Knowledge–Based Software Engineering
P. Návrat and H. Ueno (Eds.)
IOS Press, 1998

A Technique for Modelling Design Patterns

Mária SMOLÁROVÁ, Pavol NÁVRAT, Mária BIELIKOVÁ

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

Abstract. The paper proposes a technique for modelling design patterns that allows representing them in a way suitable for application design. First, the issue of design pattern representation is tackled. We identify the space of design patterns and specifically discuss two of its dimensions: abstraction and generality. Next, we describe a technique for modelling design patterns by means of description schemata at a metalevel. For general pattern structure, a metaschema representation has been proposed that defines pattern fragments and relations. Also, constraints are introduced that restrict the possible structure of pattern instances.

1 Introduction

Reuse has been more than often mentioned as the primary motivation for using design patterns in software design. There are quoted other benefits of using them, cf. [9]: they are a succinct tool for explaining, communicating existing designs, and they serve as a guidance for a designer to focus onto a set of issues that other designers have found insightful. The same author, however, is quite cautious in judging the real reusability of object-oriented design patterns. More generally, the question whether the claim that object-orientation fosters reuse is justified remains open [15].

However, the very idea of design patterns is extremely fruitful. 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 [12].

Design patterns are a technique envisaged to support development and maintenance of software systems. However, the way they are currently formulated is essentially informal. It is based on a combination of diagrammatic description and a loose structure involving free text. Now, in order for the support to be as effective as possible it is reasonable if it is implemented by some tool. Any such tool that is supposed to operate upon design patterns would require them to be represented appropriately for a mechanical manipulation. Also – even when no computerised tool is envisaged – design patterns as known today e.g., [5], are defined far too loosely and urgently need to be reformulated in a more precise and formal way.

Hence, one of actual research topics is to find a suitable way of formulating and representing design patterns. When searching for such a representation, it is instructive to see the task in a wider context. One aspect concerns attempts to formulate design structures at various levels from components to frameworks, patterns and software architectures [14]. Other

aspect refers to a paradigm: although the object-oriented is the most frequently contemplated one, we defend the standpoint that there are patterns of e.g., functional or logic-programming design, too. For example, it is our long time experience that there are some typical inference mechanisms which can be taken as design patterns of knowledge based systems.

The paper attempts to propose a technique for modelling design patterns that would allow representing them in a way suitable for application design. The rest of the paper is structured as follows. In Section 2, the issue of design pattern representation is tackled. In Section 3, we elaborate an original analysis of the concept of patterns with a special focus on their abstraction and generality. Next, we describe the technique for modelling design patterns by means of an example. In Section 5, related work is given. We conclude with suggestions for future work.

2 Design Pattern Representation

Design patterns propose solution to recurring design problems in the software development. They are problem – solution pairs written in a uniform format. Each pattern consists of name, context, problem, solution, along with example, structure, dynamics, implementation, variant, known uses, consequences, and a reference to related patterns. Most of the pattern sections contain informal textual descriptions. The structure of the pattern includes also diagrams; in case of object oriented design patterns, class diagrams are frequently used. Dynamics can be described by typical scenarios of the run-time behaviour of the pattern using object interaction diagrams. Implementation is suggested by guidelines; only fragments of programs illustrating a possible implementation are usually given in a pattern.

As an example that will be used throughout the paper, we show the object-oriented design pattern Observer from the GOF catalogue [5]. Main goal of the Observer pattern is to define a one-to-many dependency between a `Subject` and its `Observers` so that when the `Subject` changes its state, all its dependants are notified and updated. In the catalogue, OMT class diagrams are used to express pattern structure [13]. The class diagram for Observer pattern is shown in Figure 1. Though using such a standard notation makes pattern structure familiar, it does not allow to express pattern constructs completely and unambiguously.

