Abstracting and Generalising with Design Patterns

Mária SMOLÁROVÁ, Pavol NÁVRAT and Mária BIELIKOVÁ

Department of Computer Science and Engineering,
Slovak University of Technology,
Ilkovičova 3, 812 19 Bratislava, Slovakia
smolarova@dcs.elf.stuba.sk, {navrat, bielik}@elf.stuba.sk

Abstract. The paper proposes a technique that allows representing design patterns in a way suitable for design of an application. The paper analyses two important aspects of design patterns namely their levels of abstraction and generality. We aim at identifying their difference and consequently, at proposing a way to express them. Moreover, the role of abstracting and generalising in the design process is more precisely recognised. We propose to describe (some essential aspects of) design patterns in a space with two dimensions: abstraction and generality. Next, we describe a technique of modelling design patterns by means of description schemata at a metalevel. A metaschema represents pattern relevant elements and their relations. Also, constraints are introduced that restrict the possible structure of pattern instances.

1. Introduction

Design patterns are a technique envisaged to support development and maintenance of software systems. They help reduce the complexity of solving many real-life problems. Thus they are essentially abstractions of concrete design steps which presumably belong to wealth of design experience accumulated by the practitioners of the field. They constitute a set of rules describing how to accomplish certain tasks in the realm of software development [11]. The purpose of abstracting is to hide many unessential details that are necessarily involved in elaborating and subsequently implementing any design. Mere abstracting, however, yields only a high level abstract description of a specific domain solution with low-level decisions and implementation details abstracted away. Once we have this, we want more. We want the design pattern, which has proven to be a successful solution to one specific problem in one application domain to be useful for other problems and possibly in other domains, too. In other words, we need the design pattern to be general as well. These two aspects, viz. abstraction and generality often lead to much confusion. We believe the issue is worth discussing from many points of view, with one particular being design pattern formulation and representation.

The usual way of formulating design patterns (cf., [5]) relies on a combination of diagrammatic description of some aspects of the pattern's structure and on a textual description organised into a loose structure. For a user who wishes to learn the essence of the particular design pattern, this seems to be – at least according to the current state of development of the discipline – a satisfying way of formulation.

From another point of view, it cannot serve as a basis for a mechanical manipulation of software design documentation. Software design is undoubtedly one of the crucial phases of

software development. For the actual design, it is important to make use of as much of standardised design knowledge (embodying verified experience) as possible, because ideally, it would decrease cost of and increase confidence in the designed software. For software maintenance, documentation about design decisions can be very useful. For software system reverse engineering, recorded traces of design patterns applied in the original design would make the job easier. For software system reengineering, similar consideration is valid.

Because design pattern is a quite complex notion, to devise a way to represent them is not a straightforward task at all. We want to concentrate on those aspects that we find especially important from the above outlined point of view. Design patterns, when applied in the design process, are involved in a series of transformation steps. In the transformations, the design undergoes changes – among other things – across a series of abstraction and generality layers. We want to support these transformations by offering a suitable representation of the relevant aspects of design patterns.

We note there are other interesting aspects of design patterns, such as their structural level (e.g., components, frameworks, patterns, and architectures) or an underlying programming paradigm.

The rest of the paper is structured as follows. In Section 2, we introduce shortly design patterns as presented in a known catalogue [5]. In Section 3, we discuss some problems with design pattern representation. In Section 4, we describe the technique of modelling design patterns by means of an example. In Section 5, related work is given. We conclude with suggestions for future work.

2. Catalogue of Design Patterns

Design patterns capture expertise of designers. They describe commonly recurring structures of communicating components that solve general design problems within particular context [5]. Patterns provide solutions that have been developed and evolved over time and are proven to be useful in design of several systems. Design patterns are presented in a fixed format. Pattern name, context, problem, solution, along with example, structure, dynamics, implementation, variant, known uses, consequences, and a reference to related patterns are the sections used for pattern formulation in GoF catalogue [5]. Each pattern occupies several pages of natural language descriptions accompanied by diagrams and source code samples.

As an example that will be used throughout the paper, we show the Abstract Factory design pattern [5]. Main goal of the Abstract Factory pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes.

In the catalogue, conventional notations like OMT class diagrams [12] are used to describe static structure of pattern participants. Class diagram expressing the structure of the Abstract Factory pattern is shown in Figure 1.

