explicitly cast the class to non-generic `IGraph` if we want to work with 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 which works only with non-generic version of the interface, but in this case, the cast is done automatically.

Future trends. Mainstream programming languages are usually not adopting new features because their authors want to ease 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 patterns obsolete or almost disappear for instance the *Observer* pattern in .NET, but they also might bring new challenges. Authors of 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 over-complicated. Some patterns (e.g., the *Visitor* pattern) are quite complex and not so widely used so that support for their implementation in Java, C# or C++ (double-dispatch for the *Visitor* pattern) is not likely in the near future.

2.3 Tools support

Some of the complexity of implementing and maintaining design patterns is caused by introduction of new classes and new methods into the design. During the development the connection between classes and concrete design patterns

roles might be lost and the system that used to be well designed, perfectly lucid and easily extensible become 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 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 a double dispatch in this pattern is implemented. If they want to add new operation and they have the information that these classes can be visited by 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 complete lack of documentation, or either inaccuracy of textual documentation in natural language. One possibility to overcome these problems might be standardized documentation of design patterns instances. For the Java platform there is a project called JPatterns ([17]), which provides annotations to mark patterns in Java code. At the moment it only provides 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 a 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 and 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, they 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 industry is not adopting 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 its disadvantage is that it cannot correctly recognize all the design patterns, since 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.

2.3.1 Patterns4Net

In order to evaluate the ideas stated in previous paragraphs, we implemented a prototype project called Patterns4Net, which is a set of tools that support the development of object oriented software on .NET platform. These tools take advantage of 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 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 patterns in the standard distribution, but users can add their own patterns. Patterns4Net consists of Pattern Enforcer and Architecture Explorer.

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 like "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 the 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 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 infrastructural detail, on the other hand, for instance, patterns from Domain Driven Design or from Patterns of Enterprise Architecture are usually represented by domain specific classes.

3. Pattern Enforcer

3.1 Features

Patterns constraints specification

Pattern Enforcer is a tool for developers and it is expected to be used by developers only during the development process, not in the production builds. Because of these facts we decided to provide internal type safe C# Domain Specific Language (DSL, [18]) for constraints configuration instead of xml based configuration or custom constraint language.

In [18], Martin Folwer defines DSL as a computer programming language of limited expressiveness focused on a particular domain. There are two types of DSLs – internal and external. External DSLs are completely new languages with their own custom syntax, while internal DSLs are embedded into existing general purpose language such as C#, Java or Ruby by providing specific public API.

Type safe DSLs use constructs that can be verified by a compiler rather than strings with special internal syntax that could be verified only during the runtime or by a additional tool. For example, NHibernate ORM framework ([19]) has such API for a definition of objects to database schema mapping. Instead of expressing the names of properties as a strings, NHibernate exploits the C#'s feature of lambda expressions for this purpose and thus existence of properties used in the mapping is verified by a compiler. For a better idea of this approach, figure 3.2 shows a short example of the NHibernate DSL.

```
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 3.1: Example of internal type safe DSL.

Usage of type safe DSLs also enables integrated development environments support. Namely intellisense support could make the development more effective and could bring better experience for developers who don't know the DLS 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. Xml based configuration or external DSL might provide more flexible solution in such case.

DSLs usually leverages a technique called a Fluent API, which means that a method returns an object on which a user is expected to invoke another method. This chaining of methods may make the API more self describing, because the code can be then read almost as a sentence. The example of Pattern Enforcer unit tests Fluent API is shown in figure ??.

```
patternEnforcer.AssertThat<WidgetContainer>().IsComposite();
```

Figure 3.2: Example of Fluent API.

For specifying constraints about a design pattern instance we need a data structure that would capture the information about the roles in this design pattern instance. By the term design pattern instance we mean concrete classes that implement the pattern, but not their particular instances. For better illustration, an instance of the *Composite* pattern is given in figure 3.3. From the structural point of view the *Composite* pattern has two roles: the *Composite* class (in this instance represented by the `WidgetComposition` class) and the *Component* interface (the `IWidget` interface), which should be implemented by the *Composite* class.

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

Figure 3.3: Example of the *Composite* pattern instance.

In Patterns4Net, we represent patterns as a Common Language Runtime (CLR) classes derived from the `IPattern` interface. A pattern class should contain properties whose values represent the participants of the pattern. In case of the *Composite* pattern, the class for its representation would contain properties "Composite" and "Component". An instance of such class representing the pattern instance from our example would contain a reference to the `WidgetComposition` type as the value of the "Composite" property and a reference to the `IWidget` interface as the value of the "Component" property.

References to types are represented as instances of the `TypeReference` or the `TypeDefinition` classes from library Mono Cecil, which are similar to the `System.Type` type from the standard library. This is because we use Mono Cecil for processing of .NET assemblies¹. A participant of a pattern might be also a method (represented as `MethodDefinition` or `MethodReference` from Mono

¹Reasons why we have chosen Mono Cecil and more detailed information about it are presented in the subsection 3.3.2

Cecil) as in the *Simple Factory Method* pattern. Generally speaking class representing a pattern might contain whatever the pattern author finds useful but public properties of types listed above have a special meaning for Patterns4Net. Figure 3.4 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 3.4: The *Composite* pattern definition for Patterns4Net.

Now when we have the strongly typed representation of design patterns instances, we can build a type safe DSL for their constraints specification. 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 constraints easier than that. The key part is that it enables to specify constraints as lambda functions (we call such function a "check"). Check may be performed on the whole pattern instance, than the parameter of the lambda function will be the object representnig the pattern. These checks may verify relations between roles, for example that the *Composite* class implements the *Component* interface. Users can also set up checks 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, user can create check only for the value of the selected property. Finally users can also select specific methods of the selected type to provide a check for each of them. The selection of these methods is also done using a lambda function. To summarize it up: user can select a subject of the check, using lambda functions, and than he enters the check itself again as a lambda function, which takes the subject of the check as a parameter. For better idea, an example is shown in figure 3.5.

A check expression might be anything, which enables wide range of possibilities for experienced users, but Pattern Enforcer provides easy to use extensions to underlying 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.

Supported patterns

