

Pattern4Net: Efficient Development Using Design Patterns

Štěpán Šindelář and Filip Zavoral

Charles University in Prague
me@stevesindelar.cz, zavoral@ksi.mff.cuni.cz

Abstract *The flexibility provided by design patterns is usually achieved by introducing new classes into the design. The complexity of design patterns oriented software development can easily overtake the advantages of design patterns usage, which might lead to software bugs or even complete failure of the development. We present the Patterns4Net project that targets the .NET platform. Developers can annotate their classes using special attributes that document the usage of design patterns. This documentation is then used by Pattern Enforcer, a tool that verifies the correctness of design patterns implementation. Such system improves the development process of complex design pattern oriented software, because it helps discover communication errors and violations of design patterns implementation earlier.*

1 Introduction

One of the disadvantages of design patterns is the fact that they bring a new complexity into the design. This complexity is caused by the introduction of new classes and interfaces, which are used to provide better flexibility and reusability. Developers often do not have enough time to create a documentation for their classes; therefore, 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 a more abstract view and thus tackle some of the complexity.

Even if the code documentation includes information about implemented patterns, an incorrect understanding of some design patterns by one member of the development team may slow down the development process or even lead to the introduction of software bugs in the system.

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

Such problems are addressed by the Patterns4Net project, the presentation of which is the main aim of this paper. The crucial component of the Patterns4Net is Pattern Enforcer – a tool that verifies selected structural aspects of design patterns. It implements a set of

14 built-in patterns; moreover, users can add their custom patterns using the special API. Pattern Enforcer needs to know which classes are supposed to implement which pattern in order to enforce its correct implementation. For this purpose, the Patterns4Net special documentation for pattern solution participants is used.

The rest of the paper is structured as follows: the patterns representation is described in Section 2. Domain specific languages in general and our API for pattern constraints specification are described in Section 3. Section 4 shows how to use the Enforcer, Section 5 deals with the Patterns4Net Common Infrastructure architecture. Section 6 compares our system with other relevant work and Section 7 summarizes the paper and suggests future work.

2 Representation and Documentation

2.1 Documentation of Pattern Instances

A design pattern is an abstract entity, which, among other things, primarily describes a solution to a recurring problem. If such design pattern is implemented by a developer, he transforms the abstract ideas behind the pattern into a real source code. For example, an instance of the *Composite* pattern is given in Figure 1.

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

Figure 1: Example of the *Composite* pattern instance.

If we consider the example from Figure 1, Pattern Enforcer does not know that the *WidgetComposition* class should implement the *Composite* pattern. Therefore, Pattern Enforcer does not know that it should enforce the structural aspects of the correct implementation of the *Composite* pattern on the *WidgetComposition* class. For this purpose, we need to create a mapping between concrete elements in a source code

and the pattern participants they are supposed to implement. We call this mapping a design patterns participants mapping.

Patterns4Net provides an extensible mechanism for the construction of design patterns participants mapping from .NET assemblies data. At the moment, it supports pattern meta-data expressed as .NET attributes. The class that plays the main role in a particular design pattern implementation is decorated with a special attribute. Classes that implement other roles in the pattern solution can be inserted as arguments of this attribute. For a better illustration, a code example is provided in Figure 2. Here, the `WidgetComposition` class is decorated with the `Composite` attribute, which also allows us to provide the type of a *Component* as a constructor parameter. An explicit specification of a *Component* type is required when a *Composite* class implements more than one interface, otherwise the *Component* type can be inferred automatically.

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

Figure 2: Example of attributes driven documentation of pattern instances.

2.2 Pattern Instances Representation

In order to use the design patterns participants mapping, Patterns4Net needs to have data structures that represent the mapping. For this purpose, standard C# classes are used and instances of these classes represent the instances of design patterns. An object that represents a pattern instance provides a name of the pattern, and the references to the elements that participate in this pattern instance.