Alternatively to class diagrams, object diagrams and object interaction diagrams help explain pattern's run-time behaviour. By using these standard notations to better reveal a pattern, some typical pattern constructs cannot be expressed completely or unambiguously. A pattern can have for example multiple occurrence of a particular component but this fact is not indicated in the class diagram. For example, the number of `ConcreteFactory`s is two in the class diagram in Figure 1 but the catalogue assumes that the pattern in fact allows the number of `ConcreteFactory`s to be "any".

The main difficulty causing that class diagrams are unable to express design patterns appropriately and sufficiently is that class diagrams are meant to model the specific structure of classes that results in concrete program. Patterns are descriptions of a set of

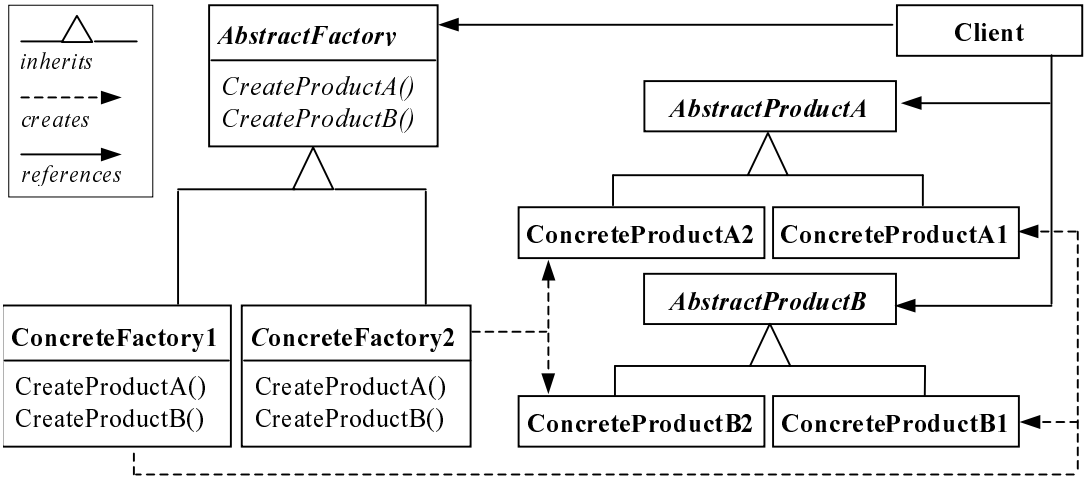

Figure 1. Class diagram expressing the structure of Abstract Factory pattern.

possible structures, i.e. they may result in many possible structures. The lack of class diagram's expressiveness is in the catalogue substituted by giving examples, by code samples, and by informal textual descriptions. The pattern reader is expected to retrieve pattern essence not only from the pattern structure but also from other parts. We are persuaded a representation that expresses design pattern essence more appropriately is necessary. Several efforts to formalize patterns that have been made recently [1], [3], [6], [8] reflect this necessity as well.

3. Design Pattern Space

Before attempting to propose pattern representation, we examine the nature of design patterns with respect to abstraction and generalisation.

In the vast literature on design patterns, one can find ambiguous judgements on their properties. In [6], design patterns are regarded to be abstractions over programs. In [4], the notions of abstraction and generalisation have been confused, as is apparent from the quote "the solution may be generalised into an abstraction". Obviously, result of the process of generalising is a generalisation. Similarly, abstracting results in a higher abstraction.

Object-oriented design patterns are abstractions above the class level. Notion of design patterns allows us to view collaborating classes as a unit with known structure and behaviour. Pattern exists at different abstraction levels in software development. In the earlier development stages, participating classes and their relations are given. Later, the abstraction level descends further by adding more details to pattern participants. A possible differentiation between abstract and concrete patterns may be that the latter are implementations of the former. A pattern is becoming more concrete if it includes methods that are not only abstract, i.e. specified but without a body with a program code, but also implemented i.e., concrete.

However, there are also aspects of design patterns that are more appropriately understood when brought into correspondence with the generality dimension. One way of specialising the design pattern is through renaming the general pattern elements by application specific ones. This results in a domain specific design pattern. When there is no record about pattern presence and about roles played by particular classes or methods in a pattern, pattern maintenance in software development is difficult [7] or even impossible. The situation is further complicated by the fact that frequently, several patterns are used in designing a system. How to combine their application in a systematic way is an important related issue. However, we shall not investigate it in this paper.

