Basis for a Course on Design Patterns: going beyond the intuition

Claudia Pons and Gustavo Rossi

Computer Science Faculty, LIFIA-UNLP
La Plata, Buenos Aires, Argentina
[cpons, gustavo]@info.unlp.edu.ar

Abstract

This article presents the motivation for teaching a regular subject (i.e. design patterns) in a more elaborated way. The proposal consists in adding formal foundations to the concepts that are usually presented to students from an intuitive point of view. Formal reasoning will allow students to use the concepts acquired during the course in a more mature way.

1. Introduction

Design patterns [6] document good solutions to recurring problems in a particular context. In the last years design patterns have become a mandatory subject in most software engineering courses all over the world. In general, Design Patterns are introduced according to the pioneer work of Gamma et al.[6]. In that book, specifications provided by design patterns are informally performed by means of concrete examples and by appealing to intuition, lacking a more definitive foundation. Consequently, many ambiguities arise which cannot be solved unequivocally. For example, the patterns mailing lists often engage in prolonged discussions whether a particular piece of code manifests an instance of one design pattern or the other. Another kind of confusion that originates from such ambiguities is whether one pattern is an special case of another, often without any satisfactory answer given.

We elaborate a course on design patterns including their intuitive and informal specification as a first introduction. After students acquire the intuitive idea of patterns we proceed by pointing out the weakness of those definitions, so that students realize the necessity to count with a formal foundation. Then, the most relevant strategies towards the formalization of design patterns are presented to students. Due to the fact that formalization of Design Patterns is still a problem under investigation, students are motivated to analyze and compare the different proposals available in recent research papers and even to suggest their own solutions.

The course turns round three questions: What are Design Patterns?, Why is it necessary to formally specify Design Patterns? and How can Design Patterns be formalized? These questions are addressed sequentially during the stages of the course which are described in the following sections.

2. What are Design Patterns and why is it necessary to formally specify them?

In this first stage of the course, Design Patterns are presented to students following the book of Gamma et al.[6]. As example, we include in Fig.1 the description of the Observer Pattern, using UML class diagrams, sequence diagrams and natural language.

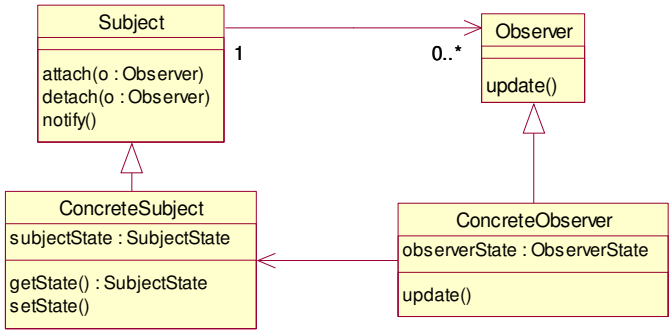
The specification of a design pattern is not the specification of a single program, but of a family consisting of all programs which satisfy the structural and/or behavioral constraints imposed by the specification. Design patterns should be formalized to enable (at least):

- Solving the question of validation e.g., does "this piece of code" implement "this pattern"? Given a program π in an object oriented programming language and the specification of a pattern Π, we are interested in the answer to the question whether π is an instance of Π (also: π is an occurrence of Π; π manifests Π; π implements Π)
- Solving questions of relationships between patterns, e.g.: Is one pattern the same as another (duplication)? Is one pattern obtained from a minor revision of another (refinement)? Is one pattern a component of another (composition)? Are two patterns unrelated (disjointness)? and so forth; :
- Solving questions of patterns composition; e.g.: how can two or more patterns be joined together? Does the combination originate new patterns?
- Tool support in the activities related to patterns.

Observer Pattern

Intent: define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically.

