Expressing variability for design patterns re-use

Nicolas Arnaud — Agnès Front — Dominique Rieu

LSR-IMAG, équipe SIGMA
681 rue de la passerelle
BP 72 38402 Saint Martin d'Hères Cedex
{surname.name}@imag.fr

ABSTRACT. *A design pattern description is much more complex than a semi-formal solution, often restricted to a class diagram. Applying a pattern mainly depends on the solution specification but a lot of useful information can be found into other items. Neglecting variants of the main solution the pattern engineer might explain can be detrimental to the pattern itself because, most of the time, they affect the solution specification. We propose that the pattern engineer represents his solution as a variable mini-system led by a use case view expressing variants. This functional model fragment is the starting point of our imitation process, where application designer will select variants to imitate.*

KEYWORDS: *design patterns, variability, imitation process, UML2.*

1. Introduction

The need for information systems quality implies rigour and continuity during the various phases of a development. Thus, capitalizing knowledge and know-how in order to re-use them during other development processes is crucial. Re-use is regarded as a pledge of this quality, and more particularly if it implements traceability of specifications. In this way, relevant patterns systems are needed. The Gang of Four design patterns (Gamma *et al.*, 1995) are obviously the most famous ones and will be used to illustrate our demonstration.

The reuse process of a pattern, which we called *Imitation*, makes it possible to extract the pattern solution (in our case, software specification fragments) and to apply it into a system under construction. However, a complete specification cannot only be reached by static modelling (i.e. class diagram). Thus, functional and dynamic aspects should be considered, as it's done for classical information system.

Nevertheless, using three views, through a modelling language such as UML (OMG, 2005), does not allow to specify all patterns contributions. In many cases, and particularly for GoF patterns, the pattern description contains information expressing possible variants for the main solution, according to all aspects: functional, dynamic or static. Our main goal is to introduce this variability into the semi-formal solution to allow the imitation process to be traceable and directed, and to prevent bad reuse of a pattern.

We first propose (§2) a more complete specification, of a pattern into mini system in which is grafted a method for expressing its variability (§3). The re-use of such specifications is implemented within an imitation process (§4). The "Observer" pattern is used as an illustration throughout the paper.

2. First approach: a complete mini system

Information systems field is rich of development processes (RUP, 2TUP,…) which combine functional, dynamic and static aspects. We propose a similar approach for specifying the solutions of design patterns as mini systems.

2.1. USE CASES VIEW : THE PATTERN FUNCTIONALITIES

A use cases model presents the system's functionalities, their dependencies and the participative actors. This result of requirements study leads to the whole specification. So, first of all, if a pattern solution provides several functionalities we propose to explicit all of them in the functional view.

Let us remind the intent of the «Observer» pattern: « *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.* » (Gamma *et al.*, 1995)

Two functionalities can be extracted from the « Observer » pattern: *subject modification* and *observers management*. The first one deals with subject state modifications and observers updates while the second allows to dynamically add or remove observers. Figure 1 illustrates the functional view of this pattern.

Here, the *Client* actor squares with the « client » of GoF patterns. The latter specifies the entry points used to accede and trigger the solution's functionalities (see Figure 1).

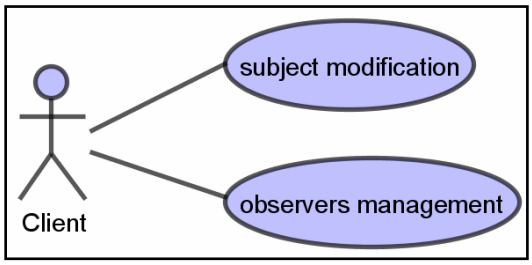

Figure 1. *Observer: use cases view*

2.2. DYNAMIC VIEW : UML2 SEQUENCE DIAGRAMS

In the *Collaborations* item of « Observer », a sequence diagram is given as an example to precise what we called the *subject modification* use case. Figure 2 presents this diagram with UML. The original diagram shows an *Observer* modifying the *Subject* but, in fact, there is no particular limitation about who can use the *setState* method.

We propose two UML2 sequence diagrams to improve the precision of the original sequence. Thus, *notify* sequence (sd *notify*) can be designed using a loop combined fragment. Also, *subject modification* sequence (sd *subject modification*) uses the *notify* method through the gate and interaction use mechanisms. Figure 3 illustrates these new sequences.

Our solution is completed with the *updateSubjectState* method (private) in order to represent the state change of *ConcreteSubject*. The realization of this method is not a concern of the « Observer » pattern, but of the application engineer who will imitate this pattern. This approach is also taken up for the observer's state and *setObserverSate* method. (see Figure 3)

We do not present the whole sequence diagrams mandatory for obtaining a complete functional view.

