Representing and Applying Design Patterns: What Is the Problem?

Hafedh Mili & Ghizlane El-Boussaidi

Laboratoire de Recherches en Technologies du Commerce Électronique (LATECE)
Faculté des Sciences, Université du Québec à Montréal
B.P 8888, succursale Centre-Ville, Montréal (Québec) H3C 3P8, Canada
{hafedh.mili, el_boussaidi.ghizlane}@uqam.ca

Abstract. Design patterns embody proven solutions to recurring design problems. Ever since the gang of four popularized the concept, researchers have been trying to develop methods for representing design patterns, and applying them to modeling problems. To the best of our knowledge, none of the approaches proposed so far represents the design problem that the pattern is meant to solve, explicitly. An explicit representation of the problem has several advantages, including 1) a better characterization of the problem space addressed by the pattern—better than the textual description embodied in pattern documentation templates, 2) a more natural representation of the transformations embodied in the application of the pattern, and 3) a better handle on the automatic detection and application of patterns. In this paper, we describe the principles underlying our approach, and the current implementation in the Eclipse Modeling Framework™.

1 Introduction

Software development may be seen as a sequence of property-preserving transformations that are applied to a set of user requirements to produce a functional software that satisfies a number of quality requirements [16]. Researchers have long tried to describe those transformations precisely. However, doing so in a domain independent way has proved elusive because of the vast amounts of both domain and development knowledge that would be required. *Design maintenance systems* (see e.g. [3]) break the process of development by transformation into, i) *choosing* a transformation, which is a knowledge-intensive and complex task, but involving little labor, and ii) *applying* a chosen transformation, which is labor-intensive but knowledge poor. They, thus, focus on applying chosen transformations, and argue that, by changing the requirements a bit, they can update the design by reapplying the same set of transformations that were chosen for the initial requirements. To some extent, the design patterns movement takes an orthogonal approach to design maintenance systems : instead of focussing on small changes in the overall requirements, they focus on localized, recurrent design problems, whose solutions they codify [9].

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 186-200, 2005.

Since the publication of the gang of four book, several researchers have worked on providing support to developers for applying design patterns, including [4], [5], [6], [18], [1], and many more. Viewing design patterns as reusable artefacts, their usage requires [13] :

- Recognizing opportunity : recognizing the pattern as a potential solution to the problem at hand,
- Understanding the artefact : understanding the pattern, its structure, and the principles underlying it, and
- Adapting the artefact: in this case, applying the pattern to the problem at hand.

Each one of these tasks requires a particular representation of the pattern. To recognize opportunity, we need a representation of the problem solved by the pattern that we can match to a representation of the problem at hand. To understand the pattern, we need a representation that is intuitive, typically mixing text with a visual notation. The third task requires a representation of the transformation embodied in the pattern.

The approaches that we have studied have tackled either the understanding task, or the pattern application task (e. g. [2], [18], [17]), and sometimes both [7],[11]. Significant research in the software metrics area has addressed the opportunity aspects, but does little for pattern understanding, or for performing the subsequent refactoring—with a few exceptions, e.g. [19]. We know of no approach that tries to handle all three tasks. We argue that a representation of the design problem is required for all three tasks:

- We cannot ascertain the relevance of a design pattern to a design problem without a *formal characterization* of the design problems that the pattern is meant to solve,
- Proper understanding of the pattern requires that we understand the structure of our software (its models) *before* applying the pattern, and *after*
- The application (instantiation) of the pattern may be expressed declaratively as a mapping between a model of the problem and a model of the solution, that can be implemented by a generic transformation engine.

In this paper, we describe our approach for representing and applying design patterns. Section 2 presents the representation of the design problem, which is illustrated using the *bridge pattern*. We describe the model of the solution and the model of the transformation in section 3.We describe our EMF-based implementation in section 4. We compare our approach to related work and discuss the issue of assessing a pattern's applicability in section 5. We conclude in section 6.

2 Modeling the Design Problem

2.1 Example: The Bridge Pattern

Figure 1 illustrates a situation that warrants the bridge pattern [9]. Assume that we want to develop a program that manipulates graphical window objects, and that we want our program to be portable across OS platforms (MS Windows, Unix-based,

etc.). A typical object-oriented design idiom consists of creating a root abstract class—call it *Window*—that defines *abstract* methods that specify the behavioral contract that the various implementations must provide. This solution is illustrated by the left hand-side of Figure 1. Assume now that we want to define new *types* of windows, e.g. square windows, which may provide additional behaviour (new methods) or refine existing ones (e.g. providing a more optimal implementation of some generic behavior). The extended design is shown on the right hand-side of Figure 1: a new subclass of *Window* has been created—*SquareWindow*—and new implementations of *SquareWindow* have been defined, one for each target platform.

