

# Reuse with Design Patterns: Towards Pattern-Based Design

Mária SMOLÁROVÁ, Pavol NÁVRAT

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

**Abstract.** Two possible approaches to reuse with design patterns (DPs) are described: pattern-based design and pattern-related code composition. For pattern-based design, a new representation of DPs similar to class diagrams as well as different modes for creating an instance of DP are proposed.

## 1 Introduction

Design patterns (DPs) [2] capture and communicate approved design solutions. DPs identify participants, their roles and collaborations, and the distribution of responsibilities. A DP explains the rationale behind the solution. It describes when a DP is to be applied, and the consequences and trade-offs of its application.

Without doubt, the key point in collecting DPs into catalogues is their reuse. Therefore we give a software reuse perspective on DPs. Particularly, we present a compositional view on reuse with DPs where DPs correspond to reusable components available and applicable during software development.

DPs may be used as building blocks when a software system is developed. But current reuse with DPs differs from standard reuse practices - it proceeds manually and basically relies on reuser's own knowledge and experience.

In this paper, we first give an overall view on reuse with DPs. An appropriate representation of DP solution is recognized as an important part of successful DP reuse. We thus propose a new way of DPs representation that complements current way of DP presentation in catalogues. A new model represents DP solution at its general and abstract level and allows to derive visible and manageable DP instances. Software reuse with DPs becomes more controlled and less error prone.

## 2 DP, DP Instance and DP Solution

For the reuse purposes, it is important to strictly

distinguish between the terms DP and DP instance. In the reuse context, DP solution should be emphasised as well.

A DP refers to the whole description as it occurs in a catalogue. A DP involves several sections describing the problem being solved, its context and solution. DP contains textual descriptions combined with diagrams or sample code. An emphasis is put on explaining the essence and applicability of DP.

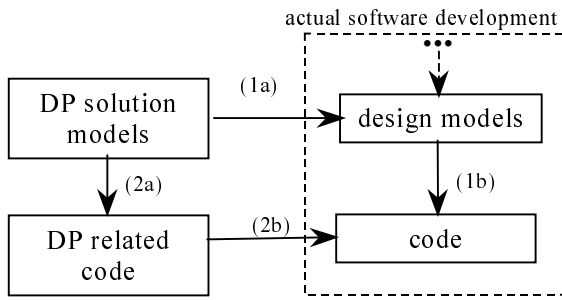
A DP instance is a product of DP reuse. It refers to those parts of software system that result from DP application. A DP instance is a part of the design/-implementation of a software system but not located in one place. Rather, it is scattered over several places and intertwined with other, to the DP non-relevant parts of design/code. Besides, there is no explicit information about the presence of DP instance in the software system so DP instances may easily be lost in the design/implementation.

When DP is reused, its solution part is applied. Currently, DP solution is expressed by means of some diagrammatic notations (for example, OMT class diagrams in [2]). They are chosen because of their availability. In fact, these types of diagrams are dedicated to model the artefacts of concrete and specific software system and do not fit to express DP solution sufficiently. DP solution in its essence significantly differs from artefacts of software system: it is general (not domain dependent), abstract (some details are omitted), and generic (describing a set of possible solutions) [9]. With common diagrams, these features are not reflected. As a consequence, not only reuse with DPs is difficult but also DP instances are in actual design not visible, not traceable and therefore hardly manageable.

## 3 Possible Approaches to Reuse with DPs

DP solution is in fact a model that can be reused when an actual software system is developed. Reuse with DPs may proceed in two principal scenarios (Fig. 1):

1. **Pattern-based design.** A DP solution is incorporated into presently developed software at the design phase (1a) and will be further refined (until its complete implementation) as an integrated part of design models (1b). During all subsequent development phases, a DP instance should be kept consistent with pattern regularities, constraints and dependencies.
2. **Pattern-related code composition.** Patterns are implemented independently of an actual software development (2a) and their integration is postponed to the code level (2b). The final software system is a mixture of pattern-related and application-related code.