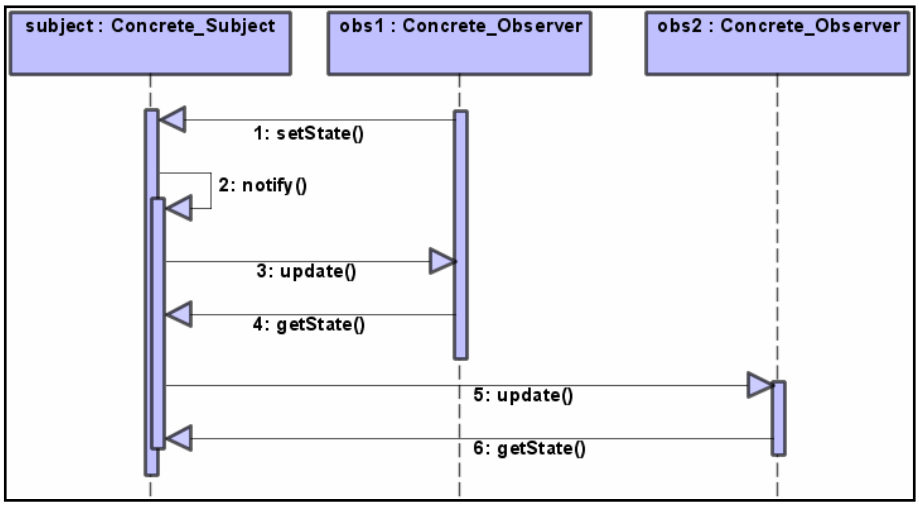

Figure 2. *Observer: GoF original sequence diagram*

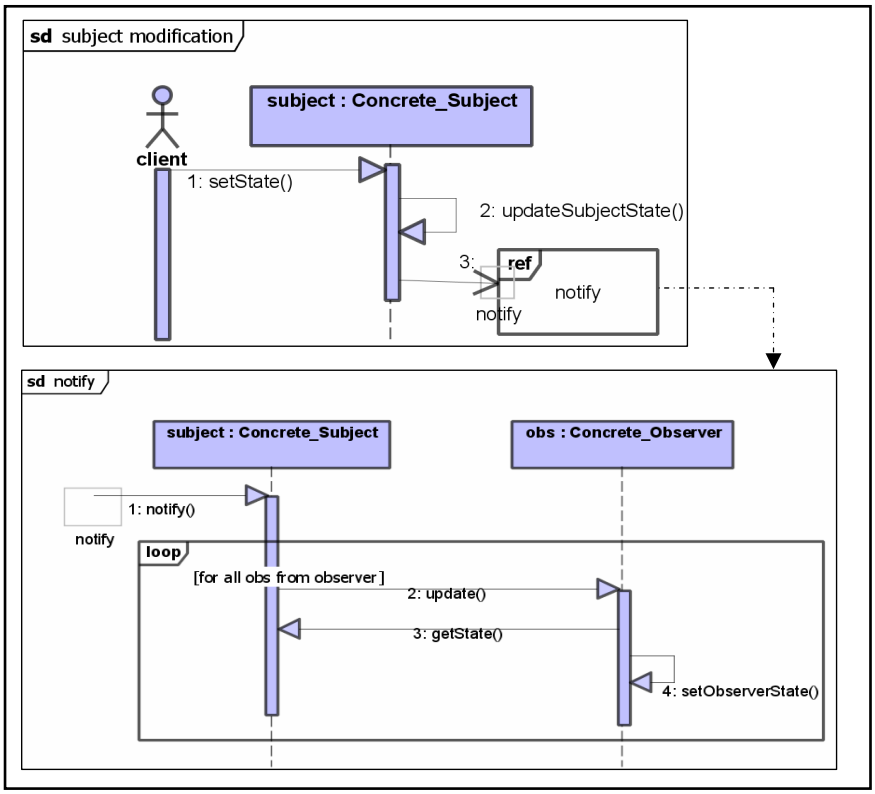

Figure 3. *Observer: UML2 sequence diagrams*

2.3. STATIC VIEW: CLASS DIAGRAMS

Class diagrams specify entities' structure and associations. Although they can mostly be deducted from the dynamic view, the designer must state static features as association cardinalities or properties and methods visibility.

Most of pattern solutions are restricted to a static view, generally a class diagram. These structures often allow textual notes expressing comments or algorithms. For example, « Observer » pattern describes that the *notify* algorithm is using pseudo-code.

Considering the sequence diagrams from the above dynamic view, we propose a new structure (Figure 4, right part) to represent the solution of the « Observer » pattern. Differences with the original solution of GoF (Figure 4, left part) are the followings:

– In this multi-view approach, a specific algorithm is expressed using the dynamic view, as for *notify* method whose explicative note is there redundant.

– Attribute *subjectsate* is no more represented; indeed the subject state may be expressed by many attributes, class links or both, depending on the context. However the application engineer must be informed that he will have to realize the state representation to make his imitation reliable. This is done using a «TODO» note which concerns both the state itself and the associated private method.