The classes for the pattern representation provide the name of the pattern and the references to the pattern instance participants as standard .NET properties. The references to code elements (that are classes, interfaces, methods, etc.) are represented by the instances of Mono Cecil's classes, which are similar to the `System.Type` type from the standard library (e.g. `TypeReference`, [1]). Mono Cecil's types are used, because

we use Mono Cecil for parsing of .NET assemblies¹. Figure 3 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 IPattern interface
    public string Name {
        get { return "Composite"; }
    }
}
```

Figure 3: The *Composite* pattern definition for Patterns4Net.

3 Patterns Structural Constraints Specification

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

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

In the rest of this section we describe the API for the patterns structural constraints specification in more detail. Since this API can be considered as an example of a Domain Specific Language (DSL, [2]) and

¹ Reasons why we have chosen Mono Cecil and more detailed information about it are presented in the subsection 5.1

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

3.1 Domain Specific Languages

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

Type-safe embedded DSLs use constructs that can be verified by a compiler rather than strings with a special internal syntax that can be verified only during the runtime or by an additional tool. For example, NHibernate ORM framework ([3]) has such API for the definition of objects to database schema mapping. Instead of expressing the names of properties as strings, NHibernate exploits the C#'s feature of lambda expressions for this purpose, and thus the existence of the properties used in the mapping is verified by the C# compiler. To give a better idea of this approach, Figure 4 shows a short example of the NHibernate DSL usage in C#. Note that all text in this figure forms a perfectly valid C# code, although it may seem as a special language.

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

Figure 4: Example of type safe DSL embedded into the C# language.

Embedded DSLs usually leverage a technique called Fluent API, which means that a method returns an object on which a user is expected to invoke another method. This chaining of methods can make the API more self describing, because the names of the methods and the names of their arguments can be then read almost as an English sentence. An example of the Fluent API from jMock, a mock object library for Java [4], is shown in Figure 5.

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

Figure 5: Example of methods chaining in Fluent API.

3.2 The API for Pattern Constraints Specification

Since we have a strongly typed representation of design patterns instances, we can build a type safe DSL for their constraints specification, where we use lambda expressions.

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. However, Pattern Enforcer provides a DSL to make the specification of these constraints easier than that. The key part is that it enables to specify the constraints as lambda functions. We call such function a "check".

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

To summarize up, users can select a subject of the check, using lambda functions, and then they can enter the check itself again as a lambda function, which takes the subject of the check as a parameter. For a better idea, an example is shown in Figure 6.

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

3.3 Built-in Patterns

The constraints for the built-in patterns were chosen less restrictively than in other tools of this type. The

```
// 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 perform the following check
.Check((composite, method) =>
    method.CallsAnyOf(composite.Component),
    (composite, method) =>
        "error in " + method.Name));
```

Figure 6: Example of constraints configuration in Pattern Enforcer.

aim was to enforce those aspects that are strongly significant to a given pattern. The implementation without them clearly cannot be called an implementation of this pattern. For example, the *Factory Method* pattern, the main participant of which is the *Factory Method* itself, would make no sense if the actual *Factory Method* was void. On the other hand, enforcing 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*.

The relatively unrestrictive API for specification of patterns constraints allows us to provide a more advanced verification than a mere verification of structural aspects. We illustrate the process of choosing the structural constraints that should be verified by Pattern Enforcer by an example of the *Template Method* pattern.

3.4 Template Method

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

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

from our *template method*; however, it is unsystematic. Instead, a simple observation can help: non-virtual methods that invoke virtual methods are usually also implementation of the *Template Method* pattern. The conclusion is that a *template method* should invoke at least one virtual method or at least one another *template method*.

To declare the *template method* as non-virtual (sealed) is considered a good practice with the *Template Method* pattern and therefore we enforce this as well.

The specification of constraints for the *Template Method* pattern is shown in Figure 7. As a first step we check that the type that declares the *template method* is not sealed and therefore it can be sub-classed. If this is fulfilled, we check that the template method calls at least one virtual method or another template method.