In many patterns, components with multiple occurrence appear. In our sample pattern, each

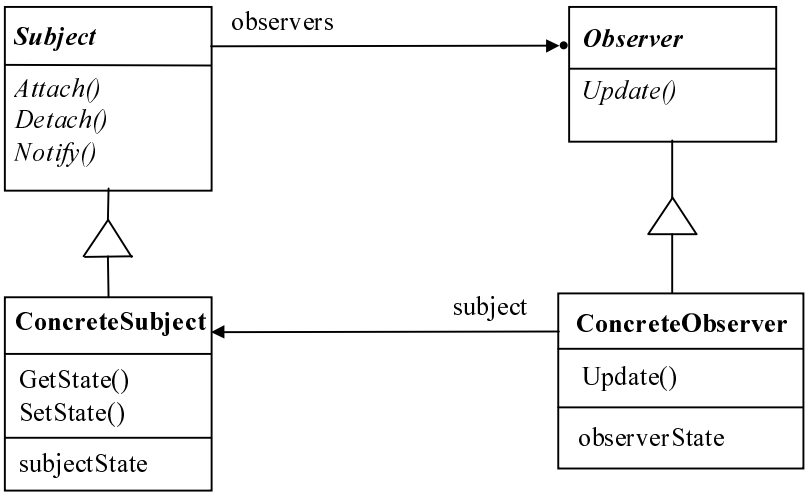

Figure 1. The Observer pattern structure.

`Subject` may have several `ConcreteObservers`. This is not expressed in class diagram in Figure 1. The lack of expressiveness is partially substituted by giving examples of possible structures, by code samples, and by informal textual descriptions given in the pattern.

It is clear that to describe design patterns accurately is a quite difficult task. In our work, we focus on the solution part of the pattern. A better representation is urgently needed; be it for better comprehension and reasoning about pattern properties, or to allow a manipulation by a computerised tool that would support the designer in some of his or her software system development activities.

3 Space of Design Patterns

One observation from the vast literature on design patterns is the striking ambiguity in very fundamental 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. A possible differentiation between abstract and concrete patterns is that the latter are implementations of the former [16].

Similar distinction can be found between components within a particular pattern. In the Observer pattern for example, the class `ConcreteSubject` is more concrete than the class `Subject` because it includes more details. Or, if one looks in the opposite direction, `Subject` is more abstract than `ConcreteSubject` because one method (`GetState`) and one variable (`subjectState`) have been abstracted away. However, even `ConcreteSubject` is still an (albeit less) abstract class because its methods are not implemented yet.

A pattern is becoming more concrete if it includes more 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 general pattern is through choosing a specific value for a pattern parameter. Referring to our sample pattern, there can be several `ConcreteObservers`. Number of `ConcreteObservers` 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 parametrisation 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.

Another way of specialising the design pattern is through renaming the general classes by application specific classes. This results in a domain specific design pattern. General names, such as `Subject` or `Observer`, are replaced by domain specific names. After the specialising process, the pattern has become more specific, but it is still at the same level of abstraction.

Thus, it should be recognised that design patterns are both abstract and general. We propose to represent both this property and the potential transformations by a space of design patterns. The space in which design pattern is transformed into a program has at least two dimensions i.e., abstraction and generality. It has roughly the shape depicted in Figure 2.

When working with a design pattern, there are four different directions to follow:

- concretisation – an abstract pattern is transformed into a more concrete representation,