Structure:



Collaborations:

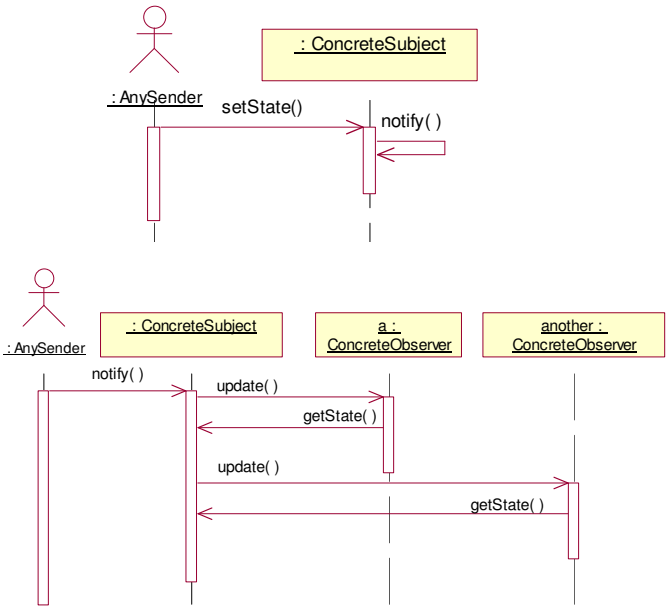


Figure 1. The Observer Pattern Specification.

Hence, there was a need for a formal means of accurately describing patterns. The formal specification of patterns is not meant to replace the existing informal descriptions but rather to complement them in order to achieve well-defined semantics and allow rigorous reasoning about patterns.

The formal specification of patterns helps pattern usage in the following situations:

- the user should decide which pattern(s) is (are) more appropriate to solve a given design problem within a context.
- the user should combine more than one pattern.
- the user need to find instances of patterns in a program.
- the user wants to transform a program to meet pattern specifications that are stored in the so-called pattern repository.
- the user wants to instantiate from a pattern specification, a possible implementation in a chosen programming language.

Notice that those activities can be manually performed by the user or can be (semi-) automatically executed by CASE tools.

In this stage of the course, a set of concrete examples of ambiguities and inconsistencies related to pattern validation and pattern inclusion will be analyzed with the students. For example, no satisfactory answer can be deduced form the specification of the Observer Pattern in [6] to the simple question below:

" If an observer is attached twice to the same subject, will it receive the message update() two times when its subject gets modified?".

Much more complex examples, collected from the pattern mailing lists repositories (e.g. http://hillside.net/patterns/mailing.htm), will be analyzed in the classroom.

As a reaction to these weakness, a number of ways to improve the precision of design pattern specification have been introduced in the last years., which will be briefly analyzed in the following sections.

3. How can Design Patterns be formalized?

During the course students will have the opportunity to analyze the main tendencies: adding OCL constraints to the pattern specification; improving the notation to specify pattern instantiation; applying metamodeling techniques and finally, building a full formalization schema.

3.1. Adding OCL constraints to the pattern specification

The Object Constraint Language OCL [14] can be used to specify invariants on the object states and to establish pre and post conditions for the operations. Fig. 2 shows an example of a set of constraints that can be attached to the UML class diagram in Fig. 1. in order to reduce the ambiguity of the specification. For example the formal specification of operation `attach()` in the class `Subject` establishes that an observer cannot be attached twice to the same subject, solving in this way the

ambiguity pointed out in section 3. On the other hand the invariant "noDanglingReferences" guarantees that observers are always attached to some subject, preventing this way from the occurrence of the problem of dangling references to deleted subject which is described in [6].

```
Context Subject::attach(o:Observer)
Pre: observers->excludes(o)
Post: observers=
           observers@pre->including(o)

Context Subject::detach(o:Observer)
Pre: observers->includes(o)
Post: observers=
            observers@pre->excluding(o)

Context ConcreteSubject::
             setState(s:SubjectState)
Post: subjectState=s and self^notify()

Context Subject::notify
Post: observers->forAll(o|o^update())

Context ConcreteObserver inv
     noDanglingReferences:
self.concreteSubject.oclIsUndefined().not
```

Figure 2. OCL constraints enriching the specification of the Observer Pattern.

3.2. Improving the notations to specify pattern instantiation

The modeling of design patterns and their instantiations are usually based on UML. However, there are still shortcomings for the representation of the instance or occurrence of a pattern. In particular, losing pattern-related information after the applications and compositions of design patterns remains a problem of UML. The modeling elements, such as classes, operations, and attributes, in each design pattern play some roles that are manifested by their names. The application of a design pattern may change the names of its classes, operations, and attributes to the vocabulary in the application domain. Thus, the role information of the pattern is lost. It is not obvious what modeling elements participate in each pattern. As a result, the benefit of using design patterns becomes compromised.