```
// check that declaring type is not sealed:
this.Type(pattern =>
    pattern.TargetMethod.DeclaringType)
    .Check(type => type.IsSealed == false,
        (pattern, type) => "...error message...");
// check that template method invokes at least
// one virtual method or another template method:
this.If(pattern =>
    !pattern.TargetMethod.DeclaringType.IsSealed)
    .Method(pattern => pattern.TargetMethod)
    .Check( method =>
        method.GetMethodCalls().Any(
            call =>
                call.TargetObject != null &&
                call.TargetObject.IsThisParameter &&
                (IsTemplateMethod(call.Method) ||
                    call.Method.IsOverrideable())),
        (pattern, method) => "...error message...");
```

Figure 7: The specification of the built-in *Template Method* pattern.

4 Usage

If a user wants to take advantage of Pattern Enforcer, one possible way to achieve it is to decorate his types with pattern attributes. For this, it is required to add a reference to the *Patterns4Net.Attributes.dll* assembly in the project. This assembly contains only the attributes definitions; thus, its footprint should be minimal. It is built for .NET version 2.0, so Pattern Enforcer can be basically used in projects built for older versions of the .NET. When the reference is added, the types can be decorated with patterns attributes from the namespace *Patterns4Net.Attributes*. Figure 8 contains an annotated implementation of the *Composite* pattern.