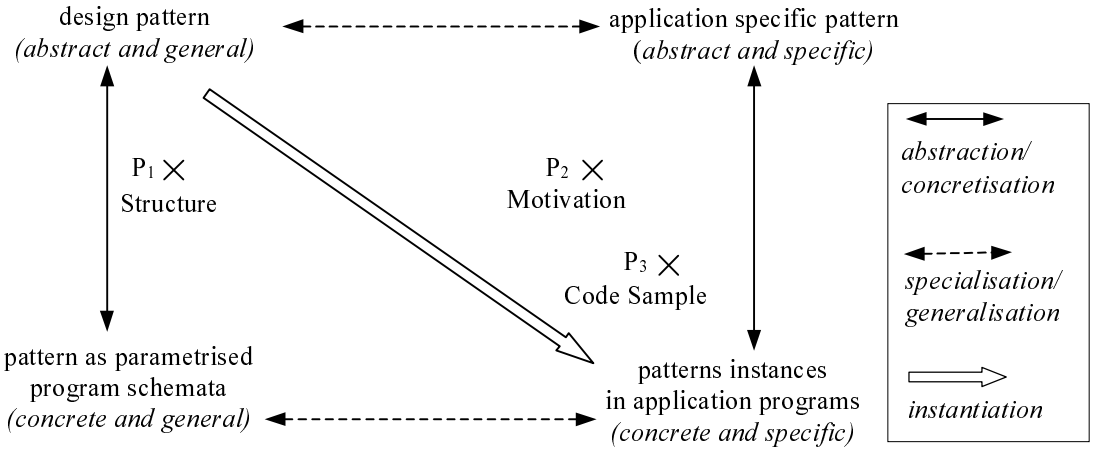

Figure 2. Design pattern space.

Another way of specialising general pattern structure is by giving values to pattern parameters. *Pattern parameters* are those parts of pattern structure that are left flexible. Referring to our sample pattern, there can be several `ConcreteFactory`s. Number of `ConcreteFactory`s is the parameter of the pattern which is supposed to be "any" at the general level, and which can be specialised for example to the value of two at the next lower level of specialisation. Our recognition of the explicit role of parameterisation parallels the endeavours to propose and develop new paradigms, e.g. generative programming [2].

An important aspect of pattern parametrisation is that specific value of one parameter may impose constraints on other parameters. When considering our example, the number of `ConcreteFactory` classes must correspond to the number of `ConcreteProduct` classes for each `AbstractProduct`. In the GoF catalogue, constraints are explicitly formulated neither in textual nor in graphical sections of pattern description. We are convinced that it is extremely useful to relate explicit constraints to the general pattern structure, as they restrict specific pattern structure to obey pattern regularities. Constraints are helpful in transforming a general and abstract pattern into a more concrete and more specific one.

Each design pattern occurs at different abstraction and generality levels during software development process. With respect to generality and abstraction dimensions, the term design pattern covers a space that is depicted in Figure 2. Four most significant states of pattern are:

- general and abstract pattern (in the top left corner) – in this state, pattern is just a named abstraction that solves a general, i.e. domain independent, design problem;
- application specific pattern (in the top right corner) – general pattern becomes specific by embedding it in an application domain; .
- pattern as parameterised program schema – an abstract pattern is transformed into a more concrete structure;
- pattern instances in application programs – concrete and specific pattern instances that live in code.

The process of software development moves the design pattern in four possible directions [9]: concretisation, abstraction, specialisation, and generalisation. While in pattern mining, i.e. discovering useful solutions and recording them as patterns, generalisation and abstraction are mostly involved, during pattern application the principal movement goes from a general and abstract pattern towards a more concrete and more specific one.

Transforming the general and abstract pattern into more specific and more concrete instances is called *pattern instantiation*. It is a vector that combines a partial transformation

from abstract to concrete, and a partial transformation from general to specific. This process leads eventually to a concrete and specific pattern instance embraced in code.

From the generality and abstraction points of view, the catalogues of design patterns explain pattern essence at different abstraction and/or generality levels (see also Figure 2). The pattern structure expressed by class diagrams (shown as point P_1 in Figure 2) concretises the pattern but it unfortunately does not preserve pattern's original generality because it assigns specific numbers to pattern parameters. The motivation example (point P_2 in Figure 2) is even further specialised because it gives an application specific pattern structure. Code samples (point P_3 in Figure 2) decrease both generality and abstraction level by binding to specific implementation language and conveying concrete implementation details.

Our aim is to formulate pattern structure at its very general level. We are convinced that an appropriate pattern representation should be able to reflect different generality and abstraction levels a pattern may exist at.