In this part of the course, students will analyze a number of techniques for explicitly representing individual design pattern in a complex system: Venn Diagram-Style Pattern Annotation, Dotted Bounding Pattern Annotation, UML Collaborations, Pattern:Role Annotations and Tagged Pattern Annotation, as follows:

Venn Diagram-Style Pattern Annotation: The first notation for identifying patterns in a design diagram is based on Venn diagrams [19]. This notation works fine with a small number of patterns per class. When a class participates in more and more patterns, the overlapping regions, where the class resides, may become hard to distinguish, especially when different gray levels need to be selected to represent different patterns. Besides the scalability problem, another shortcoming of this notation is that it is not explicit which roles each modeling element plays in each pattern.

Dotted Bounding Pattern Annotation: to avoid the shading problem, a variation of the previous notation that replaces shadings by dashed lines was proposed in [3]. This change solves the problem caused by shading. It, yet, remains hard to identify precisely the roles that modeling elements play. The scalability problem also remains since there can be many dashed lines clashing in the overlapping regions.

UML Collaboration Notation: the two conceptual levels provided by UML collaborations [13] (i.e. parameterized collaboration and collaboration usage) fit well to model design patterns. At the general level, a parameterized collaboration is able to represent the structure of the solution proposed by a pattern, which is enounced in generic terms. The application of this solution i.e. the terminology and structure specification into a particular context (so called instance or occurrence of a pattern) can be represented by collaboration usage. Dashed ellipses with pattern names inside are used to represent patterns. Dashed lines labeled with participant names are used to associate the patterns with their participating classes. While this notation improves over the previous notations with the explicit representations of pattern participants, it raises other problems. The dashed lines appear cluttering the presentation; the pattern information is mixed with the class structure, making both hard to distinguish. Moreover, this notation fails to represent the roles an operation (or attribute) plays in each design pattern. Furthermore, in [16] Gerson Sunye describes a number of unsolved semantic issues regarding parameterized collaboration, which hinder the precise specification of pattern instantiation.

Pattern:Role Annotations: to improve the diagrammatic presentation by removing the cluttering dashed lines, Gamma has defined a graphical notation, called "pattern:role" annotations documented in [19]. The idea is to tag each class with a shaded box containing the pattern and/or participant name(s) associated with the given class. If it will not cause any ambiguity, only the participant name is shown for simplification. This notation is more scalable than the previous notations and highly readable and informative. Unfortunately, the problems related shading arise again as the first notation. Similarly to the previous notation, this notation fails to represent the roles an operation (attribute) plays in a design pattern. If there are different instances of a pattern, furthermore, this

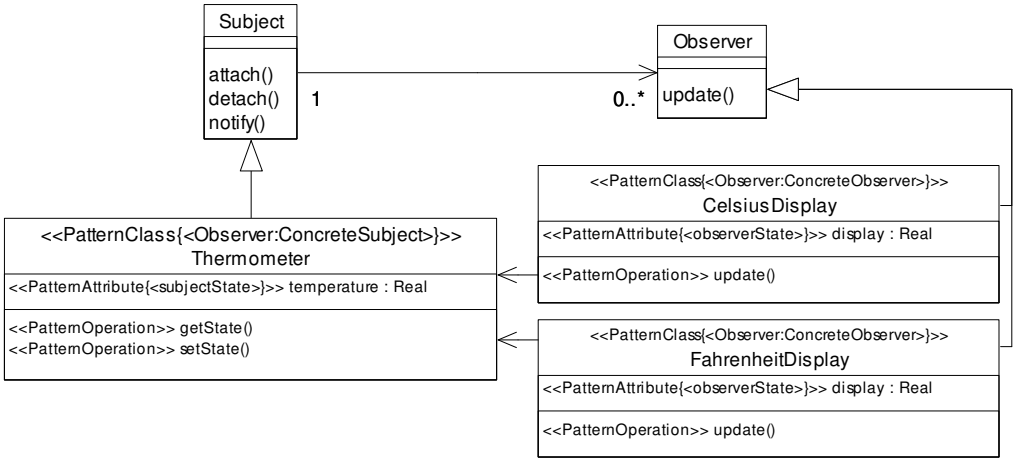


Figure 3. Tagged Pattern Annotation in an Observer Pattern instantiation.

notation cannot distinguish in which instance of the pattern a modeling element participates