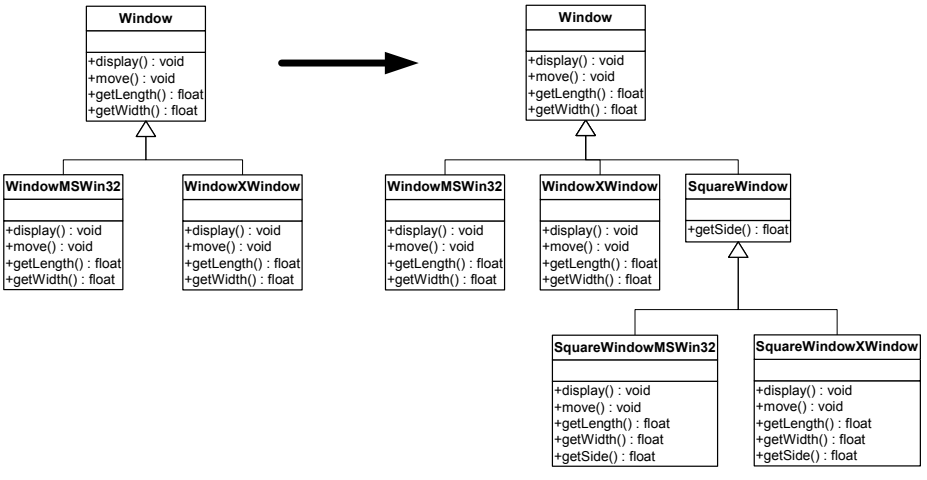

Fig. 1. An example problem solved by the bridge pattern.

The solution proposed by the *bridge pattern* consists of *decoupling implementations from abstractions* by putting them in separate class hierarchies that can evolve independently. In particular, new implementation classes are needed *only* in those cases where they provide new behaviour implementations. The example of Figure 2 shows a case where a new *abstraction* (*SquareWindow*) uses the same implementation as its parent (*Window*).

Figures 1 and 2 help explain the design pattern by showing a sample problem and the corresponding solution, i.e. a <problem, solution> *instance*. We would like to abstract, from this example, and from the textual pattern documentation, a representation of the problem solved by the bridge pattern that would support the three reuse tasks mentioned in the introduction. The subsequent subsections describe our representation.

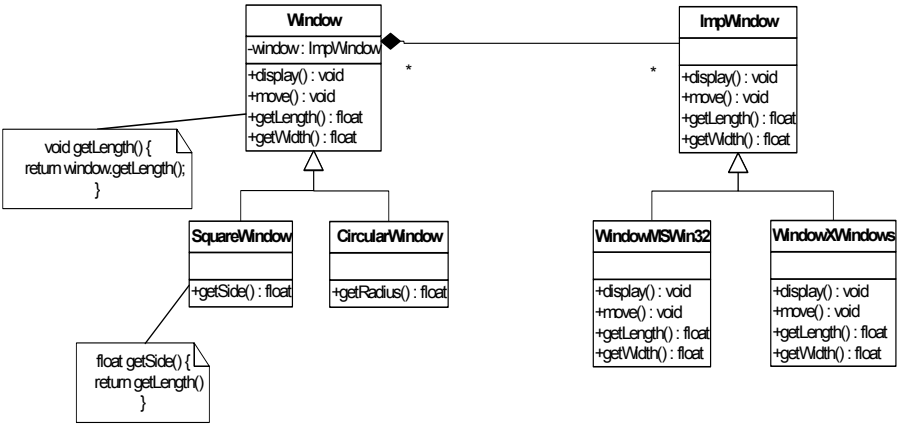

Fig. 2. The solution proposed by the Bridge pattern

2.2 A Metamodel of the Design Problem

Instances of the design problem solved by the bridge pattern are analysis and design models of applications. To describe the *class* of problems solved by the pattern, we will define a *problem meta-model*, i.e. a model whose instances are models such as the one in Figure 1. Figure 3 shows a first-cut metamodel.

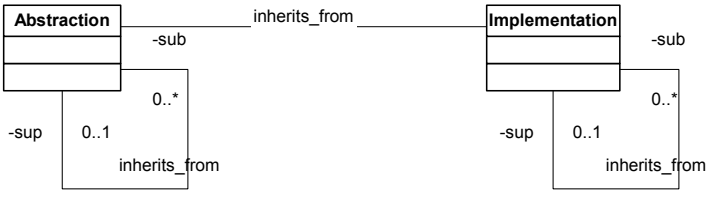

Fig. 3. A first-cut metamodel of the problem solved by *bridge*.

The classes *Abstraction* and *Implementation* are meta-classes in the sense that their instances are classes such as *Window* or *WindowMSWin32*, respectively. The associations labeled "inherits_from" represent *inheritance relationships that exist between instances of the corresponding classes.* For example, such a relationship exists between the two *abstractions SquareWindow* and *Window* (see Figure 1). Similarly, there is an inheritance relationship between the *implementation SquareWindowMSWin32* and the *abstraction SquareWindow*. Note that, for the time being, we don't worry about what it means to be "an abstraction" or "an implementation". We interpret these (meta)classes as simple tags for now; we later discuss their semantics.