```
// 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 3.5: An example of constraints configuration in Pattern Enforcer.

As we claim in the section 2.1 constraints for built-in patterns were chosen rather less restrictively than in 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*. Patterns supported by Pattern Enforcer by default are listed in the appendix A.

Relatively unrestrictive API for patterns constraints specification allow us also 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 internal state of it's instance once it is available to the "outside world". What does this bring us? Simple immutability in C# can be enforced by specifying 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 it returns it to the "outside world". Also backing fields of auto-implemented properties, which brings a notable simplification of implementation of simple properties, cannot be specified as read-only.

User Interface

Pattern Enforcer provides usual command line interface and MSBuild task that can be used in MSBuild scripts. Visual Studio project formats (e.g., *.csproj) are basically MSBuild scripts, so Pattern Enforcer can be easily included in build process. In both cases the input is .NET assembly and output is text printed to standard output in one of three supported formats (plain text, Visual Studio format or xml).

Besides these two user interfaces, Pattern Enforcer provides public API designed to be used inside unit tests. Through this API, developers can invoke whole Pattern Enforcement checking process based on patterns annotations, or they can invoke a single check of conformance of a specific class to a specific

pattern. In the former case, it is not required to decorate the checked class with patterns attributes.

Configuration

The configuration of constraints that are enforced on patterns implementations can only be done in code. However, Pattern Enforcer CLI and MSBuild task can be configured using xml. In this configuration, user can turn off all checks of given pattern and provide paths to assemblies containing custom patterns specification. Checking by Pattern Enforcer can be turned off also by special attribute `PatternEnforcerIgnoreAttribute`, which has string property `Justification`, where developers should provide a description why they have disabled the checks on this class or method.

3.2 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 reference to the *Patterns4Net.Attributes.dll* assembly in the project. This assembly contains only attributes definitions, thus it's footprint should be minimal. It is build for .NET version 2.0, so Pattern Enforcer can be basically used in projects build for older versions of the .NET. When the reference is added, the types can be decorated with attributes from the namespace `Patterns4Net.Attributes`. A code example is provided in 3.6. 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.

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

Figure 3.6: Example of internal type safe DSL.

As we can see, the implementation of the *Composite* pattern is not valid in this case, because the getter method of the `Width` property is not using the `children`

```
C:\Windows\system32\cmd.exe
\nEnforcerExample>..\..\integration\release\pattern-enforcer -config-file enfor
cer-config.xml .\bin\Debug\EnforcerExample.dll

Type WidgetComposition, pattern Composite - ERROR, (checked rules: 3, errors: 2)
The class WidgetComposition should contain a generic collection of components (e
.g. IEnumerable<IWidget>) as one of it's fields.
The composite method get_Width in class WidgetComposition seems not to iterate t
hrough the collection of components. Try using foreach loop or Ling extension me
thods to process the collection.
```

Figure 3.7: The output of Pattern Enforcer for the `WidgetComposition` class.

collection. Pattern Enforcer can be run outside the Visual Studio or inside the Visual Studio. Firstly we will describe the first option. When the Visual Studio project is built, there should be a resulting assembly in the output folder (usually `{project folder}\bin\Debug`). Say its name is *EnforcerExample.dll*. Then if the *pattern-enforcer.exe* is run from command line supplied with a path to *EnforcerExample.dll* as an argument, it should produce the output shown in figure 3.7.

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 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 file), it should be uncommented, and an invocation of the Pattern Enforcer task should be inserted as its child element. The figure 3.8 shows the xml code and also a screenshot of Visual Studio displaying the warnings.

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 its roles is constructed 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 fixture² 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 with call to `AssertThat` is used as main role

²This term is used by the NUnit framework ([20]), some of other xUnit frameworks also use the term "test suite" instead of a test fixture.