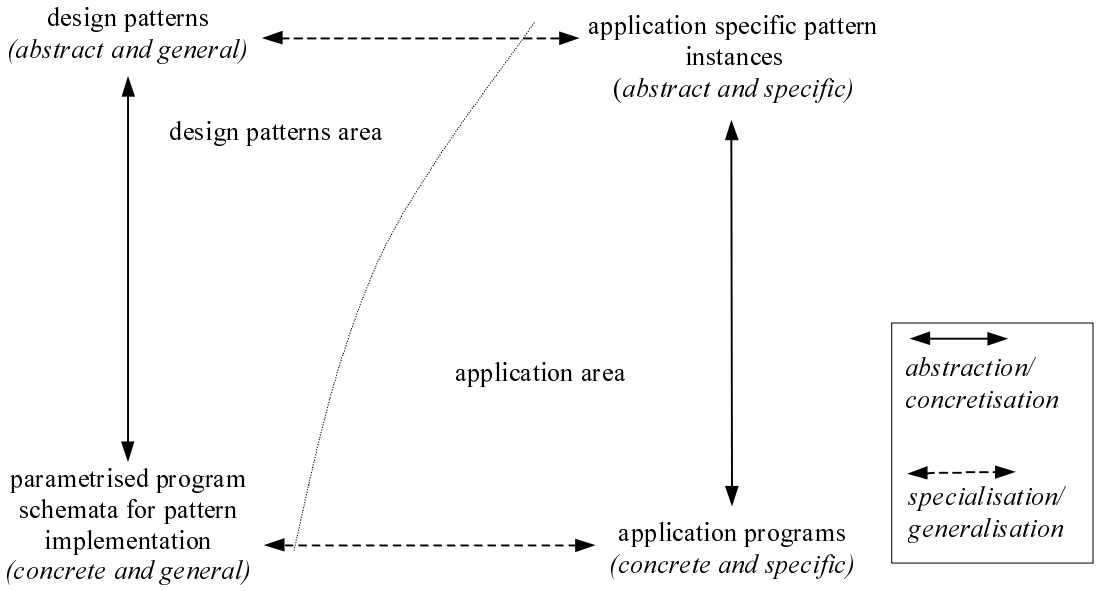

Figure 2. Pattern oriented software development.

- abstraction – the essence of the whole pattern or its parts is abstracted from a more concrete representation,
- specialisation – a general pattern can become more specific for example by instantiating some of its parameters or by embedding it in an application domain,
- generalisation – those specific structures which proved to be common in some application area are generalised into patterns.

The space of the pattern oriented software development can be divided into two subspaces (see Figure 2): the *design patterns area* which covers those pattern specialisations/generalisations and concretisations/abstractions that are domain independent and the *application area* in which patterns are transformed into/discovered from application specific cases.

The process of software development involves in principle movement in the space in all the four directions. This has first been presented in [10]. In particular, for the role of generalisation see also for example [7] where designing a framework by generalisation (i.e., bottom-up) from an application is discussed as a step by step process.

4 Design Pattern Modelling Technique

From our preceding analysis follows that pattern generality and abstraction are intrinsic pattern features. When this fact is not fully taken into account, they are origins of difficulties with formulating patterns more accurately.

In the pattern application, the developer in principle transforms a general and abstract pattern into the concrete and specific one. It is the process of *pattern instantiation*. Pattern instantiation is a vector in the design space that combines partial transformations from abstract to concrete, and partial transformations from general to specific.

The design space for Observer pattern is shown in Figure 3 with the most general and abstract pattern state in the left top corner. In this design space, a sample of states is depicted that introduce known pattern elements at different abstraction and generalisation levels.