A metamodel of the problem should also include a description of the operations that are affected by the pattern. The operations of the *Abstraction*'s will be abstract, and the operations of the *Implementation*'s will be concrete. Further, each *Implementation* must implement all of the abstract operations of the *Abstraction* from which it inherits. We represent this constraint as a constraint between the association "inherits_from", between classes, and the association "implements", between the corresponding operations (Figure 4).

There is yet more to represent. We would normally need to capture return types and parameters of the operations that are affected by the pattern. We should also cover cases where *Abstraction*'s are not pure abstract classes, but may include some implementations. To keep the model simple, we will ignore parameters[1] and partially abstract classes.

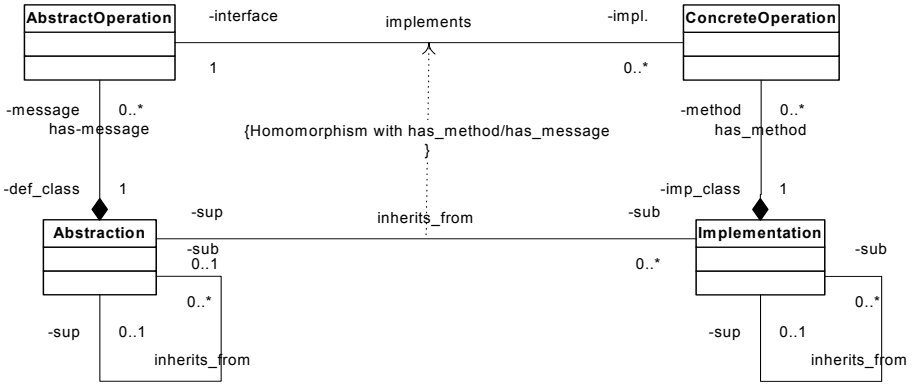

Fig. 4. A metamodel of the problem solved by the *Bridge* pattern. Take two.

2.3 The Missing Link: The Time Derivative!

To some extent, the various design patterns aim at shielding a client program from *changes* in the functionality, the environment, or the implementation of another program. Design patterns either make those changes transparent, or minimize their maintenance impact.

We argue that the *dynamic* nature of the problem to be solved is *an essential part* of the design problem, and as such, it needs to be captured explicitly. Consider the case of the *visitor* pattern. This pattern is applicable when a class hierarchy is stable, but the behaviours it supports (the set of methods) is not. Notice that if the set of behaviours is stable, but the set of types is not, plain class inheritance works just fine. Were we to use the same notation as in Figure 4, both situations would be characterized by the same metamodel, missing the essence of the problem.

[1] In our approach, what is not explicitly represented is assumed to be carried over, as is, from problem to solution. Thus, ignoring parameters in this case, simply means that they won't be modified by the application of the pattern, which is true for *Bridge*.

Accordingly, we decided to augment our problem metamodels by specifying those aspects that change. By studying the various kinds of time changes, we were able to reduce them all to changes in the cardinalities of some meta-level associations. For example, both the *Bridge* and the *Abstract factory* pattern handle cases where the number of subclasses of a given class is geared for frequent change. With *visitor* and *decorator*, the number of operations associated with a class is geared for change. *Template method* and *strategy* characterize cases where the number of *implementations* of a given operation is geared for change. And so forth. We represent these "time hotspots" by adding the symbol "++" to the cardinalities on the appropriate association ends. Figure 5 shows the new metamodel of the problem solved by *Bridge*. This model is saying that both the number of abstractions, and the number of implementations *per* abstraction, are geared for change.

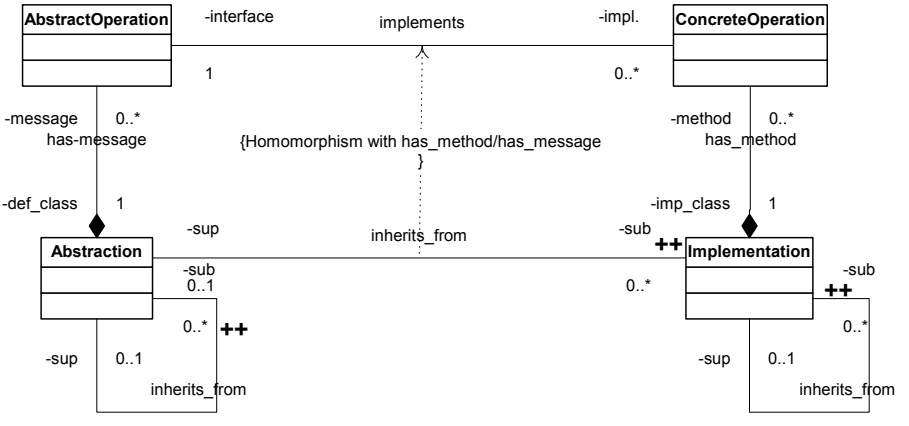

Fig. 5. A metamodel of the problem solved by Bridge, including the time hotspots.