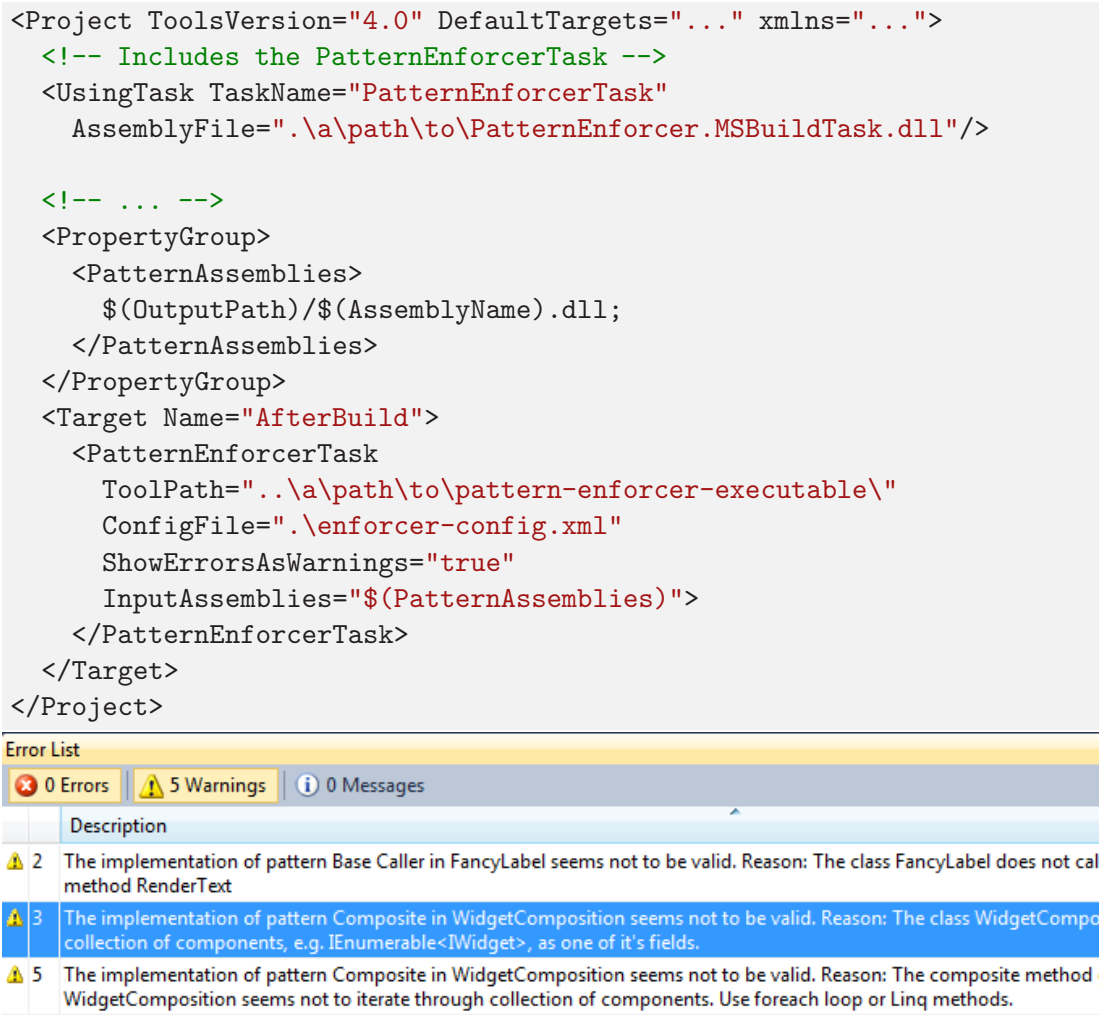


Figure 3.8: Integration of Pattern Enforcer and Visual Studio 2010.

of the pattern, other additional required information, if needed, are supplied as parameters of the `Is{PatternName}` method. The figure 3.10 shows an example of such test fixture using the NUnit framework ([20]).

Specification of custom pattern

There are two possibilities to define a pattern and constraints that will be enforced on its 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 **TODO: section number**. Here we describe how to use the second one. We will describe an implementation of pattern we will call *Base Caller*. It has two roles: the Target class and the Target's base class. The constraint we will specify is that the Target class is required to invoke the corresponding base methods in overridden methods bodies.

```

[TestFixture]
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 3.9: Example of an automatized test that invokes Pattern Enforcer.

Custom pattern is represented by a class that inherits from the `IPattern` interface and it is recommended to also implement the `IPatternAttribute` interface, which is just 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.

The `IPattern` interface requires just the getter of the property named `Name` and the getter of `AbstractionLevel` property, which is used by Architecture Explorer (null can be used as 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 figure ??.

```

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

```

Figure 3.10: Example of an automatized test that invokes Pattern Enforcer.

During the processing of patterns attributes, Patterns4Net needs to reconstruct the **BaseCaller** from CIL metadata. The metadata does not contain an instance of the attribute, instead it contains only values of constructor arguments used for its instantiation and names and the values of the properties that were assigned. For example, metadata for the **PatternEnforcerIgnoreAttribute** in figure 3.11 would contain: zero constructor arguments, because the parameterless constructor of **PatternEnforcerIgnoreAttribute** was used; and one property with name **Justification** and it's value.