**Figure 1.** Possible approaches to reuse with DP

Both approaches suppose DP solution models to be available. At this moment, the way in which the models are represented is not significant.

Though not shown in Fig. 1, we expect incremental refinements of the design models from early high-level system design to an actual implementation.

Pattern-based design is of our main focus and will be presented in next sections in more details.

Pattern-related code composition requires to refine DP solution models independently of the software system they are to be integrated into. Parameters, variables, etc. that come up in a pattern implementation should not be fixed to a particular software system. A difficult part of this approach is that several details of pattern implementation depend on the parts of currently developed software system, on their functionality, on the chosen programming language, constraints that result from overall system's architecture, etc. New notions like aspects [6] are similar to pattern-related code composition because they allow the parts of software system that crosscut its modularity to be woven into the software system's implementation at the code level. Unlike pattern-related code composition, aspects are implemented with detailed knowledge about the software system they will be integrated into.

## 4 Pattern-Based Design

The main goal of pattern-based design is to be able to weave DP solution at the design stage in such a way that particular DP instances could be recognised in further development. DP instances might be obtained on demand so that an overall view on their structure is known. Those parts of actual design that participate in a DP instance should be aware of taking part in common structure and collaboration because their additional concretisation (refinement) should proceed with respect to DP regularities and constraints.

A conceptual model for pattern-based design is depicted in Fig. 2. Here, the design repository plays a central role. It contains referential DP models, DP instance maps, and common design models. Referential DP models represent DP solution independently of any particular software system. They are placed into repository in order to be reused. Design models, on the other hand, contain all elements in an ongoing design of a software system. DP instance map relates elements from design models and referential DP models. DP instance map keeps an explicit record of each reuse of referential DP model and at any time gives an actual state of instantiation.

In order to be able to gain access to/from design repository, several supporting modules are depicted in Fig. 2. DP editor helps place new referential DP models into design repository. A person responsible for referential DPs either transforms a DP solution as described in an existing catalogue into suitable representation or derives them from existing ones.

DP instance editor allows DP instance maps to be created, modified or deleted with respect to referential DP models. An instance map serves for exact evidence of those parts of design models that contribute to DP solution. The basic mechanism for creating a DP instance is to map corresponding elements between design and referential models. Within this mechanism, different modes for DP instantiation are proposed and will be described later in more details. Independently of the mode, each DP instance and its actual elaboration can be determined.

Each DP instance can be checked for inconsistency, completeness or correctness. Inconsistency check conforms that a DP instance map is in accordance with actual design models i.e., only actual names of elements in design models occur in a map. Correctness and completeness checks detect errors in an instance map in relevance to its referential model. In completeness check those parts of a referential model that are not instantiated are detected. Correctness deals with flexibility of referential DP models - it checks if a DP instance satisfies the DP constraints.

Detected inconsistencies do not always mean an error. For example, when a software framework is developed, the creation of some pattern-related parts is intentionally postponed to a framework user. Any attempt to resolve such inconsistencies is wrong - keeping a developer informed by presenting and tracking inconsistencies would be sufficient.

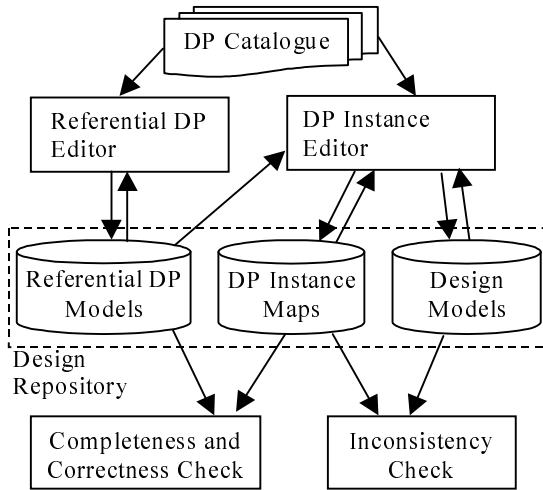


Figure 2. Conceptual model for pattern-based design

#### 4.1 Representation of Referential DPs

Referential DP models should reflect flexibility, generality and abstractness of DP solution but for the practical reasons they should be close to the representation of design models. We therefore propose both the design models and the referential DP models to be graph-like structures similarly to the UML (unified modelling language [1]) diagrams.