2.4 A Language for Problem Metamodels

The previous example gave us some idea about the kinds of constructs needed by our language. Note that concepts such as *Abstraction* or *Implementation* are not part of the language primitives : the pattern designers (or documenters) can define any metaclass and give it the meaning they want. However, these metaclasses must inherit from the UML subset that is MOF compliant. Thus, while *Abstraction* and *Implementation* are specific to the bridge pattern[2], *because they represent classes, they must both be* (UML) *classifiers*. Similarly, while *AbstractOperation* and *ConcreteOperation* are specific to this pattern, the fact that they represent operations means that they must inherit from the UML/MOF Operation.

We have also introduced the notion of *family*, which represents a set of entities of the same type that share some characteristics, and that can be referenced or handled as

[2] Actually, the notion of *Abstraction* and *Implementation* are used in several patterns, and may be made part of a shared library of metaclasses.

a group. For example, we have the notion of *class family* that represents the set of subclasses of a given class, or what Odell calls *powertype* [14]. We also have the notion of *method families* that represents the set of methods that share some characteristic (name, signature, return type, etc.). Other than these two modifications, our metamodeling language is similar to UML's metamodel. Our EMF™ implementation led us to make some adjustments, as we will see in section 4.

3 Representing the Solution and the Transformation

3.1 Representing the Solution

We used the same principles to represent the solutions produced by design patterns. In this regard, our approach is not much different from metamodel-based representations of design patterns, including [15], [1], [17], and [8]. Figure 6 shows a model of the solution provided by the bridge pattern.

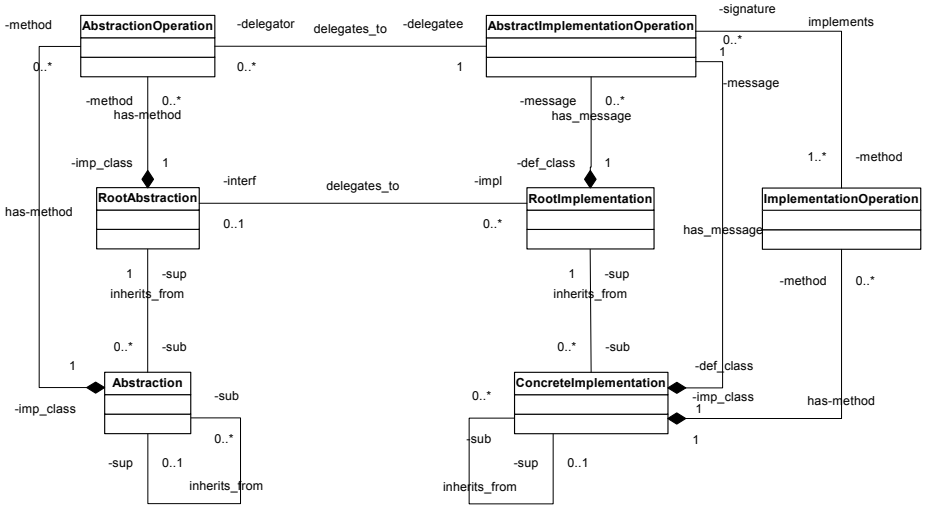

Fig. 6. A metamodel of the solution embodied by the bridge pattern.

The model is read as follows. We have a hierarchy of classes, representing abstractions (*RootAbstraction* and *Abstraction*), that delegates processing to another hierarchy of classes, representing implementations (*RootImplementation* and *ConcreteImplementation*). Note that we need to distinguish root classes from other classes in the tree, for both abstractions and implementations. Indeed, the root of the implementation hierarchy is *an abstract class* while its descendants are *concrete classes* that implement its interface. Interestingly, all of the classes of the abstraction hierarchy are *concrete classes* that delegate their processing to the corresponding methods on the implementation object.

Recall that, as was the case for the problem metamodel, the semantics of the classes *Abstraction* and *RootAbstraction* are specific to the Bridge pattern, and we are free to give them the meaning we want. Further, we don't have to use the same metaclasses that we used to describe the problem, since we will represent the transformation from problem to solution, explicitly. We discuss the representation of transformations in the next section.

The representation of solution models requires additional constructs that are not needed for problem models. One such construct is the notion of *constants* or *literals*. We have no need for literals in the bridge pattern, since all the operations that appear on the solution side come from the problem. However, some design patterns introduce methods and attributes that are supposed to appear as-is in the transformed model. For example, the Observer/Observable pattern requires that observable objects implement pattern-specific operations (notify(…), among others). Our representation language accommodates the representation of literals.

3.2 Representing the Mapping from Problems to Solutions

Applying a design pattern consists of transforming an *instance* of the class of problems solved by the pattern, to an *instance* of the class of solutions. Accordingly, we can *represent* this transformation as a mapping from elements of the problem metamodel (Figure 5) to elements of the solution metamodel (Figure 6). To *apply* the transformation to a sample input model—an analysis or a design-level UML model—we:

1) *first* map the problem (meta)model to the input model, to identify those entities of the input model that match entities in the problem model, and
2) *second*, produce the output model by transforming those so-matched entities (classes, associations, operations) according to the mapping, leaving the others unchanged.

In essence, the first step identifies the entities in the input model that play the *roles* described by the entities of the problem model. This step is typically referred to as *model marking*, and the outcome is a *marked* (input) *model*. In the case of the bridge pattern, we need to identify, in the input model, those classes that play the role of *Abstraction* and *Implementation*. The so-marked classes will be transformed according to the mapping.