```
[PatternEnforcerIgnoreAttribute(
Justification="A constant value")]
public class AnnotatedClass
{
}
```

Figure 3.11: A code example to illustrate CIL metadata for attributes.

For the purpose of reconstruction of pattern attributes from CIL metadata, 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 class should be able to reconstruct it's instance from this dictionary, which provides the following data:

- Constructor Arguments – indexed by the number of 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 property is of type **System.Type**, than it's actual value will be **TypeReference** from Mono Cecil refering to the same type.

The implementation of such constructor for the **BaseCaller** class is shown in figure 3.13. An instance of **TypeReference** class can be converted to corresponding **TypeDefinition** instance using method **Resolve()** as in the example.

The pattern, as declared in figure 3.13, can be used for annontation 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. First 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 we have described above, is inteded to be the

³More technical reasons that lead to this decision are given in section 3.3


```

public class BaseCaller : 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 3.12: Example of an automatized test that invokes Pattern Enforcer.

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 one we describe in the section 3.1. For completeness of the example, figure ?? shows the final implementation of the *Base Caller* pattern.

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 Pattern Enforcer configuration file. A configuration is an xml file, the `pattern-enforcer-config.xsd` file with definition of it's structure is supplied with Patterns4Net. The location of a configuration file is provided to Pattern Enforcer as a command line option, or the parameter of the MSBuild task.

3.3 Architecture

3.3.1 Overall architecture

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 .NET platform version 4, 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 3.14 we can see the layout of Visual Studio solution used for Patterns4Net development.

Projects that start with "Patterns4Net.PaternEnforcer" are related to Pattern Enforcer tool. Classes that provide the core functionality of Pattern Enforcer and classes that form the unit-testing public API are located in the *Patterns4Net.PatternEnforcer.Core* project. The output of the *Patterns4Net.PatternEnforcer.Cmd*


```

public class BaseCaller : IPatternCheckerProvider, ...
{
    // ... 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 3.13: Definition of a checker for custom pattern.

project is command line interface for the Pattern Enforcer and the project *Patterns4Net.PatternEnforcer.MSBuildTask* is implementation of the task for the MSBuild engine. Core functionality and unit-testing API of Pattern Enforcer are decoupled from command line interface and MSBuild task into separate project, and thus separate assembly, because the Pattern Enforcer core functionality is used also in *Patterns4Net.ArchitectureExplorer*. The resulting assembly is also ment 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, unusual suffix might confuse some users.

Project *Patterns4Net.ArchitectureExplorer* contains the code of the Architecture Explorer GUI tool. The GUI is done in Windows Presentation Foundation (WPF) framework. In this project, besides C# classes, also XAML⁴ files are included.

Automated tests are used during the development of Patterns4Net. These tests are located in the *Patterns4Net.Tests* project. This project aggregates tests for classes in 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.

⁴Extensible Application Markup Language

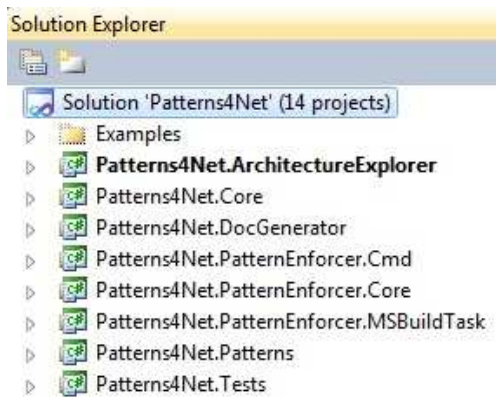


Figure 3.14: The layout of Visual Studio solution.

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

Finally the project *Patterns4Net.Patterns* contains only 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 in 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 a 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 ([20]) is used.

Extensibility. For better support of extensibility the Managed Extensibility Framework (MEF, [21]) is used. The MEF provides discovery and composition capabilities that are employed in plugins mechanism of Patterns4Net.

Code Contracts. Most of the classes define their contracts using Microsoft Code Contracts ([22]). Contracts help us to find issues earlier. Exception is thrown typically during the pre-conditions check, which is the real cause of the problem, rather than later on a source code line that expects valid input parameters. Code Contracts also serve as a complement to the API documentation.

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

3.3.2 CIL processing

The analysis of the source code could be done using the original textual source code, which can be parsed and represented as an abstract syntax tree (AST). With this approach 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 only the one language or, if we used parsers capable of parsing more languages into the same AST, we would still have to deal with some language specific constructs.

The other option, which we have chosen, is to analyse the intermediate language, in case of the .NET it is Common Intermediate Language (CIL), sometimes called Microsoft Common Intermediate Language (MSIL). 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 had tested it only on C#. One of the disadvantages of this approach is that the CIL is stack based lower level language and 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 reflection API that is available as a part of .NET base libraries. Second option is Microsoft Common Compiler Infrastructure (CCI), which is developed in Microsoft Research. Last option is Mono Cecil, which is developed as part of the Mono open-source project.

Standard .NET Reflection API looks at assemblies as a code, not as a raw data, which has two important consequences: the code loaded through .NET Reflection API can be executed; and, because the code can be executed, the runtime must check access right and might throw Code Access Security exception. Assemblies loaded into an AppDomain (which is a .NET object similar to a process in a operating system) cannot be unloaded. Finally the .NET Reflection API does not distinguish between type definition a type reference, which is an entry in assembly metadata 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 AppDomain and execution of the loaded code. On the other hand they are claimed 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 provide⁵, however the AST the CCI generates is more complex than we would need, therefore, for our purposes, the advantage of generated AST would be lessened by extra work for its 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's 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 favour 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 whole Patterns4Net project.

CIL analysis

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 its 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 its meta-data. For such purposes there is a project Mono Cecil Rocks, which contains a few extension methods for 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 constraints 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, there is an attribute "overrides" in 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 purposes of 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 analyse conditional statements – we just want to know whether method M1 on field F was invoked in body of method M2, even in dead branch of code.

⁵There is project Cecil Decompiler, but it not in production ready quality.

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 argument, so the only difficulty is to analyse with which actual parameters the method was called.

CIL virtual machine is a stack based machine, which means that all arguments for operations are taken from the evaluation stack and result are pushed onto the stack. Usually instructions pop all their arguments from the stack and push results onto the top. Stack behaviour 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 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. 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 behaviour). An 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 patterns matching

In order to check some more specific constraints such as the specification for the *Singleton* pattern implementation, we need to check whether method body contains a specific CIL instructions pattern⁶.

⁶here the term pattern has a slightly different meaning than a design pattern.

The main class for CIL instructions patterns matching is `CILPatternsMatcher`. It aggregates a collection of instances of the `InstructionMatcher` abstract class, which controls the matching process. Interface of the `InstructionMatcher` class is shown in figure 3.15. Method `Matches` is called in a loop on the current instruction. If the method returns `false`, then the CIL instructions does not match the pattern 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`. Pseudo code is given in figure 3.16.