For design models in repository, class diagrams are chosen as representatives. Class diagrams consist of a set of nodes (classes, methods, attributes) and a set of edges (relationships inheritance, association, aggregation, etc.). Since class diagrams depict the static structure of software system, dynamics will not be show in design models. Nevertheless, our main ideas may be expressed sufficiently.

Referential DP models are represented as participant graphs (PG), a slight modification of class diagrams. Unlike class diagrams, nodes are DP participants with their names expressing the role at the general level. Participants are PG parameters. Instantiation of PG, i.e. assigning values to its parameters, results in a class diagram. Nodes in PG may have multiple occurrences. That is, several nodes may exist at the instance level. Each PG may have a set of constraints attached to it that express the dependencies among multiplicity of particular nodes. Particular instances of PG are correct if and only if

PG constraints are satisfied.

An example of PG is depicted in the centre of Fig. 3. A node with multiplicity  $n$  ( $C$  in this case) is graphically depicted as double circled. When the table in Fig. 3 is taken into account, the referential model for Composite DP (depicted on the left hand side of Fig. 3) is obtained. More detailed entities are not shown because of the space limitation.

A possible instance of Composite DP is shown on the right hand side of Fig. 3. DP instance map is also depicted. For a multiple node  $C$  shown in the PG, two nodes 4 and 5 exist in this instance. It is important that the edge connecting two nodes at least one of which has multiplicity many in PG (as is the edge a connecting nodes  $A$  and  $C$  in the Fig. 3) is expected to exist for each particular occurrence of the multiple node at instance level.

For each instantiation of PG, an explicit instance map exists. Instantiation must thus not be completed at once but should proceed in several iterations over the same instance map.

#### 4.2 Different Modes of DP Application

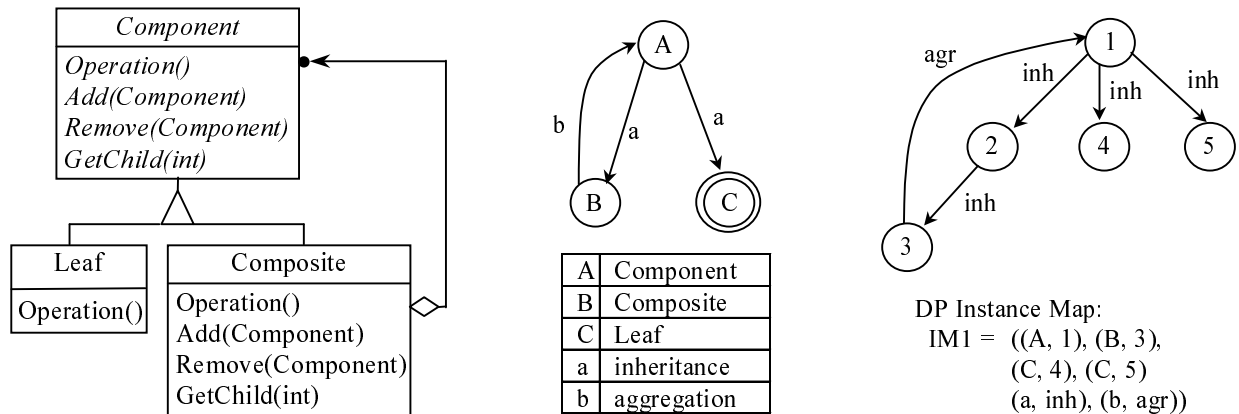
The way referential DPs are reused is not obvious. It may depend on the design stage of the ongoing development. When a DP instance is created in early stages of the high-level design, all pattern participants should be generated and added to the existing design models. In the late design, DP instance can be already embedded in the actual design and DP instance map merely helps explicate it for future developments.