```
using Patterns4Net.Attributes;
[Composite(typeof(IWidget))]
class WidgetComposition : IWidget, ICloneable {
```

Figure 8: Example of an annotated implementation of the *Composite* pattern.

The second possible way of taking advantage of Pattern Enforcer does not require annotating classes with pattern attributes. Instead, the relation between a concrete pattern and its roles is constructed by hand in an automatized test. Users can also define their custom patterns using the pattern constraints specification API.

5 Architecture

In this section, the architecture and the implementation of the common Patterns4Net infrastructure and Pattern Enforcer is discussed. We start with Common Intermediate Language (CIL) parsing, because the instruments we use for this task influence the rest of the system. Then we describe design patterns representation and discovery architecture.

5.1 CIL Processing

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

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

The other option, which we have chosen, is to analyze the intermediate language, in case of the .NET it is the Common Intermediate Language (CIL). The structure of CIL is more stable than, for example, the syntax of C#. The latest version of CIL standard [5] from 2010 has the same instruction set as the previous version from 2006. The version from 2010 extends only semantics and verification rules for some of the instructions. Another advantage is that an intermediate language is produced by all the compilers for .NET, and thus Patterns4Net can be theoretically used also for

Visual Basic.NET, IronRuby, IronPython and others, although we have tested it only on C#. One of the disadvantages of this approach is that the CIL is a stack based lower level language and the reconstruction of some constructs, such as actual parameters for a method invocation, requires special effort.

Library for CIL parsing There are three popular, publicly available libraries that could be used to parse .NET assemblies and get meta-data about the types and CIL code of the methods. First option is to use the reflection API that is available as a part of the .NET base libraries. Second option is the Microsoft Common Compiler Infrastructure (CCI, [6]), which is developed in Microsoft Research. Last option is Mono Cecil [1], which is developed as a part of the Mono open-source project.

Standard .NET Reflection API treats assemblies as a code, not as a raw data, which has two important consequences: the code loaded through the .NET Reflection API can be executed; and, because the code can be executed, the runtime must check access rights and might throw Code Access Security exception.

The other two libraries (CCI and Mono Cecil) process .NET assemblies as just binary data, hence they do not support loading the assemblies into an AppDomain and execution of the loaded code. On the other hand, they are claimed by their authors to be faster than the standard Reflection API. However, we are not aware of any serious benchmarks. Public API and features of CCI and Mono Cecil seem to be similar. Our previous experiences with Mono Cecil have resolved the choice between Mono Cecil and CCI in favor of Mono Cecil. This choice does not only influence the code that does the CIL analysis, but it also influences the other code, because we use specific Mono Cecil's data structures (e.g., TypeReference) in the whole Patterns4Net project.

5.2 Patterns Representation and Discovery

Patterns representation is described in Section 2. Here, we just remind that a pattern instance is represented as an object that provides references to the participants of this pattern instance. Mono Cecil's structures are used for types and methods identification.

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

There are two built-in strategies for the pattern participants mapping discovery. Both are based on pattern meta-data (additional information added to a

.NET assembly by its author in order to document patterns he has implemented). In both cases these meta-data are expressed as .NET attributes provided by Patterns4Net. These two strategies differ only in the way they reconstruct the pattern participants mapping from attributes meta-data.

The first one requires the attribute to declare special constructor, which is used to instantiate the attribute itself from meta-data, and then the creation of the pattern instance is left to the attribute object. In this case, the attribute and the pattern instantiation process are coupled in one class.

The other one uses the CIL meta-data to create the pattern instance directly. It means that the attribute itself can be only a dummy data-holder class, which does not actually participate in the pattern instance creation. This approach provides a better flexibility; however, it requires more work to be done.

Other strategies for the discovery of patterns meta-data (e.g., based on naming conventions) can be easily added; therefore, we do not restrict Patterns4Net only to attributes driven documentation of design patterns instances.

5.3 CIL Analysis

Mono Cecil provides only data parsed from .NET assemblies, it does not provide anything more. From CIL meta-data we can, for example, determine for a given class what type is its base type, or which interfaces it implements. However, 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 the Mono Cecil Rocks project, which contains a few extension methods for the Cecil's classes; nevertheless, it does not have all we want to support in Patterns4Net, so we also implemented our custom set of extension methods designed for CIL analysis and patterns structure constrains specification.

For example, one of the extensions we wanted to provide was a uniform API for getting information about methods overrides. In CIL, according to ECMA CIL specification [5], there is an attribute "overrides" in the meta-data of every method, which is a list of methods that this method overrides. This attribute, however, is used only in specific cases (e.g., explicit interface implementation) and normally it is left empty, because overridden methods are determined by conventions described in the ECMA CIL specification.

Methods invocation analysis For the purposes of the discovery of methods invocations in Pattern Enforcer, we needed classes that would help us with the analysis

of CIL. We do not need to analyze conditional statements – we just want to know whether a method M1 on a field F is invoked in body of a method M2, even in a dead branch of code.

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

