UNIVERSITÄT PADERBORN
*Die Universität der Informationsgesellschaft*

Heinz Nixdorf Institute & Department of Computer Science
Software Engineering Research Group
Prof. Dr. Wilhelm Schäfer
Warburger Straße 100
33098 Paderborn

# Specification of Software Patterns for Pattern-Oriented Software Development:

# **Meta-model**

# 1 Preface

The described meta model below is rather a strong recommendation than a limitation. Changes are possible, but since this meta model is the interface between the DSD project and the related student project *PG POSE* in Paderborn, you should not modify it on your own. Please let us know if you have any suggestions to enhance the meta model or adapt the meta model to your own requirements.

Although this document takes GoF design patterns [GHJV95] as examples for explaining the meta-model, the meta-model is not limited to those. The meta-model should cover other types of patterns, like architectural patterns (e.g. [BMR+96]) and also different catalogs of software patterns (e.g. J2EE Patterns [ACM03]). For further reading check the bibliography at the end of this document.

# 2 Meta-model

The following section describes the meta-model of the framework for managing patterns. The meta-model defines the possible structure of the framework and will serve as a basis for the generation of the Ecore data in the EMF. The meta-model itself is shown in figure 1.

The class `PatternCatalog` is the root element in the meta model. A `PatternCatalog` has a name, a description and contains an arbitrary number of `Patterns`. For example, there could be a `PatternCatalog` named *"Gang of Four patterns"*, which contains the 23 design patterns from [GHJV95].

The class `Pattern` which is contained in a `PatternCatalog` represents a single pattern. A `Pattern` has a name and an arbitrary number of `PatternVariants`, which are described below. For example, a `Pattern` could represent the design pattern *Observer* which is included in the `PatternCatalog` *"Gang of Four patterns"*.

To enhance the organization of a pattern catalog, patterns can be sorted into categories. For that purpose a `PatternCatalog` contains an arbitrary number of `Categories`. A `Category` has a name and an arbitrary number of `subCategories`. A `Category` refers to an arbitrary number of `Patterns`, a `Pattern` refers to an arbitrary number of `Categories`. In the `PatternCatalog` *"Gang of Four patterns"*, there could be, for example the three `Categories`, which reflect the purpose of a pattern: creational, structural, or behavioral patterns. Each purpose category could be separated into subcategories which reflects the scope of a pattern: classes or objects.

Furthermore, patterns can be organized by using keywords. For that purpose, a `PatternCatalog` contains an arbitrary number of `Keywords`. A `Keyword` has a name and refers to an arbitrary number of `Patterns`. A `Pattern` refers to an