```
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 3.15: The `InstructionMatcher` abstract class interface.

```

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

Figure 3.16: Pseudo code of CIL instructions patterns matching.

3.3.3 Patterns representation and discovery

Patterns representation is described in section ???. Here we just remind that a pattern is represented as a CLR class derived from `PatternBase`. It contains properties representing the participants of the pattern and any other information that the pattern's author finds useful. Mono Cécils structures are used for types and methods identification.

Discovery of patterns meta-data is implemented as a flexible mechanism. There is a central class named **PatternsLocator**, which aggregates objects that implement interface **IPatternsProvider**. Each of these objects provide a strategy method **GetPatterns**, which for given type or a method returns a list of patterns. The method or the type is then considered as a main role of the returned pattern instances. However, the pattern instances may contain other roles. The main role of a pattern is usually used in Pattern Enforcer's error messages, in the section about Architecture Explorer we also refer to this term several times.

The two implementations of **IPatternsProvider** that are supplied with Patterns4Net provide location of patterns meta-data based on Patterns4Net attributes. Diagram in the figure 3.17 shows the class hierarchy. The collection of **IPatternsProvider** instances is supplied to the **PatternsLocator** class as constructor parameter. The **PatternsLocator** class is usually constructed via MEF container and thus the constructor parameter is resolved automatically by MEF. If we want an instance of some class to be registered in the container as **IPatternsProvider** implementor, the only thing we have to do is to decorate such class with attribute `[Export(typeof(IPatternsLocator))]`.

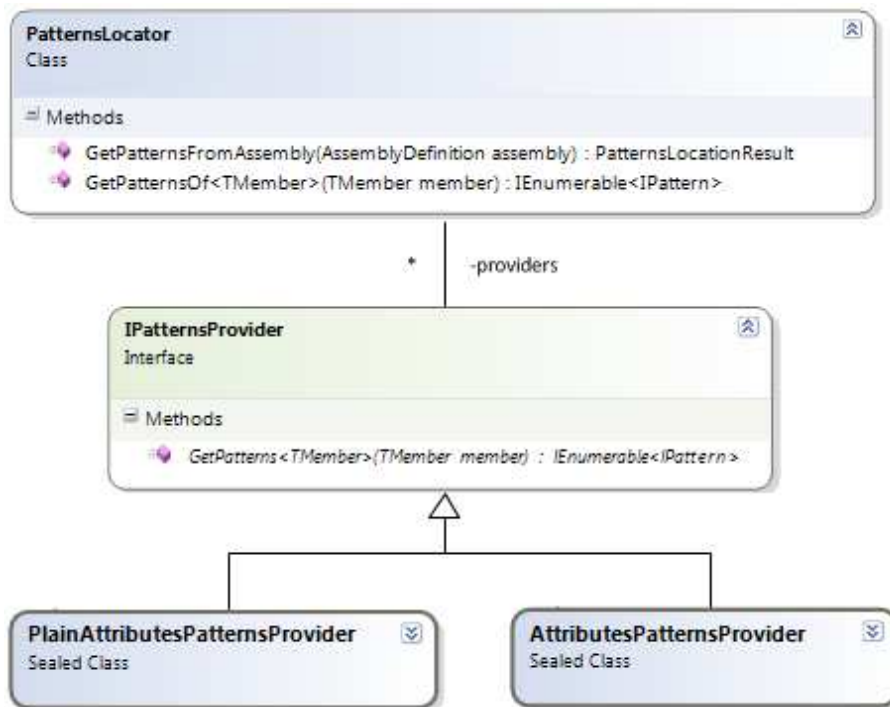


Figure 3.17: Hierarchy of classes that are used for discovery of patterns meta-data.

The **PlainAttributesPatternProvider** and **AttributesPatternProvider** classes responsibility is to transform CIL meta-data information about attributes into objects that represent the discovered patterns. This task is not as easy as it may seem.

Firstly classes that represent the Pattern4Net attributes are not identical to classes that represent the patterns. This decoupling provides flexibility, because patterns meta-data does not have to be based only on attributes, but in this case it causes some additional work to be done.

Secondly CIL meta-data parsed by Mono Cecil are not real instances of the classes that represent Patterns4Net attributes. Rather Mono Cecil provides information about what the type of the attribute is, which values are provided as it's constructor parameters and which named arguments were provided. If the user of Mono Cecil wants to construct an identical instance of the attribute, he has to do it by hand. Actually the `PlainAttributesPatternProvider` class even does not have to construct Patterns4Net attributes instances, instead the `PlainAttributesPatternProvider` class constructs objects representing the patterns directly.

There are two basic options to solve this task. We could say that Pattern4Net attributes should always implement a constructor with name-value dictionary as a parameter. The constructor would reconstruct the attribute object from this dictionary. (This approach is also used in the MEF.) Finally the attribute object itself would represent the pattern instance. The other option is to absolutely decouple the creation of an object representing the pattern and the attribute itself. In this case, there would be a separate class that would construct the object representing the pattern instance from Cecil's meta-data structures and the attribute would be just a dummy data holder.

First option provides higher cohesion – we don't have to create another class, but the attribute and the process of creation of an object representing the pattern are coupled. This approach could be suitable for scenario when a user wants to implement his custom pattern and so Patterns4Net provide this possibility, which is implemented in the `AttributesPatternProvider` class. But internally Patterns4Net uses the former approach, because it cuts down the code needed in Pattern4Net attributes classes and thus minimizes the footprint of the assembly containing Pattern4Net attributes, which has to be referenced by a project that makes use of Pattern4Net. The former approach is implemented in `PlainAttributesPatternProvider`.

3.3.4 Pattern Enforcer Design

The basic interface in the Pattern Enforcer design is 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 `PatternBase` class.

An important class is the `FluentPatternChecker`. It is a base class of most of the pattern checkers, because it provides the structural constraints specification DSL. The DLS is formed by protected methods of the `FluentPatternChecker` class, which should be invoked in the constructor of derived class in order to

specify the constraints. Because we wanted to provide a Fluent interface (chaining of method calls), these methods return special objects that have

- a reference `a` to the parent object 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 example 3.18.