It is important to precise that the use of our approach does not avoid original solutions which remain an excellent didactic feature and comprehension enhancer that must be conserved.

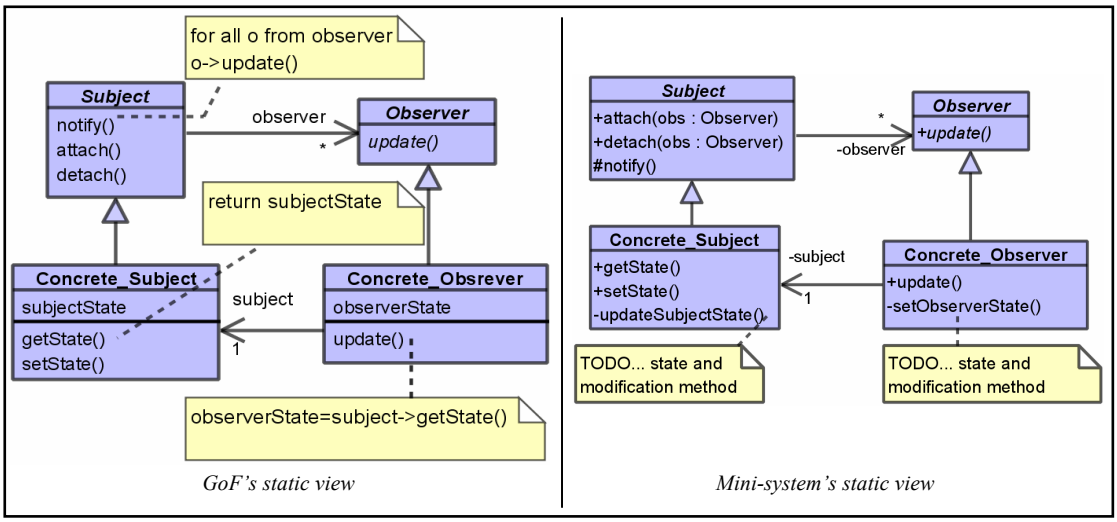

Figure 4. *Observer: static view*

2.4. LIMITS OF MINI SYSTEM

Using multiple views allows us to specify a more complete semi-formal pattern solution. Such a solution could be considered to bring to a better imitation. However, some information still cannot be expressed through our mini-system; for example the fact that the functionalities offered by the solution are essential or secondary.

Thus, contrary to the *GoF* intention, the « Observer » pattern is not restricted to the observers' notification, but also proposes observers attaching and detaching to a subject (*observers management*). The solution should introduce that the observers' management, even if it is useful, is not mandatory for a correct imitation. The following section proposes to express this type of variability of the pattern solution.

3. Variability & patterns

3.1. VARIABILITY AND VARIATION POINT

Variability is defined as the system ability to be changed according to a specific context (Van Grup, 2000).

A **variation point** is a system place where there is a variation (Czarnecki *et al.*, 2000), i.e. where choices have to be made in order to identify the **variants** to be used.

Several types of variability exist for a variation point (Bachmann *et al.*, 2001) :

– *options*: choice from *zero* to *several* variants among several,

– *alternatives*: choice of one variant among several,

– *optional alternatives*: choice from *zero* or *one* variant among several,

– *alternatives set*: choice of at least *one* variant among several.

Many approaches were proposed to represent variability in the specifications: feature diagrams of FODA (Kang and Al, 1990), use cases of (VanDerMaβen and Al, 2002), class diagrams of (Clauss, 2001), sequence diagrams of (Ziadi and Al, 2005), etc. The two last propose UML extensions based on stereotypes. The following sections show how we also use stereotypes on variants from option or alternatives types.

3.2. FUNCTIONAL VARIABILITY

Let us first naturally apply variability on the use cases view. A base operator is used to express the dependency between a variation point (<<variation>>) and one of its variants (<<variant>>). The dependency kind depends on which variability type is required (see Figure 5).

According to the variability type's cardinalities the following imitations are allowed in the Figure 5: [*way1*], [*way1 + option1*], [*way2 + option1 + option2*]. [*way1+way2*] or [*option1*] are prohibited.

In addition, confronting the pattern intention with the suggested solution often allows to extract primary functionalities (mandatory for imitation) and secondary ones. Two other stereotypes (<<primary>> and <<secondary>>) are used to categorize these first-level functionalities. In Figure 5, *functionality A* is primary while *B* is secondary.