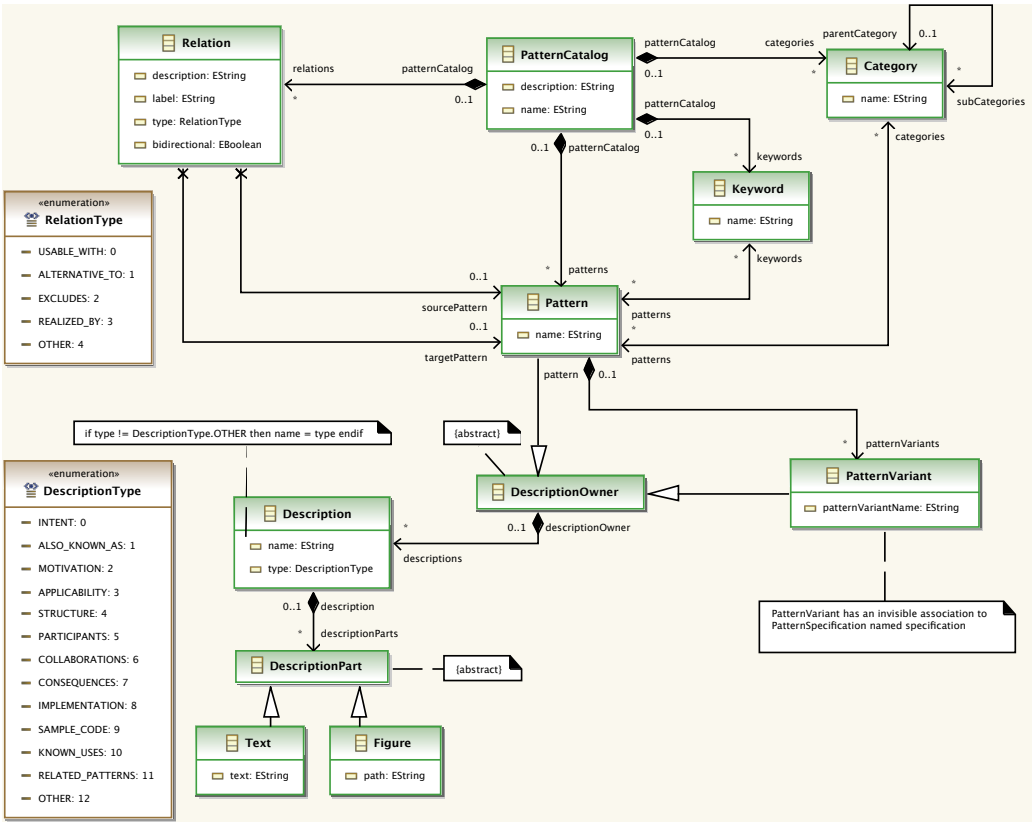


Figure 1: Meta-model of the framework for managing software patterns

arbitrary number of `Keywords`. An example for `Keywords` could be *"tree structure"* and *"algorithm encapsulation"* for the design pattern *Visitor*.

As described a design pattern is typically related to other patterns. To model such relations between `Patterns`, a `PatternCatalog` contains an arbitrary number of `Relations`. A `Relation` has a label, a description and a type. The type is specified by the enumeration `RelationType` (see chapter 2.1). A relation refers to a `sourcePattern` and a `targetPattern`. The attribute `bidirectional` describes whether the relation is bidirectional or unidirectional. For a relation example, the design pattern *Iterator* is often applied to data structures like the design pattern *Composite*.

Since patterns are described informally, there is an undefined number of pattern variants which realize a single pattern. For that purpose a `Pattern` contains an arbitrary number of `PatternVariants`. A `PatternVariant` has a name and refers to a concrete `PatternSpecification`[1]. For an example, one variant of the classic *"Visitor"* pattern is a reflection based implementation. The class `PatternSpecification` is an element from the meta model of the project group *PG POSE* of Paderborn. The association from `PatternVariant` to `PatternSpecification` is the link to the formal specification of the pattern variant. A `PatternSpecification` can be referenced by an arbitrary number of `PatternVariants`.

`Patterns` and `PatternVariants` are typically described in detail. For example, each GoF-pattern is described on about 10 pages in [GHJV95]. Each pattern description is separated in several sections, *"Intent"* or *"Motivation"* for example. For that purpose, a `DescriptionOwner` can contain an arbitrary number of `Descriptions` which represent such a section. Both, `Pattern` and `PatternVariant` are `DescriptionOwners`, thus, they can be described in detail using `Descriptions`.

A `Description` has a type, which is specified by the enumeration `DescriptionType` (see chapter 2.2) and a name. If the type is not equal to `Description.OTHER`, the name of the description should be similar to the type of the description. A `Description` contains an arbitrary number of ordered `DescriptionParts`. A `DescriptionPart` represents a paragraph of the description. A `DescriptionPart` can be a `Text` or a `Figure`.

The attribute `text` of the class `Text` can contain plain text, HTML text or any other type of text. In fact the `DiscriptionParts` are discussable elements of the meta model which could be adapted to your own requirements. For example, new `DiscriptionParts` or a type attribute to distinguish various text types (e.g. plain text or code fragments) are possible as well as the definition of only one `DiscriptionPart` which contains the whole description as HTML code.

[1] The class `PatternSpecification` is invisible in figure 1

## 2.1 Enumeration RelationType

The enumeration `RelationType` represents the different relations between two patterns. The different types are described below.

**USABLE_WITH** Describes that the source pattern can be used in combination with the target pattern. For example, the design pattern *Abstract Factory* can create and configure a particular *Bridge*.