Tagged Pattern Annotation: in [4] a new graphic notations (extensions to UML) was presented. This UML extension makes it possible to explicitly represent a pattern in the composition of patterns. Each individual pattern is explicitly documented, so that it can be identified easily. These extensions overcome the shortcomings of previous notations allowing for the explicit representation of the roles of each class, operation, and attribute in a pattern. The UML profile includes three stereotypes: PatternClass, PatternAttribute and PatternOperation, whose base classes are Class, Attribute and Operation, respectively. Each stereotype also defines one tagged value. These tagged values define exactly what role a class, an attribute or an operation plays in a design pattern. The name of the tagged value is "pattern" and the value of the tagged value is a tuple in the format of <name:string [instance:integer], role:string>. The "name" in the tuple is the pattern name in which a model element participates. The name fields of PatternAttribute and PatternOperation can be omitted if the class plays a role only in one pattern, and this omission will not create any ambiguity. The "instance" in the tuple indicates the instance of the pattern the model element participates. The "role" in the tuple shows the role that a model element plays in the pattern. For instance, the Thermometer class plays the role of ConcreteSubject in the Observer pattern in the example shown in Fig. 3. Then, the stereotype <<PatternClass{<Observer,ConcreteSubject>}>> is attached to the Thermometer class, establishing in this way that the stereotyped class participates in the only instance of the Observer pattern and plays the role of ConcreteSubject. As another example, the *temperature* attribute plays the role of subjectState in the Observer pattern. Then, the stereotype <<PatternAttribute {<Observer, subjectState>} is attached to the *temperature* attribute, declaring in this way that this attribute participates in the only instance of the Observer pattern by playing the role of subjectState.

A model element may simultaneously play different roles in different patterns. In this case, a new tagged value with the same format is attached to the model element for each additional pattern it participates.

The limitation of this notation is that the pattern-related information is not as noticeable as the "pattern: role" notation with shading, which is a trade-off. For a small number of patterns, this new notation can combine with the dotted bounding notation by bounding each pattern with dashed circles so that the pattern boundaries are explicitly depicted.

3.3. Pattern Metamodeling

Le Guennec and Sunye propose in [9] a minimal set of modifications to the UML meta-model to make it possible to model design patterns and represent their occurrences in UML, opening the way for some automatic processing of pattern applications within CASE tools.

3.4. Building a Complete Formalization Schema

Proposals presented so far have highly positive practical impact due to the fact that they make it possible to enhance the formality of both the pattern specification and its instantiation. However, the meaning of the specifications remains still semi-formal, hindering the solution of the patterns validation problem. Only a complete formalization of both, the pattern and its instantiation will enable us to solve the question of validation.

Given a pattern Π and a system π, the problem of *pattern validation* can be reduced to the problem of *refinement validation* in a formal language in the case we could find a formal specification language in which it were possible to specify the structure and behavior of the design pattern (i.e. $Spec_{\Pi}$); and also that the formal specification

language expressive enough to capture the specification of the object oriented systems (i.e. $Spec_{\pi}$) and finally, that this language were equipped with a refinement calculus.

Given these three conditions, the following theorem holds:

Theorem: π is an instance of Π iff $Spec_{\pi}$ is a refinement of $Spec_{\Pi}$.

Consequently, the validation problems can be decomposed into the following steps:

- Step 1: Finding a formal specification of the design patterns: $Spec_{\Pi}$.
- Step 2: Finding a formal specification of the object oriented program: $Spec_{\pi}$
- Step 3: Finding the specification of the refinement relationship: $R_{\pi-\Pi}$
- Step 4: Applying the refinement calculus provided by the formal language on the results of the previous steps.

Figure 4 shows an sketch of the validation procedure, which is described in the following subsections, step by step.

Step 1:

In principle, we must demonstrate that patterns can indeed be formalized. Although there is no yet a general agreement, a number of formal specifications for some design pattern have already been provided: Tommi Mikkonen presents in [12] a way to formalize temporal behaviors of design patterns whit the DISCO method, paying special attention to their natural utilization when composing specifications of complex systems. In [5] and [11] formal specification of design patterns are elaborated using the formal language LePUS and DPML respectively. Taibi and Ngo in [18] propose a Pattern Specification Language (BPSL) aimed to achieve equilibrium by specifying both structural and behavioral aspects of design patterns. The language combines two subsets of logic: one from the First-Order Logic (FOL) and the other from the Temporal Logic of Actions (TLA). Saeki in [17] defines a behavioral specification of GOF Design Patterns with LOTOS.