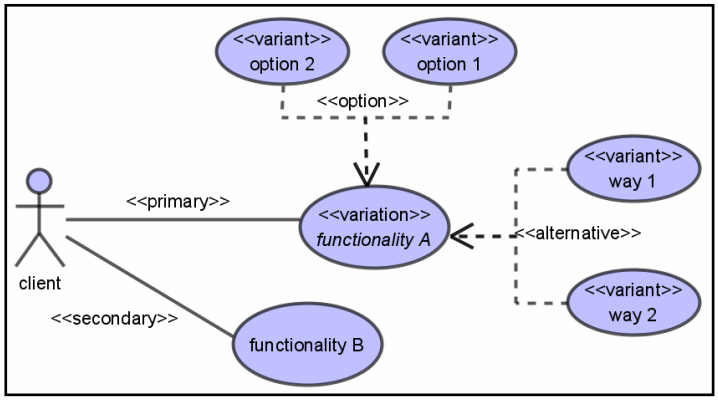

Figure 5. *Functional variability*

Variability on a pattern solution is generally expressed in additional information given by the pattern engineer. GoF patterns more particularly contain an *Implementation* part that describes technical and conceptual variants that fit with the variation point notion.

For example, the primary functionality of « Observer » pattern is the subject state modification including observers automatic update, allowed by an implicit notification. However, *Implementation* item discusses the fact that the state modifier (here the client) can be in charge of notifying the observers, more precisely by calling the *notify* method. According to our approach we can deduct a variation point around *observers notification*, which distinguishes two alternative variants: *implicit notification* and *explicit notification*. Figure 6 illustrates the « Observer » variable functional view.

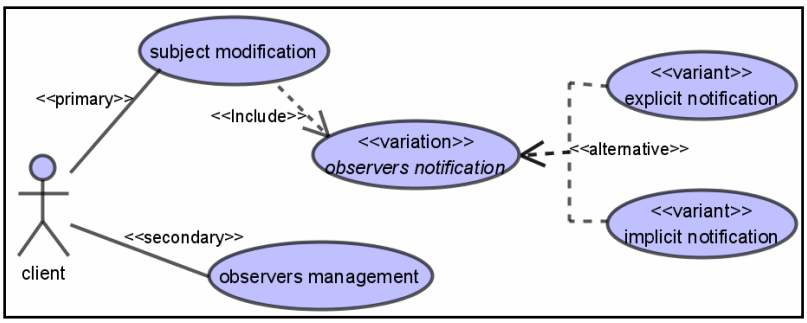

Figure 6. *Observer: variable functional view*

Use cases being specified, it is necessary to complete the mini-system with dynamic and static aspects, considering the previously defined variability.

3.3. DYNAMIC VARIABILITY

UML2 allows sequence diagrams inclusion, thanks to the interaction fragment reference mechanism. This feature is used in our work to express dynamic variability including variant sub-sequences into high-level ones, keeping an overall visibility. It also allows expressing common parts of the dynamic view, i.e. from parts which do not correspond to variant or variation points.

Any variation point from functional view must be represented by an interaction fragment referencing variant interactions (and may detail common sequences). In order to guarantee a coherent specification, a sequence is recommended to have the same name as the use case it explicits. Moreover, <<variation>> and <<variant>> stereotypes are also transposed on the interaction fragments (see Figure 7).

Using this construction rule, a variable dynamic view of « Observer » can be realized. Figure 7 focuses on *subject modification* functionality. The two variants (*implicit notification* and *explicit notification*) reference *notify* (see Figure 3) while the high-level sequence (*subject modification*) contains common messages. In this case, no specific messages are included into the variation point (*observers notification*).