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

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

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

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

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

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

The main class for CIL instructions sequences matching is the `CILPatternsMatcher`. It aggregates a collection of instances of the `InstructionMatcher` abstract class, which represents an instructions sequence. Interface of the `InstructionMatcher` class is shown in Figure 9. The `Matches` method is called in a loop provided with the current instruction. If the method returns false, then the CIL instructions do not match the expected sequence and the whole process ends with a negative result. Otherwise, property `Found` is checked and if true, then the next `InstructionMatcher` is used in the next iteration; if it was the last `InstructionMatcher`, then process ends with success. In the next iteration the current instruction is set to the one returned by last call of `Match`. A pseudo code is given in Figure 10, variable `matchers` represent an array of instances of the `InstructionMatcher` class.

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

Figure 9: The `InstructionMatcher` abstract class interface.

6 Related Work

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

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

Figure 10: Pseudo code of instructions patterns matching.

6.1 FxCop and Gendarme Tools.

It may not be obvious, but Pattern Enforcer is similar to static analysis bug-hunting tools such as FxCop [7] or Gendarme [8]. These tools search the source code for the idioms that are generally considered as bad. For example, strings should be, in most cases, compared using `string.CompareOrdinal`, but not using `==` operator. There are two main differences between Pattern Enforcer and these tools

- Pattern Enforcer checks only the code that is annotated.
- Pattern Enforcer checks structural aspects and code idioms; however, Gendarme and FxCop check only code idioms.
- Whereas Gendarme and FxCop look for bad idioms, Pattern Enforcer verifies that expected idiom is present.

Gendarme is an open-source tool that is meant to be an alternative to FxCop. It uses Mono Cecil for CIL analysis. 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`.

6.2 Pattern Enforcing Compiler (PEC) for Java

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

For the annotation of patterns instances, PEC uses marker interfaces, which can only be used for annotation of classes, but not methods, and even when interfaces can have arguments (generic arguments), these

can capture only a limited number of additional information. The authors of PEC admit these weaknesses of interfaces as a technique for the annotation of patterns and in [9], they propose the standard java annotations, similar to .NET attributes, in PEC. However, we are not aware of any updated version of PEC that uses standard Java annotations.

PEC uses a static analysis and it also enforces the rules dynamically by inserting assertions into the resulting program, which we do not support in our Pattern Enforcer. Dynamical enforcement provides more accurate results. On the other hand, dynamical enforcement slows down the resulting program and the program still has to be manually tested in order to discover possible bugs.

6.3 Other Tools

CoffeeStrainer [10] is a tool that is somewhere between static analysis bug-hunting tools whose objects of interest are idioms, smaller pieces of code, and pattern enforcement tools. Unlike other static analysis bug-hunting tools CoffeeStrainer enforces rules that result from particular design decisions; for this it provides means for custom rules specification. CoffeeStrainer targets the Java platform.

Pattern-Lint [11] can check conformance to a 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 [12] 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 (Prolog-based solution for design patterns formalization and verification) and tools that come with LePUS3 (visual approach to formalization).

We can conclude that we are not aware of any design patterns verification tool for the .NET platform. Pattern Enforcer is, among all of these tools, also extraordinary with its special C# API for structural constraints specification, because most of the other approaches use special language for patterns formalization, or, in case of the Pattern Enforcing Compiler for Java, they do not provide special means for structural constraints specification at all.

7 Conclusions

The aim of this work was to explore existing approaches for design patterns support in development environ-

ments and to present the Patterns4Net project. With Patterns4Net users can explicitly document their intent to implement a particular design pattern. Pattern Enforcer is able to verify most of the structural aspects of design patterns. There are 14 built-in patterns (e.g., *Singleton*, *Visitor*), but custom patterns can be added using special API for specification of their structural constraints.

Future versions of Pattern Enforcer should support more enhanced features. Some of the more general rules from Pattern Enforcer, such as the immutability check, could be extracted from its source and proposed to open-source community as additional rules for well-established open-source project Gendarme.

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

Acknowledgements

This work was supported by the grant SVV-2010-261312.

References

1. "Cecil – mono." <http://mono-project.com/Cecil>, May 2011.
2. M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010.
3. "NHibernate forge." <http://nhforge.org>, May 2011.
4. "jmock - an expressive mock object library for java." <http://www.jmock.org/>, May 2011.
5. T. Ecma, "Tg3. common language infrastructure (cli). standard ecma-335," 2010.
6. "Common compiler infrastructure: Metadata api." <http://ccimetadata.codeplex.com/>, May 2011.
7. "Fxcop." <http://msdn.microsoft.com/bb429476.aspx>, Aug. 2010.
8. "Gendarme." <http://mono-project.com/Gendarme>, May 2011.
9. H. Lovatt, A. Sloane, and D. Verity, "A pattern enforcing compiler (pec) for java: A practical way to formally specify patterns," 2007.
10. B. Bokowski, "CoffeeStrainer: statically-checked constraints on the definition and use of types in java," in *Software Engineering/ESEC/FSE99*, pp. 355–374, Springer, 1999.
11. 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.
12. T. Taibi, *Design patterns formalization techniques*. Igi Global, 2007.