In fig. 5 we show an specification of patterns Observer elaborated by the students, using the formal language Object-Z.

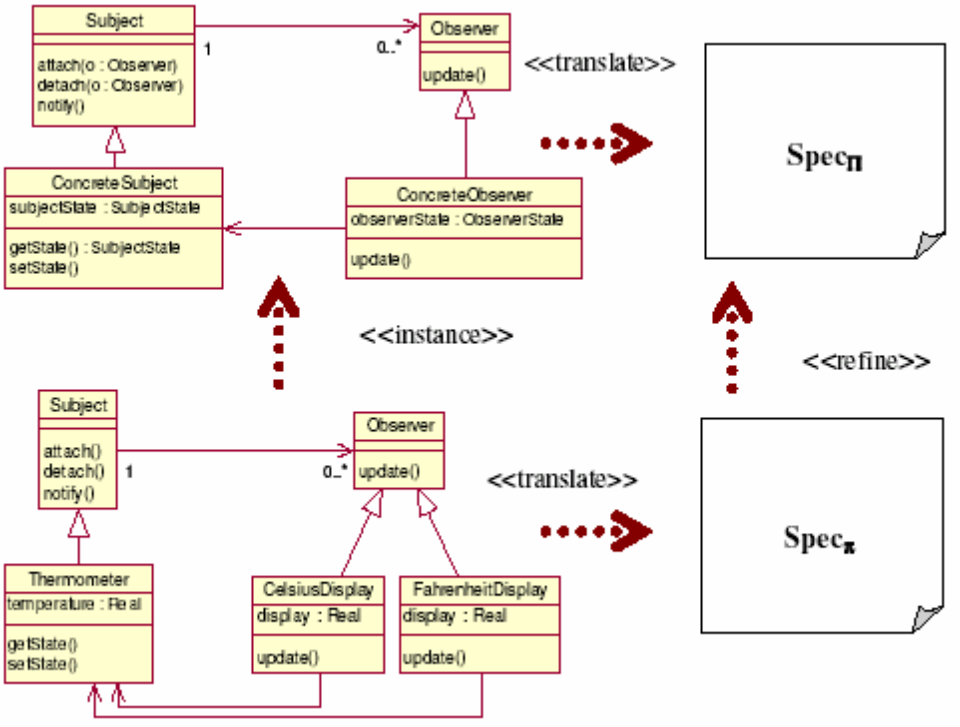


Figure 4. full formalization schema for pattern definition and instantiation.