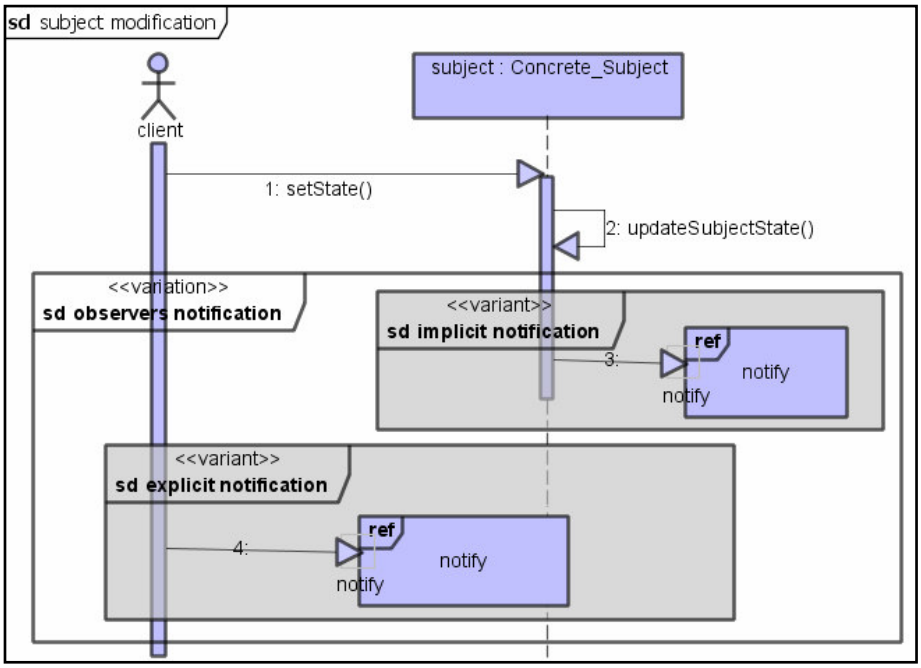

Figure 7. *Variable sequence diagram of "subject modification"*

3.4. STATIC VARIABILITY AND GENERICITY

Static contributions must be considered to complete the specification of the mini system. The proposal here is different from a classical system construction because a static model is given for each specified use case.

Thus, the static structure is disseminated into several fragments which will be assembled to form a traditional static view at imitation time, according to the variant choices made by the designer. Each variant considered in variation or more generally in a functionality should be represented in the corresponding fragment, including the affected properties (attributes, methods, associations…). Note that most of static features can be deducted from the dynamic view. To complete « Observer »'s system, Figure 8 presents static fragments for *subject modification*, *observers notification*, *explicit notification*, *implicit notification* and *observers management*.

In Figure 8, *notify* visibility is different regarding to the considered fragment. An explicit notification implies a public method to allow external call (see message 4 in Figure 7). On the other hand, an implicit notification implies a protected method. It is not possible, at *observers notification* level, to define this method's visibility.

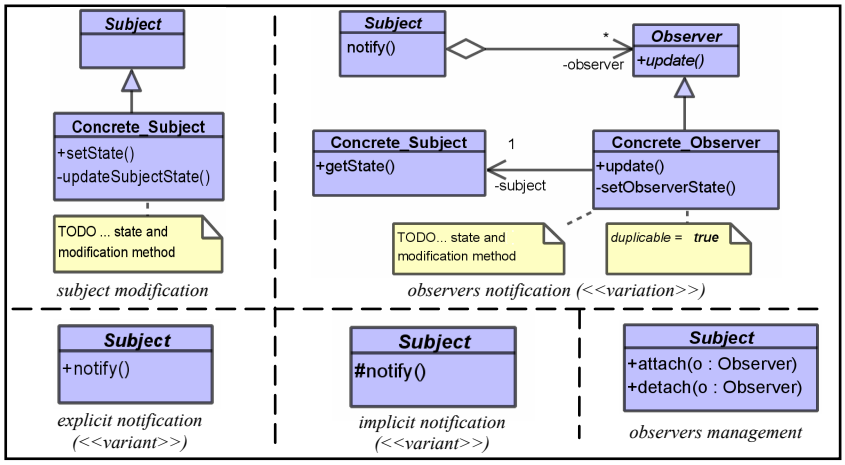

Figure 8. *Observer: static fragments*

These fragments can be re-used in their actual state. While choosing the *implicit notification* variant, *notify* is already protected. During the imitation process, a merge operation between "selected" fragments can construct the re-used static view. However, these class diagrams are not totally locked: as said in 2.3, the information system designer will have to "fill-up" the holes specified by <<TODO>> notes, as for example the subject state properties and private methods.

Other generic properties still need to be expressed, as the fact that a designer may have to define several concrete observer classes. A previous approach, presented in (Arnaud *et al.*, 2004), extends UML elements with meta-properties expressing genericity: for example, a boolean meta-property on class elements, to precise if a class can be imitated more than once (see *duplicable=true,* Figure 8).

4. Imitation process

The previous section has shown how a pattern engineer can add variability and genericity to the pattern solution within what is called an imitable model (mini-system), composed of three complementary views. We present in the following the imitation process which will allow the application engineer to re-use this specification for the construction of an information system.

A traceable process is proposed, which will guide the designer from pattern choice (thus imitable model) to integration of this imitation in order to complete an information system (see. Figure 9). This process is composed of two sub-processes: the reduction process and the application process. Thereafter, only general aspects of these two processes are presented.