Figure 7 shows a mapping metamodel, i.e. a model that represents mappings between problem models and solution models. A <problem model,solution model> mapping is represented by an instance of the class *ModelMapping*. For example, the mapping from the bridge problem model (Figure 5) to the bridge solution model (Figure 6) is represented by an instance of *ModelMapping*. An instance <model1,model2> of *ModelMapping* is an aggregation of, i) mappings between their classes (classes of model1 and classes of model2), and ii) mappings between their associations. In turn, the mapping between two classes (an instance of *ClassMapping*) is an aggregation of, i) mappings between attributes (instances of *AttributeMapping*), and ii) mappings between operations (instances of *OperationMapping*). And so forth.

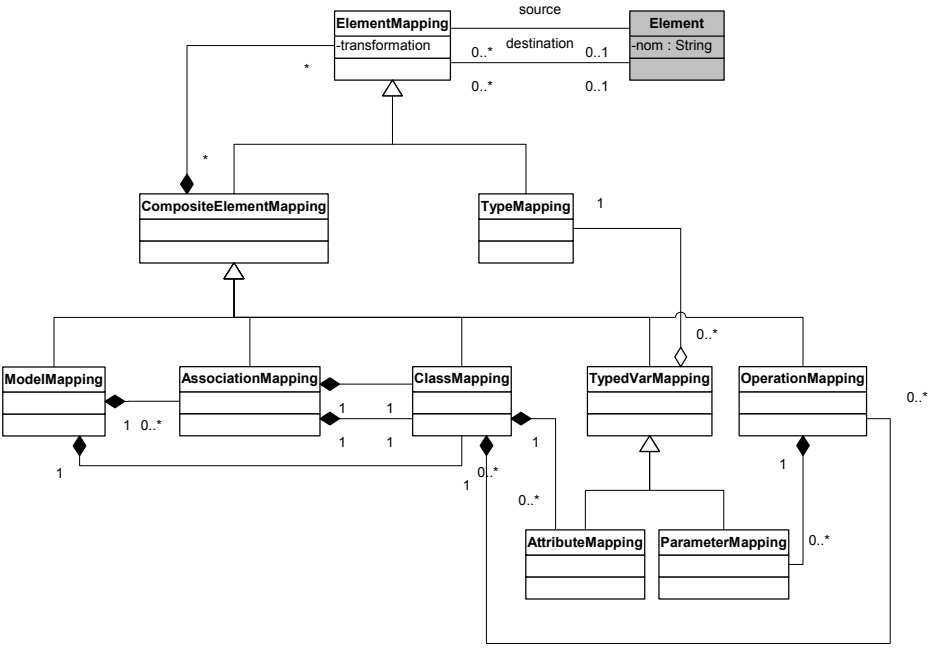

Fig. 7. A model for representing mappings between problem models and solution models.

All of the mapping classes inherit from *ElementMapping*. Each mapping has a source element, a destination element, and a description of how the source element is transformed into the destination element (attribute "transformation" of the class *ElementMapping*). The source and destination are instances of the class *Element* (with grey background), which represents MOF's *ModelElement*. A mapping with no source element means an element that is added by the application of the pattern. A mapping with no destination element means an element that is removed by the application of the pattern.

4 Implementation

We have implemented our representation of design patterns, and the transformation procedure in the Eclipse™ environment using the *Eclipse Modeling Framework*™. EMF is a modeling framework that supports code generation and XMI-based persistence. EMF includes a package—called *ECore*—that provides a simplified implementation of MOF. Section 4.1 describes the implementation of the various metamodels (problems, solutions, and mappings). The transformation algorithm is described in section 4.2.

4.1 Common Metamodel for Problems and Solutions

Figure 8 shows the common metamodel for representing problem models and solution models of design patterns. Initially, we planned to represent this metamodel as *an instance* of EMF's ECore package. This model would then be instantiated to describe problem models and solution models for specific patterns. Those models would, in turn, be instantiated to represent specific application models. Developers, working in the Eclipse environment with the EMF plug-in, would then *load* representations of various patterns (problem models, solution models, and mappings) from secondary storage, and apply them to the application models they are working on. However, EMF's built-in serialization mechanism supports the XMI serialization of only those models that have ECore (or an extension thereof) as a metamodel. Accordingly, instead of defining our pattern metamodel as an instance of ECore, we implemented it using an extension of ECore classes. Figure 8 shows the metamodel, which we will comment briefly. The ECore classes correspond closely to MOF entities (and to UML's meta-meta-model), and are greyed out.

First, in order to define a new metamodel, and thus, a new type of models, we have to define a subclass of *EPackage*, and register the new metaclasses within this subclass. In our case, this subclass is called *ModelPackage*. Figure 8 shows a class *ModelClass*, that extends the ECore class, *EClass*. *ModelClass* represents all the classes that appear within problem models or solution models. In our bridge example, the classes *Abstraction* and *Implementation*, from the problem model, and *RootAbstraction*, *Abstraction*, *RootImplementation*, and *Concrete-Implementation*, from the solution model, are all instances of *ModelClass*.