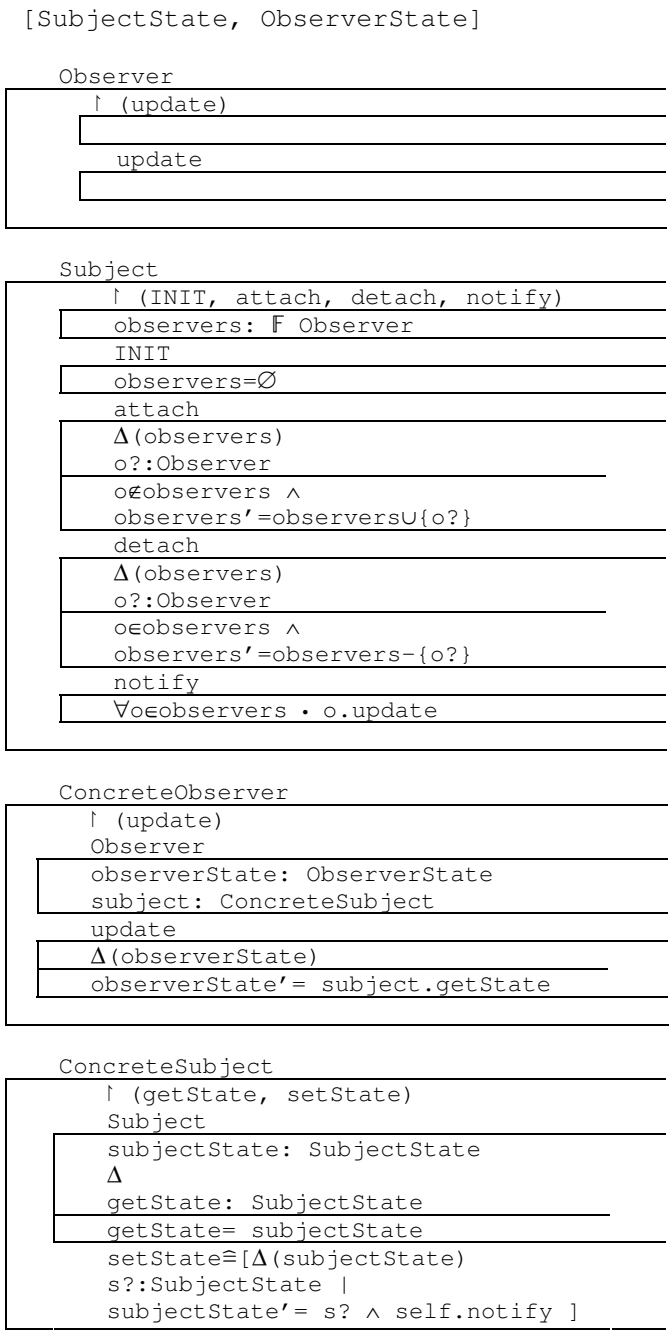


Figure 5. Object-Z specification of the Observer Pattern.

In Object-Z [2] a class is represented as a named box with zero or more generic parameters. The class schema may include local type or constant definitions, at most one state schema and initial state schema together with zero or more operation schemas. The operations define the behavior of the class by specifying any input and output together with a description of how state variables change. Concretely, an Object-Z class is a 6-upla (vblist, parents, localDef, State, Init, $\{Op_i\}i\in I$) such that vblist is the visibility list, parents is the list of inherited classes, localDef are local definitions, State is a state schema, Init is the initial state schema and Op_i are operations on State∧State'.

The Object-Z specification in Fig.5 emulates the UML specification of the pattern depicted in Fig.1, where ConcreteSubject and ConcreteObserver are subclasses of Subject and Observer, respectively. The abstract class Observer specifies an empty state schema and empty schema for the operation `update()`, while the class ConcreteObserver defines an state variable named `observerState` of type `ObserverState` to hold the state of the concrete observer. Similarly, the class ConcreteSubject specifies an state variable named `subjectState` of type `SubjectState` to hold the state of concrete subjects. Besides, the query operation `getState()` is specified as a derived state variable (which is the most practical way to represent query operations in Object-Z).

Step 2:

In general we count with a semi formal specification of the program given by means of UML diagrams, or simply programming code. From this semi-formal specification we should produce a formal one. Proposals towards the automatic creation of a formal specification from UML models have been presented by Kim and Carrington [7], Davies and Crichton [1], Pons et al.[15], Ledang [10], among others. Fig. 6 shows the specification in Object-Z of the Thermometer system, derived from the UML specification in Fig. 3.

Step 3:

In Object-Z to verify the refinement relation between two given specifications A and C, it is necessary to count with a relation R on A.State ∧ C.State. This relation, called *retrieve relation*, is an explicit documentation of how the properties of an abstract element are mapped to its refined versions, and on the opposite direction, how concrete elements can be simplified to fit an abstract definition.

Lets record that our main hypothesis estates that a program π is an instance of the pattern Π if an only if $Spec_\pi$ is a refinement of $Spec_\Pi$. In this case, the *retrieve relation* can be seen as an i*nstantiation relation* which establishes how the properties (such as, classes, attributes and operations) defined in a pattern are mapped to its instantiations on a concrete system, and on the opposite direction, which roles concrete elements play in a pattern.

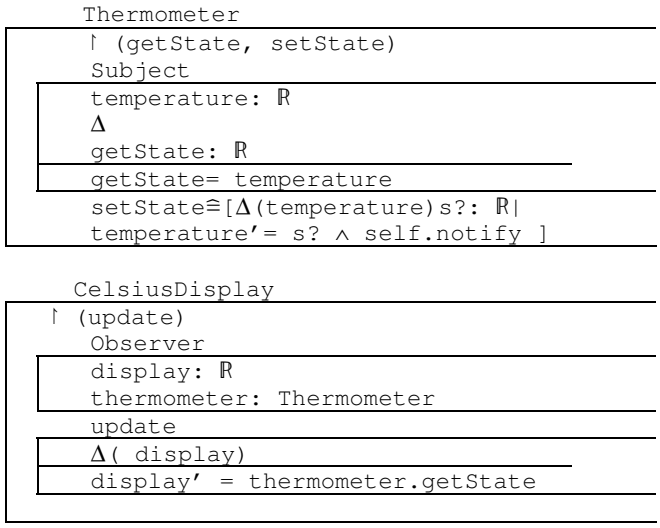


Figure 6. part of the Object-Z specification of the Thermometer System.