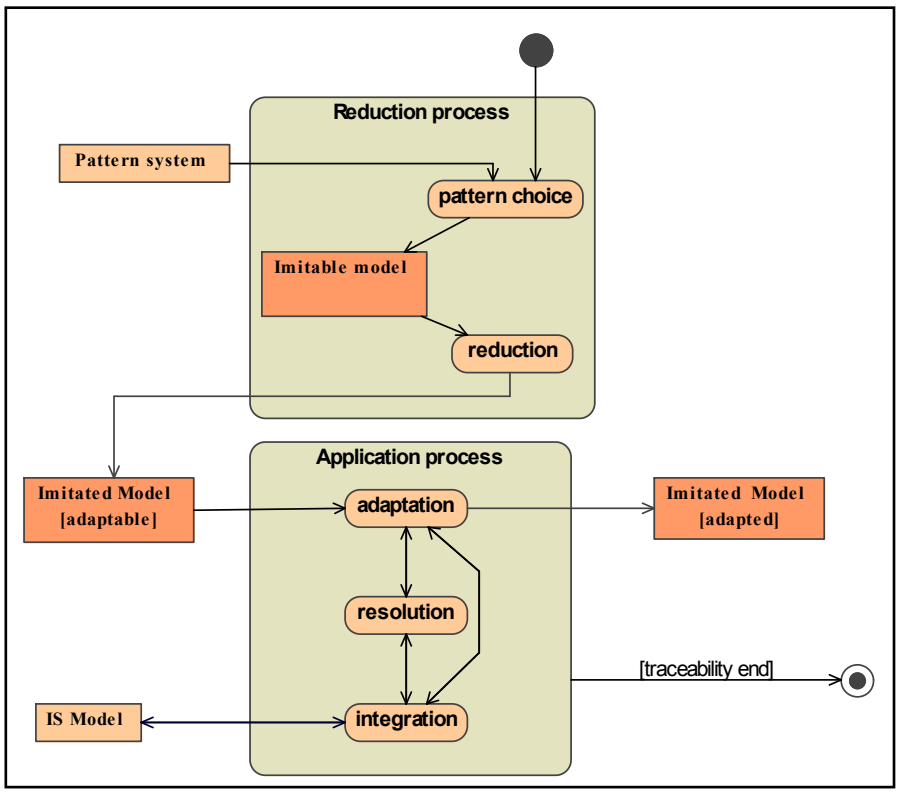

Figure 9. *Imitation process*

4.1. REDUCTION PROCESS

The reduction process is divided into two activities:

– The *Pattern choice* consists in the selection of the pattern the designer wants to re-use. The solution of a pattern is called **imitable model** and is composed of three variable views: functional, dynamic and static (see §3).

– The *reduction* activity allows the designer to choose, through the functional view, the variants he wants to imitate, evicting the un-useful ones (Figure 10, top-left corner). This selection implies other evictions in dynamic and static view of the imitable model (Figure 10, middle). It then allows, for each view kind, to automatically **reduce** the **imitable model** into an **imitated model** (in the adaptable state) corresponding to the specific variant combination the system designer needs (Figure 10, top right corner and bottom). We can consider the **imitated model** (in **adaptable** state) as the pattern's better solution for the designer problem.

4.2. APPLICATION PROCESS

This process is composed of three activities which can be iteratively executed.

– *Adaptation* allows the designer to adapt the imitated[adaptable] model while remaining conform with genericity rules. For example, the class names might fit with the imitation context. In Figure 11 *Subject* was renamed into *Distribution* and *Concrete_Observer* has been imitated into *HistoDiag* and *SectorDiag* (duplicable=true). The obtained model is called **imitated model**, in **adapted** state.

– *Resolution* consists in TODO notes treatment. In our example it can deal with implementing the subject state and its private modification method (see Figure 4, right part). In Figure 11, this note has been resolved thanks to four attributes (*north*, *south*, *east*, *west*) and a complete specification of *updateValues()*. Figure 11 also shows that some resolution may be realized only during integration activity.

– *Integration* merges imitated[adapted] model specification into the target information system. An information system is thus seen as a whole set, but also as original specifications corresponding to the know-how of the designer. Integration has been partially treated in (Arnaud *et al.*, 2005) where two integration operators are proposed: by delegation and fusion.