**ALTERNATIVE_TO** Describes that the source pattern is an alternative to the target pattern. For example, the design pattern *Abstract Factory* is similar to the design pattern *Builder*, because it may also construct complex objects.

**EXCLUDES** Describes that the source pattern excludes the application of the target pattern. For example, the design patterns *Prototype* and *Abstract Factory* are competing patterns in some ways.

**REALIZED_BY** Describes that the sources pattern is realized by using the target pattern. For example, `AbstractFactory` classes from the design pattern *Abstract Factory* are often implemented with factory methods from the design pattern *Factory Method*.

**OTHER** The default value, which can be used when no other type fits. In this case, label and description attributes of the `Relation` class have to adequately describe the relation.

## 2.2 Enumeration DescriptionType

The enumeration `DescriptionType` represents the different types of sections to describe a pattern or a pattern variant. The different types are described below.

**INTENT** Briefly introduces the pattern, its use, its intent and the particular design issue or problem the pattern is addressed to.

**ALSO_KNOWN_AS** Describes other well-known names for the pattern, if any.

**MOTIVATION** Describes a scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.

**APPLICABILITY** Describes the situations in which the pattern can be applied and how such situations can be recognized.

**STRUCTURE** Provides a graphical representation of the classes in the pattern by typically using a UML diagram

**PARTICIPANTS** Describes the classes and/or objects participating in the pattern and their responsibilities.

**COLLABORATIONS** Describes how the participants collaborate to carry out their responsibilities.

**CONSEQUENCES** Describes the trade-offs and results of using the pattern.

**IMPLEMENTATION** Describes pitfalls, hints, or techniques which should be considered when implementing the pattern.

**SAMPLE_CODE** Describes code fragments that illustrate how the pattern could be implemented.

**KNOWN_USES** Describes examples of the pattern found in real systems.

**RELATED_PATTERNS** Describes which design patterns are closely related to this one and what their differences are.

**OTHER** The default value, which can be used when no other type fits. This way, new description types can be added by the user, description name has to adequately describe the section.

# 3 Implementation

The implementation of the described meta model will be provided and maintained by the german part of the DSD project. The implementation will be given as an Eclipse plug-in.

The meta model is modelled and implemented using the Eclipse Modeling Framework (EMF)[2] . EMF is a modeling framework, which supports the code generation based on a structured data model. The generated code which is implementing the meta model must not be modified (see chapter 1). On top of this meta model implementation the pattern management plugin can be build.

[2]Eclipse Modeling Framework Project (EMF) online at: http://www.eclipse.org/modeling/emf/

## References

[ACM03] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns – Best Practices and Design Strategies*. Core Design Series. Prentice Hall, Sun Microsystems Press, 2 edition, 2003.

[BHS07a] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing*, volume 4 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2007.

[BHS07b] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture – On Patterns and Pattern Languages*, volume 5 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2007.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*, volume 1 of *Software Design Patterns*. John Wiley and Sons, Ltd, 1996.

[CS95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*, volume 1 of *Software Patterns Series*. Addison-Wesley, 1995.

[Fow99] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[Han07] Robert S. Hanmer. *Patterns for Fault Tolerant Software*. Software Design Patterns. John Wiley and Sons, Ltd, 2007.

[HFR99] Neil Harrison, Brian Foote, and Hans Rohnert, editors. *Pattern Languages of Program Design*, volume 4 of *Software Patterns Series*. Addison-Wesley, 1999.

[KP04] Michael Kircher and Jain Prashant. *Pattern-Oriented Software Architecture – Patterns for Resource Management*, volume 3 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2004.

[MRB97] Robert Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design*, volume 3 of *Software Patterns Series*. Addison-Wesley, 1997.

[MVN06] Dragos Manolescu, Markus Voelter, and James Noble, editors. *Pattern Languages of Program Design*, volume 5 of *Software Patterns Series*. Addison-Wesley, 2006.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2000.

[VKC96] John Vlissides, Norman Kerth, and James Coplien, editors. *Pattern Languages of Program Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.