The class *ModelOperation* is used to represent all kinds of operations, be they virtual (abstract), concrete, or literal. Two boolean instance variables are used to distinguish between the various types: «abstract » and « literal ». Note also that we extended the ECore class *EReference* by our own *ModelReference* class in order to : 1) be able to represent inheritance relationships at the meta level (*SubtypeReference*), and 2) to represent the time variability of the cardinality, i.e. the so-called *time hotspots* (symbol ++ used in Figure 5).

4.2 Implementing the Transformation

We implemented model mappings in a similar fashion to problem and solution models : by extending ECore classes. As for the transformation algorithm itself, it takes three inputs:

1) the input model that we wish to transform, with properly marked entities
2) the mapping between the problem model and the solution model, and
3) the solution model

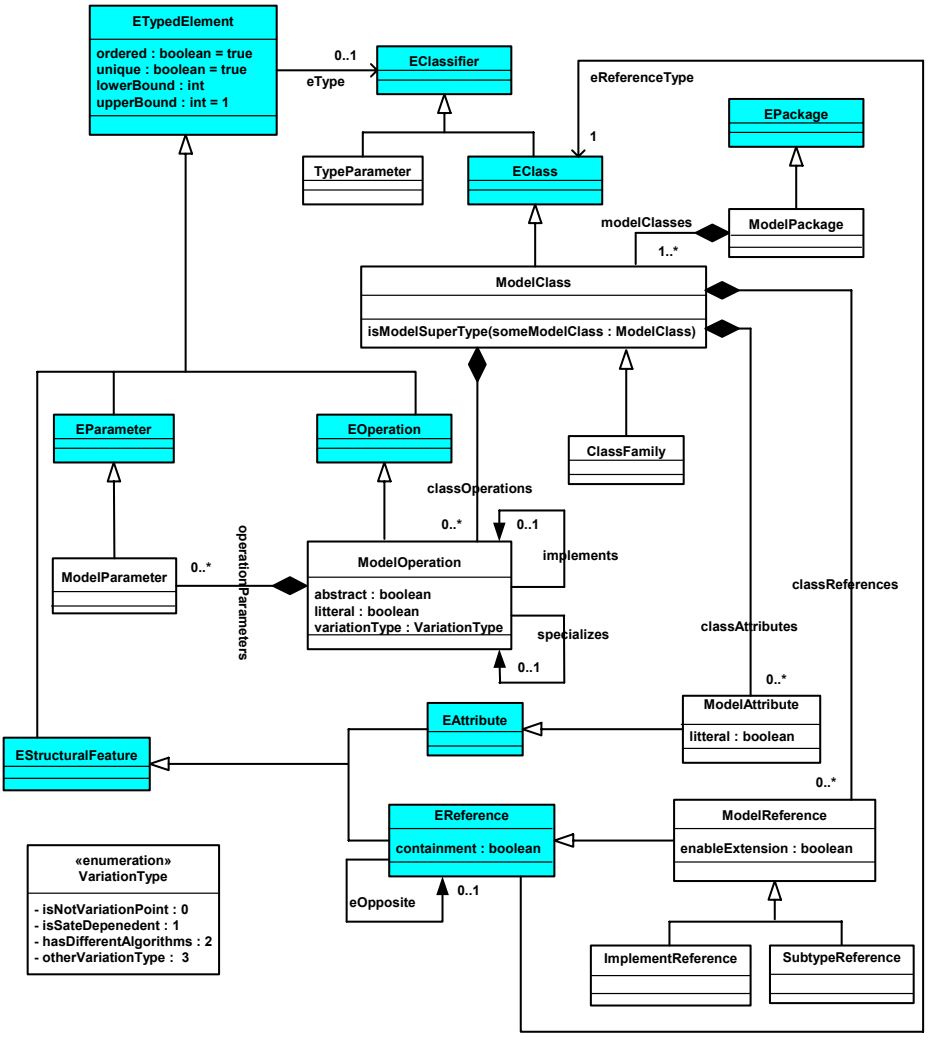

Fig. 8. Implementing the metamodel for pattern problem models and solution models in EMF.

The transformation engine uses a recursive algorithm, starting with aggregate, maps it to an empty aggregate on the destination side, and then recursively maps its components. For example, starting at the highest level, given the marked input model, we first generate an empty destination model, and then transform the classes in the input model, putting their transforms in the destination model. The same is true with classes, where we first generate en empty class, and then map its attributes and operations.

Notice that the same <problem --> solution> mapping may be applied to several entities in the input model. In the bridge pattern, for example, the mapping

<*Implementation* --> *ConcreteImplementation*> will be applied to all the classes in the input model that have been marked by the *Implementation* tag.