We have already discusses about the shortcomings of UML for the representation of the occurrence of a pattern. We have seen that, in general, it is not obvious which modeling elements participate in each pattern, consequently, in this cases no retrieve relation can be derived from the UML specification. Later on, we presented a number of additional notations to overcome the problem. Assuming that pattern instantiation is represented using Tagged Pattern Annotation [4], we could derive a retrieve relation in Object-Z in the following way:

R_{ST}:Observer.ConcreteSubject ↔ Thermometer
∀s:Observer.ConcreteSubject,t:Thermometer •
((s,t)∈R_{ST} ⇔s.subjectState=t.temperature)

R_{OC}:Observer.ConcreteObserver↔CelsiusDisplay
∀o: Observer.ConcreteObserver,
c: CelsiusDisplay •
((o,c)∈R_{OC} ⇔ o.observerState = c.display)

R_{OF}:Observer.ConcreteSubject↔
FahrenheitDisplay
∀o: Observer.ConcreteSubject,
f: FahrenheitDisplay •
((o,f)∈R_{OF} ⇔ o.observerState=f.display)

The Tagged Pattern Annotation allowed us to recover the retrieve relation from the UML diagram in a straightforward way. For example the stereotype <<PatternClass{<Observer,ConcreteSubject>}>> which is attached to the Thermometer class, gives rise to the Relation R_{ST} which establishes the connection between the concrete class Thermometer and the role it plays in the pattern (i.e. ConcreteSubject). As another example, the stereotype <<PatternAttribute {<subjectState>} that is attached to the temperature attribute, originates the expression

(s,t)∈R_{ST} ⇔ s.subjectState=t.temperature

that declares the relationship between the concrete attribute temperature and the role it plays in the pattern (i.e. subjectState).

This translation from Tagged Pattern Annotation to Object-Z can be automatically performed.

Step 4:

Refinement is formally addressed in the context of Object-Z specifications [2] as follows:

An Object-Z class C is a refinement (through downward simulation) of the class A if there is a retrieve relation R on A.State∧C.State such that every visible abstract operation Aop is recast into a visible concrete operation Cop and the following hold:

(Initialization) ∀C.State• C.init ⇒(∃A.State • A.init ∧R)
(Applicability) ∀A.State;C.State • R ⇒
(preAop ⇒ preCop)
(Correctness) ∀A.State; C.State; C.State'•
R∧ preAop ∧Cop ⇒ ∃.A.State'• R'∧ Aop

This definition allows preconditions to be weakened and non-determinism to be reduced. In particular, applicability requires a concrete operation to be defined everywhere the abstract operation was defined, however it also allows the concrete operation to be defined in states for which the precondition of the abstract operation was false. That is, the precondition of the operation can be weakened.

On the other hand, correctness requires that a concrete operation is consistent with the abstract whenever it is applied in a state where the abstract operation is defined. However, the outcome of the concrete operation only has to be consistent with the abstract, and not identical. Thus if the abstract allowed a number of options, the concrete operation is free to use any subset of these choices, solving non-determinism.

Using this formal reasoning we are able to prove that the refinement relations depicted in Fig.7 hold, that is to say:

- class Thermometer is a refinement of class ConcreteSubject via the retrieve relation R_{ST},
- class CelsiusDisplay is a refinement of class ConcreteObserver via the retrieve relation R_{OC},
- class FahrenheitDisplay is a refinement of class ConcreteObserver via the relation R_{OF}.

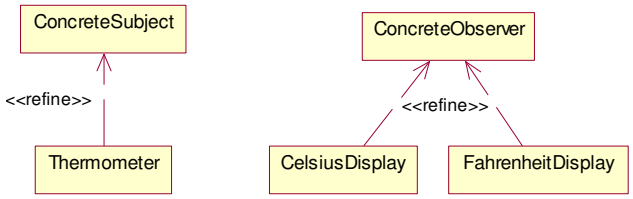


Figure 7. Refinement as a way to solve the pattern validation problem.