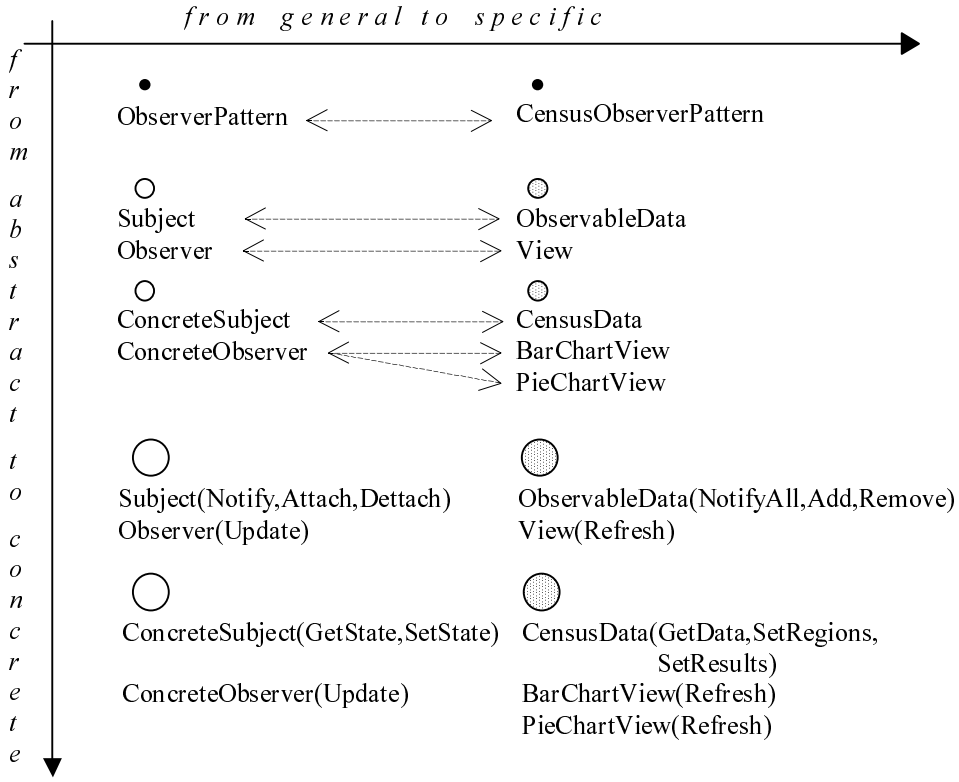

Figure 3. The design space for the Observer pattern.

`ObserverPattern` is shown as one element at the highest abstraction level. Descending along the abstraction level, abstract classes `Subject` and `Observer` and concrete classes `ConcreteSubject` and `ConcreteObserver` are considered. When concretising further, more details, e.g. pattern elements are added. Pattern elements are classes and their methods. The `Subject` class, for example, is known with its methods `Notify`, `Attach` and `Dettach`. In the next step, methods could be refined by identifying their parameters and return value. Pattern will become definitely concrete when implemented in some programming language.

We point out that transforming a pattern into distinct abstractions still preserves its generality level. A pattern may become less general when class and method names used in general pattern schema are replaced by the specific ones. Beside this, pattern parameters are specialised by setting to specific values. Pattern parameters are „hot spots" of the pattern structure which make the structure flexible.

Specialisation of our sample Observer pattern results in a domain specific design pattern (see also Figure 5). Two `ConcreteObservers` are chosen and are named `BarChartView` and `PieChartView`. Mapping between general and specigic level is depicted in Figure 3 by dashed arrows.

Viewing a pattern in a space one dimension of which is generality forces to distinguish its general and specific aspects. Pattern may be understood as a general structure giving a good design solution as well as a specific thing that lives in software. In our approach, we propose to represent general pattern structure as a metaschema from which pattern specific instances are derived.

At the general level, the metaschema describes those fragments that are used in pattern instantiation along with their properties and relations. Fragments are depicted by a rectangle in the metaschema. Basic fragments of the metaschema are classes and methods that play a role

for a pattern. Beside them, the metaschema contains a special fragment so called pattern fragment. Pattern fragment records the pattern presence. The most relevant fragments are directly related to it.

Relations between fragments are depicted by a rhomb. The relation has its name, maximal and minimal cardinality. It can also have an attribute that may specify what kind of relation it is dealt with. In the metaschema, relations that are common for many patterns occur, for example *is_concretised*, *declares* or *implements*, as well as those that are pattern specific, for example *subject* or *observers*.

Flexible parts of pattern structure – pattern parameters – are in our metaschema modelled by those relations that are of one to many cardinality. Pattern parameters will be assigned to specific values in the specialisation process. Important is that specific value of one parameter may impose constraints on other parameters. To preserve pattern structure during pattern instantiation, these constraints must be ultimately preserved. So we amend each pattern metaschema by pattern constraints that express dependencies among pattern parameters.