5 Discussion

5.1 Related Work

Ever since the publication of the GOF patterns, the representation and application of design patterns has received a lot of attention. Several approaches have been proposed, depending, in part, on the way design patterns are used. In the so-called *top-down* approach [6], developers *instantiate* a design pattern by specifying its components, as in [15] [6] [5] [1] [11] [8]; Budinsky et al.'s work on code generation by pattern instantiation may be seen as a special case [4]. The *bottom-up* approach consists of identifying perfect instances (hits) or imperfect ones (near hits) of specific design patterns, as in [5] [1], [10]. A hybrid approach attempts to re-engineer existing models to make them conform to a specific design pattern, as in [2] [5] [18]. Clearly, each one of these three usages has its own representational requirements.

Those approaches that set out to provide an explicit representation of design patterns were limited to the structural aspects (object model). Some approaches used meta-models to represent design patterns, while others simply offered a set of models, with no concern for a common, pattern-specific metamodel. In either case, the representation focused on the description of the *solution* : what the instantiated pattern will look like. This was the case for most of the top-down, forward-engineering approaches, which used design patterns as design templates that needed to be instantiated. However, we know of no approach that attempted to represent *the problem*; the work of Budinsky et al. may be the exception that confirms the rule [4]: the problem was described using natural language, according to the GOF pattern documentation template [9], but that description didn't lend itself to formal processing. To the best of our knowledge, our work is the only one that attempts to represent the problem explicitly. Such a representation enables us to *formally* characterize those situations where the pattern is appropriate. Such a representation would also enable us to specify the transformations that are embodied in the pattern. The time variability aspect—what we called *time hotspots*—is also unique to our approach, and we consider it to be a *central* aspect of the design problem solved by the design pattern.

With regard to the transformations, only those approaches that focussed on re-engineering existing models with patterns did provide an *explicit* representation for the transformation [2],[18],[19]. However, in such cases, the structure of the pattern itself is not explicit: it is embodied in the transformation. In our case, both the structure of the pattern, and the transformation embodied in its application, are represented explicitly. Further, the transformation is specified declaratively, making it possible to develop a generic, pattern-independent transformation engine (see section 4.2) that takes a marked input model, a pattern mapping model, and a pattern solution model, and produces a properly transformed output model.

5.2 Problem Model Semantics and Marking

In our current implementation, the entities that belong to problem models or solution models (e.g. *Abstraction*, *Implementation*) have no proper semantics. The only semantic constraints are inherent in their type (whether the entities represent a class or an operation) or in the relationships they have to other entities within a given problem (or solution) model. For example, if we look at the bridge problem model, we only know that *Abstraction's* have *AbstractOperation's*, and that *Implementation's* have *ConcreteOperations*, but we don't know what either concept means, beyond the fact that *Abstraction* and *Implementation* are classes, and *AbstractOperation* and *ConcreteOperations* are operations.

One way of capturing the semantics of these entities is to provide membership predicates for them that test a subset of the properties that are typically represented in input models, either directly—stored properties, such as the scope of a feature (instance versus class)—or implicitly –computed, such as the number of associated entities of a particular type. For example, we would define *AbstractOperation* as an operation that is, well, *abstract*, which is a property that is captured by the EMF metamodel—the class EOperation has such an attribute. Similary, *Abstraction* can be defined as a class whose methods are *all* abstract. Clearly, such a definition is useful in many patterns, and may be included in a *library* of such (meta)modeling concepts that are shared between several patterns. Pattern writers and documenters would have the option of using such concepts as is, extending them, or composing them—through mutiple inheritance.

We considered many languages for expressing membership predicates, including OCL, the early drafts of the (upcoming?) QVT standard (Query, View, Transformation), and a number of object-rule languages (e.g. JESS, ILOG JRules, OPSJ). We chose object-rule languages (and JRules in particular), because of their expressive power and because of the availability of mature, high performance tools for interpreting rules on Java objects. The following two rules, used to illustrate the syntax, show two ways of identifying abstract classes. We assume in this case that the class *Abstraction* is stored in a static member of our metaclass *ModelClass*, with the name ABSTRACTION.

```
rule mark_abstract_classes {
when {
        ?aClass: EClass (isAbstract());
} then {
        modify ?aClass {tag = ModelClass.ABSTRACTION;}
        }
}
rule mark_abstract_classes_from_operations {
when {
        ?aClass: EClass();
        not EOperation(isConcrete()) in ?aClass.getEOperations();
} then {
        modify ?aClass {tag = ModelClass.ABSTRACTION;}
        }
}
```

The first rule uses a simple test : the result of (actual) boolean method "boolean isAbstract()" on EClass. The second rule matches any EClass (any class in an input model) such that none of its operations are concrete[3], and marks it as an abstract class. Thanks to rule chaining, we could have tags that depend on complex patterns being built up incrementally, starting with simpler patterns. In fact, the entire problem model itself can be written as one (or several alternate) rule(s) that use pre-assigned tags [12].