The actual DP application scenario cannot be foreseen. We therefore propose distinct modes for DP application within pattern-based design: pure additive, non-additive, and mixed.

For all modes, a new DP instance map of referential DP model is created but in each mode design models are treated differently. In pure additive mode it is assumed that all parts corresponding to the DP solution are added to design models. In non-additive mode, on the contrary, it is supposed that all nodes in PG have their counterparts in design models and the instance map relates them to the referential DP model. Mixed mode is a combination of pure additive and non-additive modes – a part of DP is already embedded in existing design models but there are still parts of DP solution that have to be added to actual design.

### 5 Related and Future Work

Much effort to move reusing with DPs on more exact basis may be seen by researchers and practitioners. A



**Figure 3.** Composite DP - *left*: class diagram; *centre*: referential model; *right*: a possible instance

more precise DP representation shall lead to the absence of ambiguity in DP reuse. Proposals for formal specification of DPs have been a subject of several investigations [7, 8]. A serious problem of formal approaches may be the loss of reuse benefits since the formality puts the developer under a lot of obligations.

A diagrammatic approach to DP representation has been proposed in UML [1] where parameterized collaboration diagrams depict DP solution at the general level. Its weakness is that lower level's parts of DP solution, like methods or attributes, cannot be instantiated within specific needs. Also, to create several instances of one DP solution is troublesome.

In FRED [3], DP solutions are expressed at the general level as design contracts using the textual specification language Cola. The specialisation template binds a design contract to certain entities in the Java framework. Unlike our instance map, it comes out at the time of implementation.

In [5], DP solutions can be composed at the design level. They are represented as an UML extension of packages. DP packages can be specialized at the general level in a way that is similar to class inheritance, but their integration is based on simple renaming.

Our approach relies on a graph-based representation for general DP solution. With the proposed participant graphs, application of DP solution into actual design models is more controlled but still flexible. Several instances of the same referential DP model within the same software system can be created.

Explicit DP instances enable different application modes. A DP instantiation does not have to be completed iteratively according to iterations over development stages.

In the future work, we would like to investigate the appropriateness of hygraphs [4] to represent both class and participant graphs. By incorporating hygraph's

bubbles, different abstraction levels, at which the same graph can occur, could be more properly reflected. Another valuable effort would be to use the XML (extensible markup language) as a specification language for referential DPs.

#### Acknowledgements

The work reported here was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20.

#### References

- Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide. AddisonWesley (1998)
- Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995)
- Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J., Koskimies K., Paakki J.: Managing Object-Oriented Frameworks with Specialization Templates. ECOOP'99 Int. Workshop on Object Technology for Product Line Architectures, Lisbon, Portugal. (1999)
- Harel, D.: On Visual Formalism. *Communications of ACM*, 31(5):514530. (1988)
- Keller, R. D., Schauer, R.: Design Components: Towards Software Composition at the Design Level. In: Proc. of ICSE'98 International Conference on Software Engineering, IEEE Computer Society (1998)
- Kiczales, G., Lamping, J., Mendhekar, A., Lopes, C. V., Maeda, Ch., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Proc. of ECOOP'97 European Conference on Object-Oriented Programming, LNCS 1241, Springer Verlag (1997)
- Lauder, A., Kent, S.: Precise Visual Specification of Design Patterns. In: Proc. of ECOOP'98 European Conference on Object-Oriented Programming, LNCS 1445, Springer Verlag (1998)
- Mikkonen, T.: Formalising Design Patterns. In: Proc. of ICSE'98 International Conference on Software Engineering, IEEE Computer Society (1998)
- Smolárová, M., Návrát, P., Bielíková, M.: Abstracting and Generalising with Design Patterns. In: Proc. of ISICIS'98 Int. Symposium on Computer and Information Sciences, IOS Press Ohmsha (1998)