```
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 3.18: 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. These classes have also their dependencies requested as constructor arguments so 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))]`, dependencies of such type will be resolved by MEF recursively.

3.4 Comparison

FxCop and Gendarme tools.

It may not be obvious at the first look, but Pattern Enforcer is similar to static analysis bug-hunting tools such as FxCop or Gendarme ([24]). These tools search the source code for 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 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 is verifies that expected idiom is present.

Gendarme is open-source tool that is ment 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`.

A Pattern Enforcing Compiler (PEC) for Java.

Because a Pattern Enforcing Compiler (PEC) for Java ([25]) has been the main source of inspiration for our thesis, 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 than checks whether the classes actually implement specified patterns.

Annotation. For annotation of patterns instances, PEC uses so called marker interfaces. The authors have choosen this technique, because implemented interfaces are listed in generated API documentation and so an intergration 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 limited number of additional information. Authors of PEC admit these weakneses of interfaces as a technique for the annotation of patterns and in [25], 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 stndard Java annotations.

Pattern Enforcement. PEC uses static analysis and it also enforces the rules dynamicaly 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 dynamically one simple assetion is enough to enforce it. The disadvantage of dynamical enforcement is that it slows down the resulting program and to discover bugs the program still has to be manually tested.

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 will just create 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 standard Java reflection API and Javaassist, which is similat 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 signal the errors during the check of pattern implementation. This means that usually when 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 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 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 a MSBuild task and unit-testing API.

Other tools.

CoffeeStrainer ([26]) 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.

Pattern-Lint ([27]) 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 discuss also in section 2.1, and tools that come with LePUS3, which we also present in the same section.

4. Architecture Explorer

4.1 Features

Further meta-information

Relationships. Architecture Explorer does not only display relations that are part of some pattern implementation, but 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 semantic. 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` 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.

Layers and packages. Architecture Explorer does not support hierarchical packages like UML does, but instead provides a concept of *Layer* which is a container for packages. Normally *Layers* correspond to each .NET assembly, but users can define *Layers* on their own using assembly attribute as in the figure 4.1. First level namespaces in a layer are 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 4.1: Definition of *Layer* using assembly attributes.

Inputs

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.

Outputs

Architecture Explorer has interactive graphical user interface (GUI). It displays class diagrams on the four levels of abstraction allowing user to view the architecture from higher level overview (*Layers level*) to detailed diagram of specific class and all it's collaborators (*Class level*). Relationships and classes that play a role in some pattern implementation are graphically remarked. In the following paragraph we describe how the Architecture Explorer chooses which classes will be displayed at which level. Before that we introduce a related terminology.