There remains one aspect that membership predicates cannot capture: the *probable* evolution scenarios of the input model, which would make a design pattern a desirable alternative. This information is *dynamic* and will not be implicit in the input object model, which provides only a *snapshot* of the target application at the present time. There are two possible strategies for capturing this information. First, we make it a property that designers or analysts will have to enter before they can submit their models for marking. Experienced analysts and designers, with some knowledge of the application domain (e.g. a product line) will know this information but, conceivably, our tool can prompt analysts or designers for potential "time hotspots"—themselves following specific patterns. Second, we can look at consecutive versions of the same software to determine which parts have evolved and how, and use that information to identify the time hotspots. This second approach requires no judgement, but will only work for long-lived software whose source code, throughout several versions, is available.

6 Conclusion

Our work deals with providing developers with a repository of reusable model-based artifacts, and with the tools needed to assist them in using those artifacts. Developing with reuse involves three main tasks, 1) evaluating the opportunity of using an artifact for the problem at hand, 2) understanding that artifact, and 3) integrating the artifact—typically through model transformation—in the system at hand [13]. In this paper, we dealt specifically with the issue of representing and applying/enacting design patterns. Our approach relies on an explicit and precise description of the design problem solved by a given pattern. This description, provided in the form of a meta-model, supports the three reuse tasks.

Our approach is generic and consistent with model-driven engineering. Recognizing the opportunity for reusing a design artifact—design pattern in this case—remains a big challenge, similar to model marking in the context of MDA. One reason is that design problems come from non-functional requirements, which are usually not explicitly represented (or representable with available notations) in software models. To some extent, design patterns are point *solutions*, or *implementations*, for a general design *requirement*: provide model resilience through functional requirements change. Specifically, each design pattern addresses the general design requirement for a specific *functional pattern*, which can be characterized as the combination of a static structure, and an *evolution pattern*. To this

[3] There is no such method on org.eclipse.emf.ecore.EOperation. This is shown for illustration purposes only.

extent, we believe that our representation of design problems, which captures both the static structure of a functional pattern, and its evolution patterns, is a step in the right direction.

References

1. Albin-Amiot, H., Guéhéneuc, Y.G.: Meta-modeling Design Patterns: application to pattern detection and code synthesis. Proceedings of ECOOP Workshop on Automating OO Software Development Methods, 2001.
2. Alencar, P.S.C., Cowan, D.D., Dong, J., Lucena, C.J.P.: A transformational Process-Based Formal Approach to Object-Oriented Design. Formal Methods Europe FME'97, 1997.
3. Baxter, I.: Design Maintenance Systems. Communications of the ACM, vol. 35, no. 4, (1992) 73-89.
4. Budinsky, F.J., Finnie, M.A., Vlissides, J.M., Yu, P.S.: Automatic Code Generation from Design Patterns. IBM Systems Journal, vol. 35, n° 2, (1996) 151-171.
5. Eden, A.H., Gil, J., Hirshfeld, Y., Yehudai A.: Towards a mathematical foundation for design patterns. Technical report, dep. of information technology, Uppsala University, 1999.
6. Florijn, G., Meijers, M., van-Winsen, P.: Tool support for object-oriented patterns. Lecture Notes in Computer Science, vol. 1241, (1997) 472-495.
7. Fontoura, M., Lucena, C.: Extending UML to Improve the Representation of Design Patterns. Journal of OO Programming, vol. 13, n° 11 (2001).
8. France, R., Kim, D.k., Ghosh, S., Song, E.: A UML-Based Pattern Specification Technique, IEEE Trans. on Software Engineering, vol. 30, n° 3, (2004) 193- 206.
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995).
10. Guéhéneuc, Y-G., Sahraoui, H.: des signatures numériques pour améliorer la recherche structurelle de patrons. Proceedings of Langages et Modèles à Objets 2005, Berne, Suisse, (2005).
11. Maplesden, D., Hosking, J., Grundy, J.: Design Pattern Modelling and Instantiation using DPML. Proceedings of 14th International Conference on Technology of OO Languages and Systems (2002).
12. Mili, H., El-Boussaidi, G.: Design patterns : recognizing opportunity through rule-based semantic marking. LATECE Technical report, LAT-2005-12 (2005).
13. Mili, H., Mili, A., Yacoub, S., Addy, E.: Reuse-Based Software Engineering: Techniques, Organization, and Control. John Wiley & Sons, (2002) ISBN 0-471-39819-5.
14. Odell, J.: Power Types. Journal of Object-Oriented Programming (JOOP), (1994).
15. Pagel, B-U., Winter, M.: Towards Pattern-Based Tools. Proc. of EuropLop (1996).
16. Partsch, H., Steinbruggen, R.: Program Transformation Systems. Computing Surveys, vol. 15, no. 3, (1983) 199-236.
17. Sanada, Y., Adams, R.: Representing Design Patterns and Frameworks in UML, Towards a Comprehensive Approach. Journal of Object Technology, vol. 1, n° 2, (2002)143-154.
18. Sunyé, G., Le Guennec, A., Jézéquel, J.M.: Design pattern application in UML. Proc. of the 14th Object Oriented Programming European Conference, (2000) 44-62.
19. Tahvildari, L., Kontogiannis, K.: Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach. The Journal of Software Maintenance and Evolution: Research and Practice, John Wiley Publishers, Volume 16, Issue 4-5, (2004) 331-361.