In Figure 4, a metaschema for the Observer pattern is shown. `Subject`, `Observer`, `ConcreteSubject`, and `ConcreteObserver` are the metaschema fragments that correspond to participating classes in the pattern. `Attach`, `Detach`, `Notify`, `GetState`, `SetState`, and `Update` correspond to methods playing a role in the pattern. Pattern fragments are connected through relation fragments. For example, the *observers* relation links fragments `Subject` and `Observer`. Typically, this kind of relation is achieved by declaring class data members that are of other class type. Cardinality of this relation is one to many because each `Subject` may have many `Observers`.

Another one to many relation is between `Observer` and `ConcreteObserver` classes.

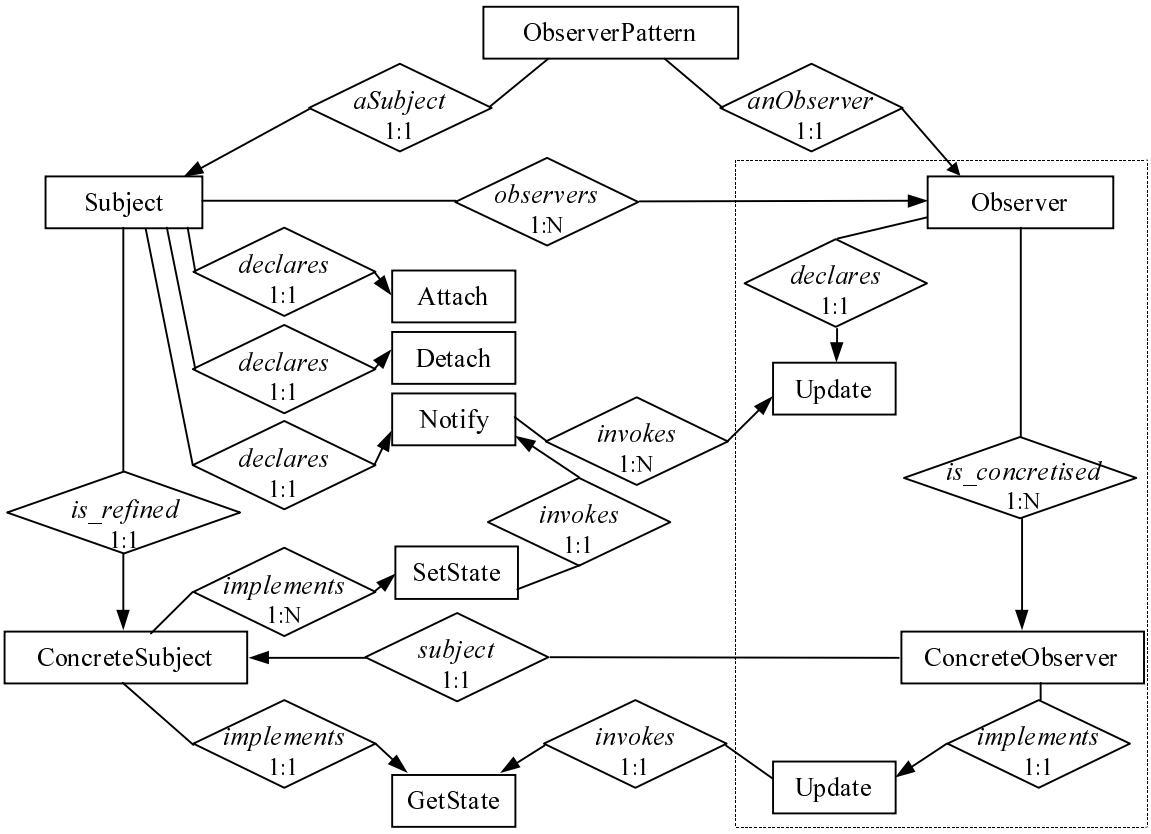

Figure 4. Metaschema for the Observer pattern.