4. The Design Patterns Representation Technique

In this section, a technique for representing patterns with respect to its abstractness and generality is introduced. Our objective is to propose a representation that expresses pattern as a set of possible specific pattern structures that may exist at different abstraction levels.

In our approach, patterns are represented as a metaschema at the general level. A metaschema depicts pattern relevant elements and their relations. Constraints may be attached to a pattern metaschema in order to express dependencies that must be preserved. All specific instances that are derived from such a metaschema are supposed to satisfy pattern constraints.

Basic elements of the pattern metaschema are depicted by a rectangle. In case of object-oriented patterns, basic elements of metaschema are classes and methods that are named after the roles they play in the pattern. It is in principle possible to extend metaschema so that the basic elements express also less abstract details, for example method parameters or return value of methods. Because of space limitations, less abstract elements are omitted here.

Each metaschema contains also one special element, so-called pattern element. The pattern element corresponds to the abstract state when pattern is considered as a unit. At specific level, pattern element records the presence of pattern instance. The most relevant elements are directly related to pattern element.

Relations between pattern elements are depicted by a rhombus. The relation has its name, maximal and minimal cardinality. Those relations that have cardinality one to many (1:N) express the pattern genericity, with N being a pattern parameter. Possible dependencies among parameters values are expressed by pattern constraints.

In Figure 3, a metaschema for the Abstract Factory pattern is shown. `AbstractFactory`, `ConcreteFactory`, `AbstractProduct`, and `ConcreteProduct` are the metaschema elements that correspond to principal classes in the pattern. `AbstractFactoryPattern` is the pattern element.

In the pattern structure from the GOF catalogue (Figure 1), the relation between `AbstractFactory` and `ConcreteFactory` is shown as generalise (inheritance) relationship. The purpose of the `AbstractFactory` class is to declare a common interface that is implemented by each `ConcreteFactory`. Obviously, this is an abstract – concrete relation when method's implementation is abstracted away in the abstract class. In our metaschema, `AbstractFactory` and `ConcreteFactory` are related through *is_concretised* relation. with cardinality one to many because many `ConcreteFactory`s may be derived.

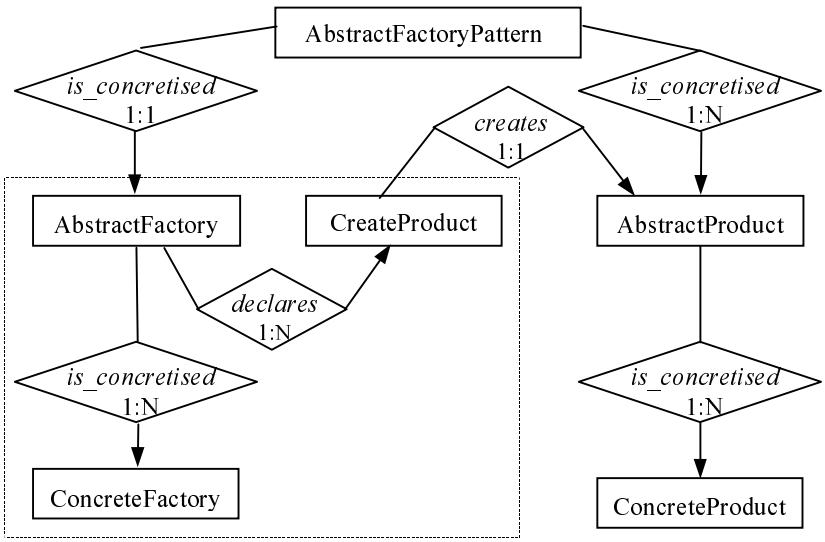

Figure 3. Metaschema for the Abstract Factory pattern.

To define a family of classes with identical interfaces is a common theme in design patterns that helps reduce implementation dependencies in object-oriented design [5]. In Figure 3, we encapsulated the family of Factory classes by a dashed line rectangle. All classes in the family respond to the same interface consisting of a number of `Create-Product`'s methods. The interface is declared in the `AbstractFactory` class and must be implemented in potentially many `ConcreteFactory`s. Correct instantiation of this part of metaschema is expressed by the first Abstract Factory constraint:

(1) *for each ConcreteFactory*
cardinality_of(AbstractFactory.CreateProduct) =
cardinality_of(ConcreteFactory.CreateProduct)

Similar constraint may be attached to many patterns in the catalogue, for example to Observer, Bridge, Prototype, etc.