Classes that implement patterns *Entity*, *AggregateRoot* or *ValueObject* known from Domain-Driven-Design approach are level 0 classes. Classes that implement some infrastructural pattern like *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 build in patterns are defined in xml configuration file and can be changed by users as well. Term element denotes a class, a class member, a package or a layer.

Now we can define all four levels of abstraction that Architecture Explorer provides.

- Layers level – layers are displayed as rectangles. Each one contains it's packages, which are displayed as well as rectangles. All elements have labels. 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 one class of level zero, 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 one 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 its 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.

4.2 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 serve 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, its content is added into the current diagram. Therefore, if two assemblies are consequently loaded, the diagram will contain all elements from both of them. If there are some already loaded elements in the diagram and a user wants Architecture Explorer to display only elements from another assembly, he must use a button that clears the diagram before loading the new assembly.

A user can browse through the diagram using either buttons in the toolbar, or by clicking on elements displayed in the diagram. A click on a diagram element causes that appropriate level is displayed – if the click was on a class, a class level is displayed, and likewise for other element types. A button intended to go up to next higher level is available in the toolbar as well as the buttons intended to go back and forward during the navigation through the diagram.

Dockable side panels display 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 loading of an assembly or when Pattern Enforcer is running.
- Errors window contains a grid that displays errors from Pattern Enforcer.

Architecture Explorer can display API documentation generated from source code. Source of this documentation is a xml file produced by C# compiler. In Visual Studio to set up the compiler to produce this file, a user has to open the project properties panel, switch to the build tab and check the option "XML documentation file". 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. To add a documentation for a custom pattern, a user should edit 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.

Definition of custom pattern.

The definition of a custom pattern is described in section 3.2. 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 Architecture 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. All configuration files and their schema definitions are located in the *Config* folder.

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* ([28]) 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* ([29]) tool is used, but it's adoption to WPF is not as easy as it might seem at first look, so it resulted in an introduction of separate project called *Graphviz4Net*, which is discussed in the following chapter.

4.3.1 Model-View-ViewModel

The *Model-View-ViewModel* pattern is a variation of well known *Model-View-Controller* pattern. It's detailed description could be found in the article [28]. 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 ([30]) 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 4.2 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.

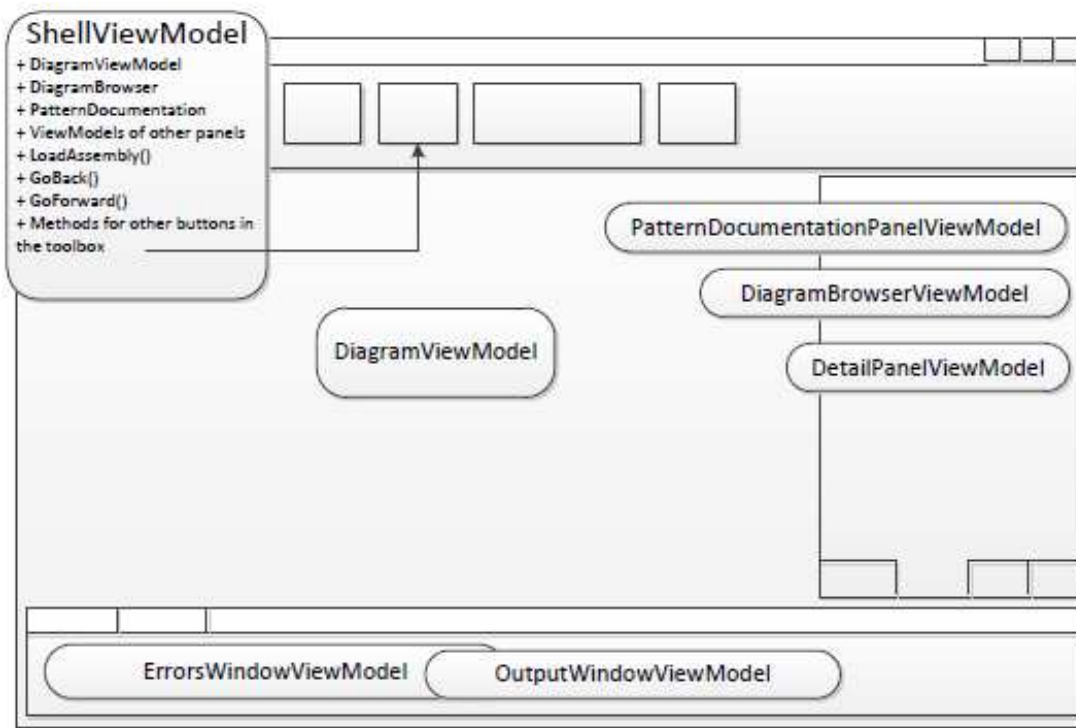


Figure 4.2: Decomposition of the user interface into several ViewModel classes.

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 `EventAggregator` object, which is a singleton. This method has one argument, which is an object that represents the event and its arguments. 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 `EventAggregator` 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 cause an update of several GUI elements.