There may be many `ConcreteObserver` in the pattern instance whilst each of them implements all abstract methods declared in `Observer` class. Such construct with a common interface on the top and many concrete implementations of it defines a family of Observer classes. Client that uses this construct, i.e. `Subject` in our sample pattern, may manipulate each member of this family uniformly so it is independent from concrete interface implementation. Relation between `Observer` and `ConcreteObserver` is abstract – concrete relation when method implementation is abstracted away in the `Observer` class. In our metaschema, `Observer` and `ConcreteObserver` are related through *is_concretised* relation.

To define a family of classes is a common theme that occurs in many design patterns. A typical construct that is used in GOF catalogue to declare a family of classes is encapsulated by a dashed line rectangle in Figure 4. To this construct, following pattern constraint should be attached:

(1) *for each ConcreteObserver*
cardinality_of(Observer.declares) = cardinality_of(ConcreteObserver.implements)

For Observer pattern specific constraints state that the number of `ConcreteObserver`s should correspond to cardinality of *observers* relation (2) and the cardinality of *observers* is bound to the cardinality of *invokes* relation of `Subject`'s `Notify` (3).

(2) *cardinality_of(Observer.is_concretised) = cardinality_of(Subject.observers)*
(3) *cardinality_of(Subject.observers) = cardinality_of(Subject.Notify.invokes)*

One possible instance of the Observer pattern is depicted in Figure 5. All application specific classes and methods, for example `PieChartView`, `CensusData` or `SetRegion`, are typed after the roles they play for the pattern. The pattern class `CensusObserverPattern` points to the principal classes in the pattern – `ObservableData` and `View`. All relations are depicted uniformly by an arrow accompanied by the relation name. The specific cardinality of the relation may be explicitly stated. It need not to be, as in the case of cardinalities of relations *observers* and `Subject` `Notify`'s *invokes* which are constraint by (2) and (3).

Constraints that are attached to the metaschema may help check structural dependencies in instances originated from metaschema.

5 Related Works and Conclusions

In this paper, a new technique for modelling of design patterns has been proposed which allows representing a pattern at two distinct generality levels: general and specific. For general pattern structure, a metaschema representation has been proposed. A metaschema defines fragments relevant to the pattern and relations that connect these fragments. Also, constraints are introduced that restrict the possible structure of pattern instances.

Our approach is quite general with respect to what can be the sort of objects that the fragments and relations represent. Throughout this paper, we have shown how the metaschema describes object-oriented patterns, however it is in principle possible to represent any collaborating elements by a metaschema.

One of the motivations for representing patterns at the metalevel is to have an appropriate tool that could support the pattern oriented software development.