4. Conclusion

In the last years, the design pattern subject has been incorporated in the curricula of most software engineering courses all over the world. In general design patterns are taught from an informal angle. Despite the fact that there is an important number of theoretical works giving a precise description for design patterns and providing rules for analyzing their properties it is seldom the case that those formalisms are taught to students. This kind of courses qualifies an student to perform a light use of the technology, that is to say she/he can create object oriented models applying design patterns and she/he can discover design patterns immersed in legacy models, but frequently feeling unconfident about the correctness of her/his actions and results.

On the other hand, the standard Computer Science Curricula [20] includes courses on logics and formal languages; but there is a deep gap between both areas: practical software specification techniques on the one hand and formal specification techniques on the other hand. Only few students can join together both areas of knowledge, realizing the benefits of combining them.

In this article, we provide motivation for an undergraduate course on design patterns incorporating both informal and formal approaches. We survey the main publication in the area and propose an organized way to present them to the students.

Acknowledgements

This work was partially supported by Microsoft Research RFP 2005.

5. References

[1] Davies J. and Crichton C. Concurrency and Refinement in the Unified Modeling Language. Electronic Notes in Theoretical Computer Science 70,3, Elsevier, 2002.

[2] Derrick, J. and Boiten,E. Refinement in Z and Object-Z. Foundation and Advanced Applications. FACIT, Springer, 2001

[3] Dong, Jing. Representing the Applications and Compositions of Design Pattern in UML. Procs. of the ACM Symposium on Applied Computing, Melbourne, USA, pgs 1092–1098, 2003.

[4] Dong, Jing and Yang, Sheng. Extending UML To Visualize Design Patterns In Class Diagrams. SEKE 2003.

[5] Eden, A. H. "Formal Specification of Object-Oriented Design." International Conference on Multidisciplinary Design in Engineering 2001,November 21-22, 2001, , Canada

[6] Gamma, E. Helm, R. Johnson, R. and Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995.

[7] Kim, S. and Carrington, D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML´99 Conference, Lecture Notes in Computer Sciencie 1723 (1999).

[8] Lauder, Anthony and Stuart Kent. Precise visual specification of design patterns. In Eric Jul, editor, ECOOP '98 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol. 1445. pages 114-134. Springer, 1998.

[9] Le Guennec, Alain Sunye, Gerson, and Jezequel,Jean-Marc. Precise Modeling of Design Patterns . "UML" 2000 - The Unified Modeling Language. York, UK, LNCS 1939.

[10] Ledang, H. and Souquieres, J.. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. Procs. of Asia-Pacific Software Engineering Conf. IEEE Computer Society. Australia. December 4-6, 2002.

[11] Mapelsen, D., Hosking, J. and Grundy,J.. "Modelling: Design Pattern Modelling and Instantiation Using DPML". Proceedings of the Fortieth International Conference on Tools Pacific - Volume 10 (Feb. 2002), Sydney, Australia, pp. 3-11.

[12] Mikkonen, Tommi. Formalizing design patterns. In ICSE'98, pages 115-124. IEEE CS Press, 1998.

[13] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification. August 2003. http://www.omg.org.

[14] OCL 2.0. OMG Final Adopted Specification. October 2003

[15] Pons, C., Baum, G., Felder M. Formal Foundations of Object Oriented Modeling Notations. 3rd Int..Conference on Formal Engineering Methods, IEEE ICFEM 2000. UK.

[16] Sunye, Gerson, Alain Le Guennec, and Jean-Marc Jezequel. Design Patterns Application in UML. Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 44-62, 2000. Springer-Verlag Berlin Heidelberg 2000.

[17] Saeki M. (2000): Behavioral Specification of GOF Design Patterns with LOTOS. — Proc. IEEE Asia Pacific Software Engineering Conference, APSEC'2000, Singapore, pp. 408–415.

[18] Taibi T. and Ngo D.C.L (2003): Formal specification of design pattern—A balanced approach.—J. Object Technol., Vol. 2, No. 2, pp. 127–140.

[19] Vlissides, J.. Notation, Notation, Notation. C++ Report, April 1998.

[20] Computer Science Curricula 2001. The Joint IEEE Computer Society/ACM Task Force. http://www.computer.org/education/.