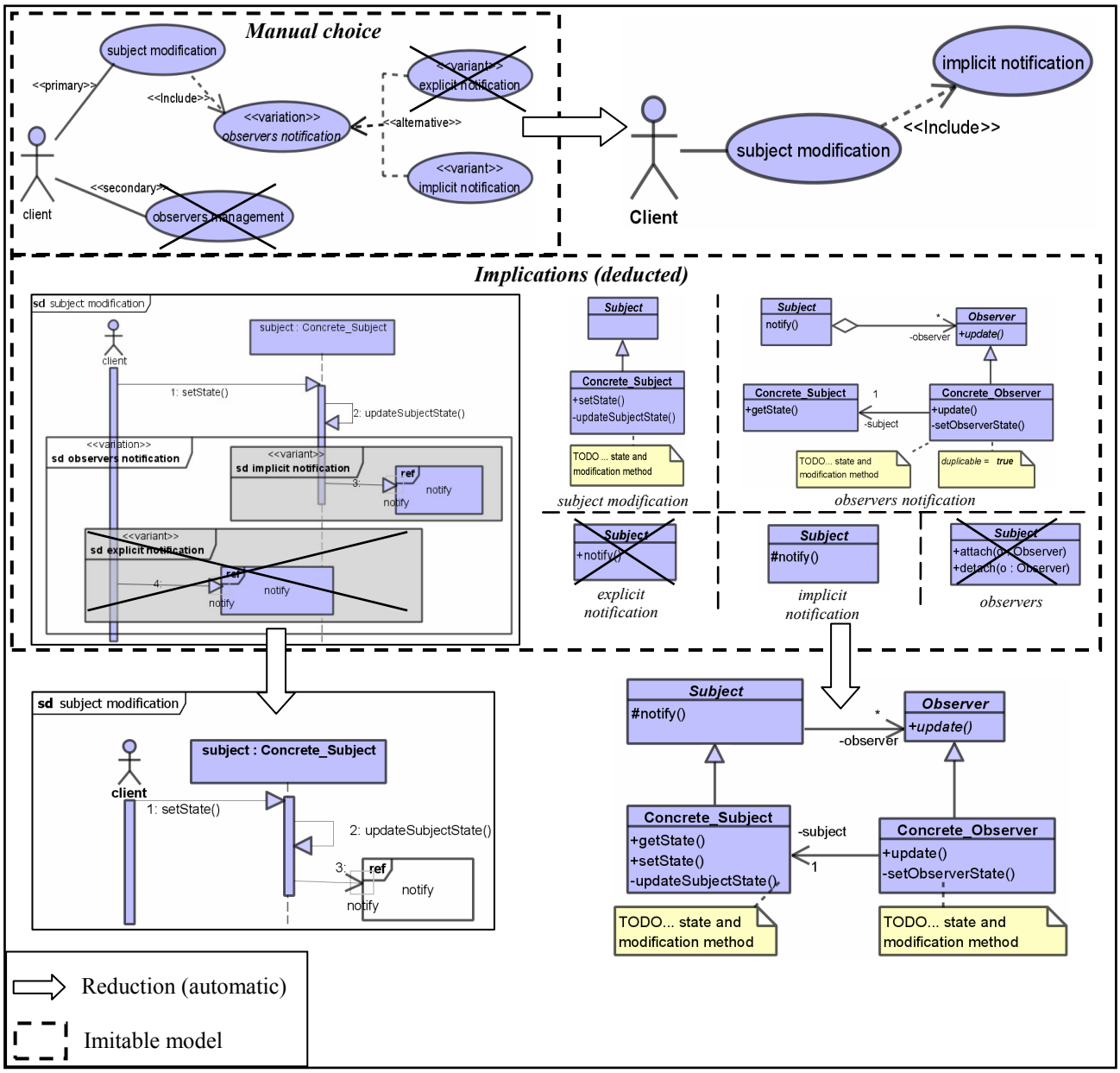

Figure 10. *Reduction of Observer's imitable model to an imitated[adaptable] model*

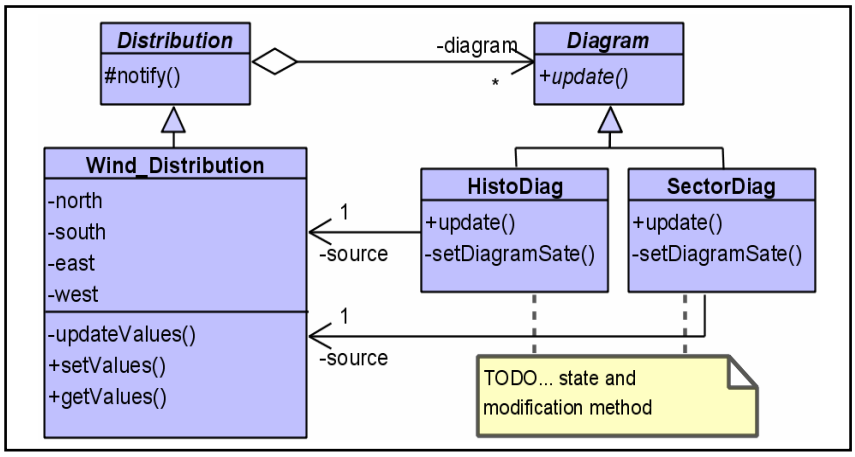

Figure 11. *Static view of a imitated[adapted] model*

5. Related works

Works aiming at pattern solution enhancement in order to guarantee a correct re-use can be characterized among several criteria. Three of them are presented here.

Specifications completeness. In most works (Meijler *et al.*, 1997), (Arnaud *et al.*, 2004), the only static aspects (classes, attributes, associations) are focused. (Albin-Amiot *et al.*, 2001) consider some dynamic aspects thanks to a more pattern-specific UML-based meta-model. For instance, the meta-class *PDelegatingMethod* is used to specify that a method realizes delegation. Code generation is there an assumed goal. In (France *et al.*, 2004), static view (Structural Pattern Specification) and dynamic view (Interaction Pattern Specification) are jointly processed.