In the Abstract Factory metaschema, the dependencies among pattern parameters can be expressed by the following constraints:

(2) *for each AbstractProduct*
cardinality_of(AbstractFactory.is_concretised) =
cardinality_of(AbstractProduct.is_concretised)

(3) *cardinality_of(AbstractFactoryPattern.is_concretised) =*
cardinality_of(AbstractFactory.declares)

The constraint (2) forces to keep equality between the number of `ConcreteFactory`s and the number of `ConcreteProduct`s for each `AbstractProduct`. According to constraint (3), the number of `AbstactProduct`s must correspond to the number of `CreateProduct` methods in the family of Factory classes.

One possible instance of the Abstract Factory pattern is depicted in Figure 4. In this case, specialisation by assigning the values to pattern parameters as well as by embedding the pattern into the user interface toolkit application domain has been done. All application specific elements, for example `MotifWidgetFactory`, `Window`, `PMScrollBar`, `CreateWindow`, are typed after the roles they play in the pattern. Relations are depicted uniformly in the diagram. The kind of relation is determined by its name. Its specific multiplicity is explicitly stated.

Constraints that are attached to the general metaschema allow checking structural

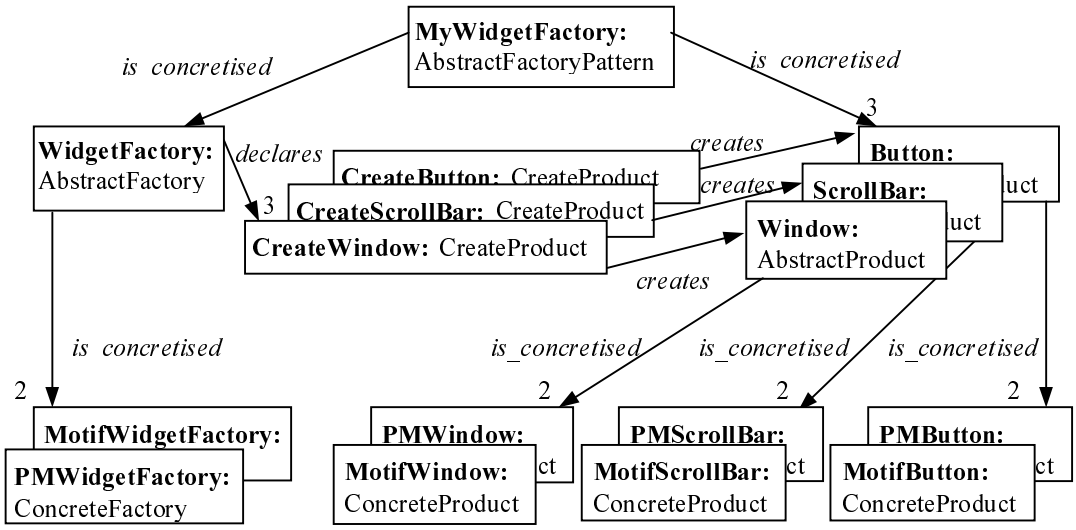

Figure 4. A possible specialisation of Abstract Factory pattern.

dependencies of metaschema's specific instances. For the Abstract Factory pattern instance in Figure 4, the number of `AbstractProduct`s is assigned to three and it must correspond to the number of `CreateProduct` methods. Otherwise, pattern regularities would not kept. Also in case that the application undergoes modification and the number of `AbstarctProduct`s changes, the pattern constraints ensure that the number of methods for product creation corresponds to the number of `AbstarctProduct`s.

5. Related Work and Conclusions

The paper proposes a technique that allows representing design patterns in a way suitable for design of an application. The paper analyses two important aspects of design patterns namely their levels of abstraction and generality. We aim at identifying their difference and consequently, at proposing a way to express them.

We describe design patterns in a space with two dimensions: abstraction and generality. A new technique of modelling design patterns by means of description schemata at a metalevel is introduced. A metaschema is proposed to represent pattern relevant elements and their relations. Constraints that restrict the possible structure of pattern instances are attached to the metaschema.

Our approach is quite general with respect to what kind of components the metaschema elements represent. Throughout this paper, we have shown how the metaschema describes object-oriented patterns: metaschema elements are classes and methods. However, it is in principle possible to represent any collaborating elements by a metaschema.

