A design pattern is a description of communicating objects and classes that are customized to solve a general design problem in a particular context. The main aim of patterns in object-oriented design is to make the design reusable and flexible. This is very important because frequent changes in the functional requirements are usual nowadays.

Patterns4Net provides two tools:

- **Pattern Enforcer** verifies some of the structural aspects of selected design patterns implementation. For example, it verifies if a user "zooms" to a particular class, all related classes, even infrastructural, are displayed. Which elements are displayed and which are not, is chosen according to design patterns they implement. For instance, the NullObject pattern represents rather implementation detail. Users can browse the diagrams in an interactive graphical user interface.

- **Architecture Explorer** generates interactive UML-like class diagrams from .NET assemblies. Instead of a one large diagram with lots of unnecessary infrastructural classes, it uses the information about implemented design patterns to create more diagrams with different levels of abstraction. For example, the top level view shows only the high level, domain specific classes that are important for understanding of the overall architecture. However, it does not provide any special API or language embedded into the C# language. We believe this approach is easier to use than most of the other formalization techniques that use special languages.

**Design Patterns**

Keep in mind that the diagrams with different levels of abstraction. For example, the top level view shows only the high level, domain specific classes that are important for understanding of the overall architecture. However, if a user "zooms" to a particular class, all related classes, even infrastructural, are displayed. Which elements are displayed and which are not, is chosen according to design patterns they implement. For instance, the NullObject pattern represents rather implementation detail. Users can browse the diagrams in an interactive graphical user interface.

**Patterns4Net**

The main conception behind the Patterns4Net is that developers will annotate their code using .NET attributes mechanism and the Patterns4Net will provide tools that will take advantage of this documentation and will support the development process of design patterns oriented software.

The main source of inspiration for Pattern Enforcer was Pattern Enforcing Compiler for Java (PEC). It is extended Java compiler that verifies design patterns. For patterns annotation PEC uses marker interfaces. This technique has few drawbacks (e.g., methods cannot be annotated with interfaces), which the authors of PEC have admitted and they planned to support Java annotations (similar to .NET attributes used in our Enforcer) in the next version. We are, however, not aware of any updated version of PEC with Java annotations. PEC also does not provide any special API or language for custom patterns specification as we do.

**Existing Approaches**

**Misunderstanding**. Even well documented patterns can be misunderstood, which can slow down the development process or even lead to an introduction of software bugs in the system.

**Related Work**

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

Pattern Enforcer can be integrated into the Visual Studio build process or used as a stand-alone command line program.

Acknowledgment.

**Conclusion**

Pattern Enforcer is a tool that verifies correctness of selected structural aspects of design patterns implementations. For example, it verifies that all elements that can be visited by the Visitor instead override and correctly implement the Accept method, which is essential for the correct implementation of this pattern, but unfortunately not enforced by the standard compiler. There are 14 built-in patterns, but users can also add their custom patterns or even simple idioms using the special type-safe domain specific language embedded into the C# language. We believe this approach is easier to use than most of the other formalization techniques that use special languages.