4.3.2 Diagram Classes Design

A diagram is represented by hierarchy of classes that are depicted in the figure 4.3. Each of these classes inherits from the base class `DiagramElement`, which means that each instance of these classes have a reference to its parent object. The top level `Diagram` object returns a reference to itself as the value of this property.

Each type of relationship is represented as a separate class, because graphical templates for

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, the **CecilDiagramLoader** 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 **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 [31] published in the year 2010 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 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.

Pattern recognition tools.

A tool presented in [32] 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 [33]. 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.

UML reverse engineering.

The software called UMLGraph ([34]) 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.

Figure 4.3: Hierarchy of classes that represent a diagram.

5. Graphviz4Net

Graphviz ([29]) is an open-source graph visualization tool, which we use in Patterns4Net to create class diagrams. Graphviz is implemented as typical UNIX filter. Filters read data from standard input and write results to standard output. For graphs representation, Graphviz uses special language called DOT. An example of DOT file is given in figure 5. 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, but with attributes that provide information about the generated layout. Output format can be set up by command line option.

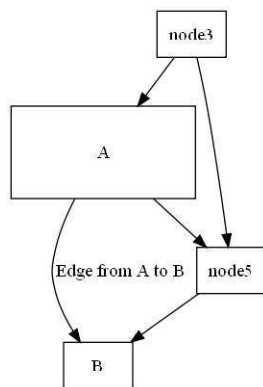
```
digraph G {
    node [shape=rect];
    node1 [label="A", width=2, height=1];
    node2 [label="B"];
    node3; node5;
    node1 -> node2 [label="Edge from A to B"];
    node3 -> node1; node3 -> node5;
    node1 -> node5; node5 -> node2;
}
```

Figure 5.1: An example of DOT file.

To employ Graphviz in generation 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 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.



```

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 5.2: Graphviz output for graph from figure 5.

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).

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 the graph structure, but not by sub-graphs, because edges may cross sub-graph boundaries. A diagram of discussed classes and interfaces is shown in figure ??.

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.

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 building of graphical elements, the *Layout Builder* is responsible for supplying 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 (it is probable 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

Graphs representation is described in the previous section, here we just remark that all the interfaces presented there follow the *Flexible Generic Interface* pattern as described in section ??.

DOT parsing. One of the tasks Graphviz4Net has to deal with is parsing the Graphviz output, which is a text in the DOT language. The grammar of the DOT language is defined in the documentation ([35]). We developed a parser based on ANTLR parsers generator 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).

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 its 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 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*. Part of the default implementation of the factory is shown in figure ?? . Note that 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 its 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 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 possible approaches for design patterns support in development environments and to develop Pattern4Net a set of tools that support pattern oriented development of .NET programs. Patterns4Net provide a standardized way to document design patterns instances in the source code and two tools that take advantage of this documentation. Pattern Enforcer verifies some of the structural aspects of design patterns implementations and Architecture Explorer provides 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 team of several programmers, 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. However, a comprehensive evaluation of this approach is beyond the scope of this thesis.

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.

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. Chapter I.
- [8] T. Taibi, *Design patterns formalization techniques*, ch. Chapter VI.
- [9] J. Warmer and A. Kleppe, “The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series),” 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] T. Taibi, *Design patterns formalization techniques*, ch. ChapterXII.
- [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] “File:composite pattern in lepus3.png.” http://en.wikipedia.org/wiki/File:Composite_p 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] M. Fowler, *Domain specific languages*. 2010.
- [19] “Nhibernate forge.” <http://nhforge.org>, May 2011.

- [20] “Nunit - home.” <http://www.nunit.org/>, Aug. 2010.
- [21] “Managed extensibility framework.” <http://mef.codeplex.com/>, Aug. 2010.
- [22] “Contracts - microsoft research.” <http://research.microsoft.com/en-us/projects/contracts/>, Aug. 2010.
- [23] T. Ecma, “Tg3. common language infrastructure (cli). standard ecma-335,” 2010.
- [24] “Gendarme – mono.” <http://www.mono-project.com/Gendarme>, May 2011.
- [25] H. Lovatt, A. Sloane, and D. Verity, “A pattern enforcing compiler (pec) for java: A practical way to formally specify patterns,” 2007.
- [26] 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.
- [27] 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.
- [28] J. Smith, “Wpf apps with the model-view-viewmodel design pattern.” <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>, Aug. 2010.
- [29] “Graphviz - graph visualization software.” <http://www.graphviz.org/>, Aug. 2010.
- [30] “Caliburn micro: A micro-framework for wpf, silverlight and wp7.” <http://caliburnmicro.codeplex.com/>, Aug. 2010.
- [31] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz, “Enriching reverse engineering with annotations,” *Model Driven Engineering Languages and Systems*, pp. 660–674, 2010.
- [32] 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.
- [33] 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.
- [34] “Automated drawing of uml diagrams.” <http://www.umlgraph.org/>, May 2011.
- [35] E. Koutsofious and S. North, “Drawing graphs with dot.” <http://www.graphviz.org/Documentation/dotguide.pdf>, 2006.