Variability expression. (Budinsky *et al.*, 1996), (Sunyé, 1999) and (Le Guennec *et al.*, 2000) deal with pattern implementation variants. These works clearly correspond to our *reduction* activity. Although functional variability has not been used for pattern-based applications it is actually explored in research fields like product lines (Ziadi *et al.*, 2005) or requirements engineering (Bennasri, 2005),

UML metamodel extension. Due to length restrictions our UML extensions proposed to express variability and genericity have not been detailed. Many approaches (Meijler *et al.*, 1997), (France *et al.*, 2004) lead to a specific extension for each pattern. The goal here is to extend UML with general concepts to be applied on any pattern.

6. Conclusion

This paper proposes a re-use process for object-oriented design patterns considering both generic and variable aspects of the solutions. The first benefits of our approach deals with pattern engineer's job, allowing him to complete the textual discussion about his pattern's variability expressing it within the solution itself. In the other hand, he also can specify generic aspects within the pattern solution. Theses generic and variable solutions are expressed using a three-view (functional, dynamic, static) mini-system called imitable model.

Thanks to the functional view (a use case model) the information system designer can *reduce* the imitable model by selecting some specified variants. This operation provides him a specific solution (called imitated and adaptable model) he can *apply* to his context through three activities: *adaptation* (directed by generic properties), *resolution* (completing specification holes) and *integration* (introducing the imitation result into a target information system).

The imitable model specification requires a notable investment from the pattern engineer. Nevertheless, it is also profitable for identifying and structuring the variable facets of his main solution and maybe reconsidering the main problem, enhancing the pattern reusability spectrum. From re-users side, a system designer that re-uses such patterns is guided through a highly automatizable imitation process bringing traceability (imitable model, imitated model with adaptable and adapted states) and allowing intelligent rollbacks.

Our further work will deal with imitation process instrumentation (using model transformation approaches) and its generalisation to analysis and business patterns.

7. References

Albin-Amiot H., Guéhéneuc Y.G., « Meta-modeling Design Patterns : application to pattern detection and code synthesis », *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, June 2001.

Arnaud N., Front A., Rieu D., « Deux opérateurs pour l'intégration d'imitations de patrons », *Congrès INFORSID'05*, Mai 2005.

Arnaud N., Front A., Rieu D., « Une approche par méta-modélisation pour l'imitation des patrons », *Congrès INFORSID'04* , Mai 2004.

Bachmann F., Bass L., « Managing variability in software architecture », *ACM SIGSOFT Software Engineering Notes*, Volume 26, n°3, Mai 2001

Bennasri S., Une approche intentionnelle de représentation et de réalisation de la variabilité dans un système logiciel, Thèse de doctorat, Université de Paris I, Février 2005.

Budinsky, F.J., M.A. Finnie, J.M. Vlissides et P.S. Yu, « Automatic code generation from design patterns". *IBM Systems Journal*, 1996

Clauss M., « Generic modeling using UML extensions for variability », *OOPSLA 2001, Workshop on Domain Specific Visual Languages*, pages 11-18, Septembre 2001.

Czarnecki K., Eisenecker U. W., *Generative Programming – Methods, Tools and Applications*, Addison-Wesley, 2000.

France R.B., Dae-Kyoo K., Sudipto G., Eunjee S., « A UML-Based Pattern Specification Technique », *IEEE transactions on software engineering*, vol. 30, no. 3, March 2004

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Element of Reusable Object-Oriented Software*, Addison-Wesley professional computing series, 1995.

Kang K., Cohen S., Hess J., Novak W., Peterson S., Feature-Oriented Domain Analysis (FODA) feasibility study, Technical report CMU/SEI-90-TR-21, Software Engineering Insitute, Carnegie Mellon University, Novembre 1990

Le Guennec A., Sunyé G., Jézéquel J.M., « Precise modeling of design patterns », Proceedings of UML 2000, volume 1939 of LNCS, pages 482--496. Springer Verlag, 2000.

Meijler T.D., Demeyer S., Engel R., « Making design patterns explicit in Face », *European Software Engineering Conference*, 1997.

Object Management Group, « Unified Modeling Language : Superstructure », version 2.0, Août 2005.

Sunyé, G., « Génération de code à l'aide de patrons de conception », *Langages et Modèles à Objets - LMO'99*, Villefranche s/ mer, 1999.

Van der Maβen T., Lichter H., « Modeling variability by UML use case diagrams », *International Workshop on Requirements Engineering for Product Lines (REPL'02)*, pages 19-25. AVAYA labs, Septembre 2002.

Van Grup J., Variability in Software Systems, the key to software reuse, Licentiate Thesis, University of Groningem, Sweden, 2000.

Ziadi T, Jézéquel J.M., « Manipulation de lignes de produits logiciels : une approche dirigée par les modèles », *Ingénierie Dirigée par les Modèles (IDM'05)*, Mai 2005.