One of the primary motivations for representing patterns is to have an appropriate tool letting to represent the system also in terms of design patterns. Tools that assist in software development with patterns have been discussed in several papers. In [11], a unified representation describing patterns as the "hook&template" metapattern has been proposed.

In [1], a tool that supports the specification of design patterns and their realisation in a given program is presented. This metaprogramming approach to pattern representation is quit different from ours. Design patterns are understood as a sequence of steps that must be carried out in pattern application. A precise method of specifying how a design pattern is applied by phrasing it as an algorithm in a meta-programming language is proposed.

Another tool supporting pattern instantiating is described in [3]. It has a repository of pattern prototypes, each representing the pattern as a set of fragments that fulfil a particular

role for the pattern. Pattern is a tree with all fragments related to its root. Examples of pattern fragments are classes, methods, attributes, association relations, containment relations, and inheritance relations. The difference to our approach is that even the most fine-grained pattern fragments are linked to the root fragment regardless of their abstraction degree.

In [8], a design pattern is represented as metaschema, so-called primal schema. Primal schema defines roles and relationships of classes. It consists of a basic set of abstract classes and their relationships capturing the essence of the pattern. In this sense, their approach is very similar to ours. The main difference is that the possibly multiple occurrence of pattern's participants is not taken into account. Also, constraints are not embraced.

The proposed technique for design pattern representation will be further developed. We are working on defining the diagrammatic language in more detail with respect to both its syntax and semantics. Here, it might be productive to consider Unified Modeling Language [15] as a basis for possible enhancements.

Acknowledgements

The work reported here was supported by Slovak Science Grant Agency, grant No. G1/4289/97.

References

[1] A.H. Eden, J. Gil and A. Yehudai, Precise Specification and Automatic Application of Design Patterns. In: Proc. of the 12th IEEE International Automated Software Engineering Conference ASE'97, 1997.

[2] U.W. Eisenecker, Generative Programming with C++. In: Proc. of Modular Programming Languages JMCL'97, H. Mössenböck (ed.), Springer Verlag, LNCS 1301, 1997, pp. 361-365.

[3] G. Florijn, M. Meijers and P. vanWinsen, Tool Support for Object-Oriented Patterns. In: Proc. ECOOP'97, Jyväskylä, M. Aksit, S. Matsuoka (eds.), Springer Verlag, LNCS 1241, 1997, pp. 472-495.

[4] J. Gil and D.H. Lorenz, Design Patterns and Language Design. *IEEE Computer* **31** (March 1998) 118-120.

[5] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, ISBN 0-201-63361-2, 1994.

[6] E.E. Jacobsen, Design Patterns as Program Extracts. In: Object-Oriented Technology, ECOOP'97 Workshop Reader, Workshop on Language Support for Design Patterns and Frameworks, Jyväskylä, J. Bosh and S. Mitchell (eds.), LNCS 1375, Springer Verlag, 1997.

[7] C. Kramer and L. Prechelt, Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Proc. Working Conference on Reverse Engineering, Monterey, IEEE CS Press, 1996, pp. 208-215.

[8] T.D. Meijler, S. Demeyer and R. Engel, Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment. In: Proc. of ECES/FSE'97, M. Jazayeri and H. Schauer (eds.), LNCS 1301, Springer Verlag, 1997, pp. 94-110.

[9] P. Návrat, A Closer Look at Programming Expertise: Critical Survey of Some Methodological Issues, *Information and Software Technology* **1** (1996) 37-46.

[10] B.U. Pagel and M. Winter, Towards Pattern-Based Tools. In: Proc. of EuroPLoP'96, Kloster Irsee, Germany, July 1996.

[11] W. Pree, Meta patterns – a means for capturing the essentials of reusable object-oriented program execution. In Proc. of ECOOP '94; Bologna, M. Tokoro and P. Remo (eds.), LNCS 821, July 1994, pp. 150-162.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Object-Oriented Modeling and Design. Prentice Hall, 1991.

[13] M. Smolárová and P. Návrat, Software Reuse: Principles, Patterns, Prospects, *Journal of Computing and Information Technology* **1** (1997) 33-48.

[14] M. Smolárová, P. Návrat and M. Bieliková, A Technique for Modelling Design Patterns. In: P. Návrat and H. Ueno (eds.), Knowledge-Based Software Engineering, IOS Press, Amsterdam, 1998, pp. 89-97.

[15] UML, Unified Modeling Language, version 1.1. UML Summary, UML Semantics, UML Notation Guide, Rational Software Corp., 1997, `http://www.rational.com/uml`.