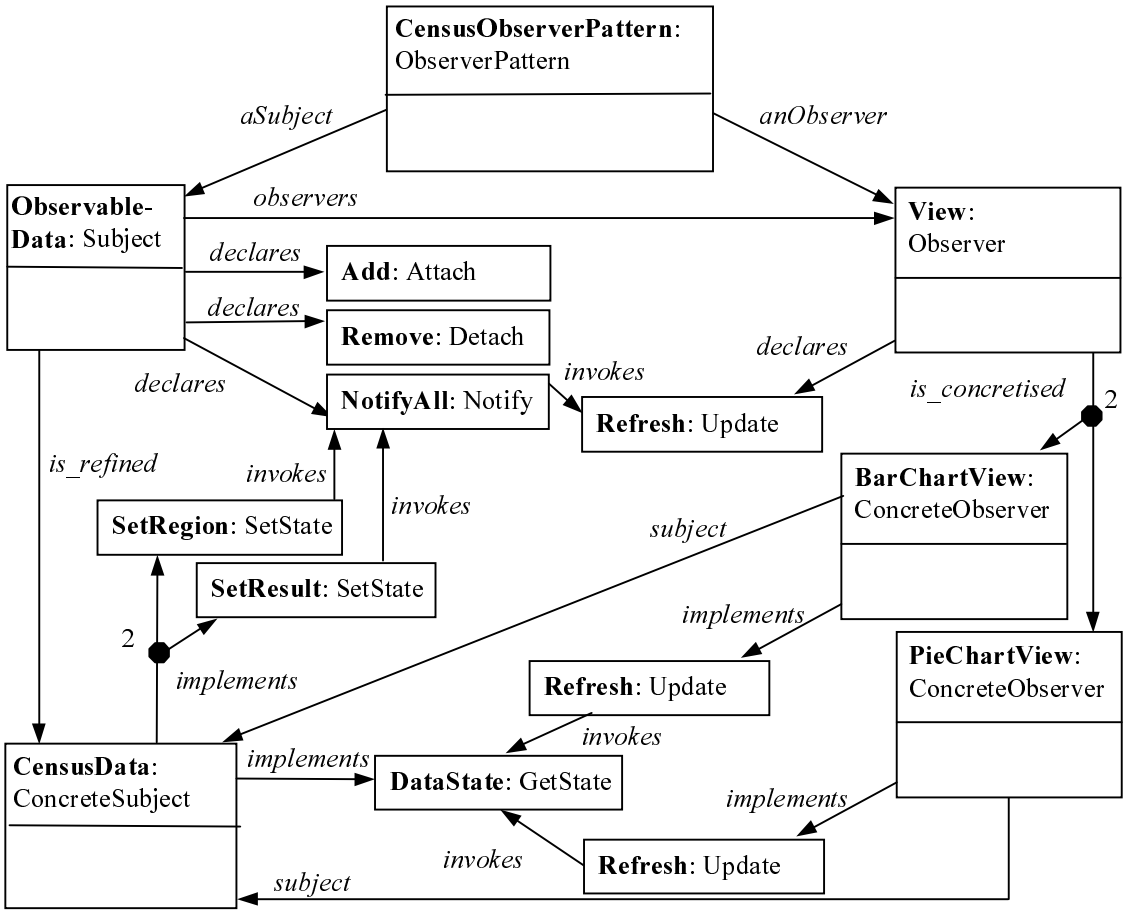

Figure 5. A possible instance of Observer pattern.

Merits for a tool that assists in software development with patterns has been discussed and the "hook&template" metapattern, a unified representation describing the patterns, has been proposed in [11]. Similar approach to the representation of patterns has been presented in [12]. The proposed metapatterns consider possible combinations of hook and template classes.

A different approach to pattern representation based on metaprogramming is presented in [1]. A design pattern is specified as a sequence of steps in its application. A language of commonly used steps that allow expressing pattern application as an algorithm has been defined.

In [3], a metamodel based on representing patterns as fragments is proposed. Every pattern consists of a set of fragments that fulfil a particular role for the pattern. All pattern fragments are linked to the root fragment. Examples of pattern fragments are classes, methods, attributes, association relations, containment relations, and inheritance relations. The difference to our approach is that all pattern fragments are linked to the root fragment regardless of their abstraction degree.

In [8], patterns are represented as metaschemata that describe application dependent pattern instances. Their model makes explicit all the information that the application developer is able to see and/or change. In this sense, their approach is very similar to ours.

The proposed approach to modelling of design patterns should 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 UML [17] 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 vs. Language Design. 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.

[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] K. Koskimies and H. Mössenböck, Designing a Framework by Stepwise Generalization. In: Proc. of ESEC'95 European Software Engeneering Conference, Sitges, W. Schaefer and P. Botella (eds.), LNCS 989, Springer Verlag, Sept. 1995, pp. 479-497.

[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] T. Menzies, Object-Oriented Patterns: Lessons from Expert Systems, *Software Practice and Experience* (to appear).

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

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

[12] W. Pree, Design patterns for object-oriented software development. Addison-Wesley, Wokingham, 1994.

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

[14] M. Shaw and D. Garlan, Software Architecture Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, 1996.

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

[16] J. Soukup, Implementing Patterns. In: Proc. of PLoP'94, also listed in Pattern Languages of Program Design, Addison-Wesley, ISBN 0-2010-201-60734-4, 1995.

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