Representing Variations in Structure Diagrams Using Conditionally Scoped Associations

M. Naci Akkøk

Department of Informatics,
University of Oslo,
Postboks 1080 Blindern, 0316 Oslo, Norway

nacia@ifi.uio.no
http://heim.ifi.uio.no/~nacia/

Abstract. Structural models like class diagrams, component diagrams or network diagrams are usually not equipped for representing temporal and/or configurational variations needed in modeling product lines, frameworks, interfaces or network connections that change over time (as in certain types of mobile and ad-hoc networks). This paper introduces the concept of the scope of an association indicating that relationships (including interfaces and other types of connections) are not always static but may be valid under certain conditions only, and offers a notation that allows for representing such conditional change in associations between classes, valid also for many types of connections in various kinds of structural diagrams. An example from the domain of post-development (legacy system) integration is give, both because it is the case that originated the ideas in this paper, but also for demonstrating how the concepts and the technique can be used in other areas like designing for re-integration as well.

Keywords. Modeling variation points – scope of associations – conditional associations – modeling legacy system integration – frameworks – product lines

1 Introduction

Design-time representation of variation, configuration and reconfiguration at the structural level – as for example when modeling product line architectures or application frameworks using structural models like class diagrams – are problematic. Similar issues exist in representing post-development (legacy system) integration as exemplified later on in this paper, in network connections that change over time and in component-based systems that need to be configured differently depending upon certain conditions.

In all these cases, the issue is to be able to express conditional or temporal change in structural models that usually do not lend themselves for representing change except maybe minimally.

This paper introduces the concept of the *scope of an association*, which indicates that relationships in structural models are not static but are actually valid under certain conditions only, and that they may be established, dissolved and reestablished over time. It then offers one possible (UML-friendly) notation for representing such conditional change in the associations between classes in a class diagram, which can be used for representing change in all the types of structural models mentioned earlier.

1.1 Notations for Representing Variations in Frameworks and Product Lines

There exist some notational solutions for representing variations in frameworks and product lines through use of object-oriented (OO) mechanisms like inheritance (related to the concepts called generalization/specialization in UML) and abstract classes. See
Figure **1** for an e-shopping framework class hierarchy example (taken from Fontoura et al. [1] with minor modifications).

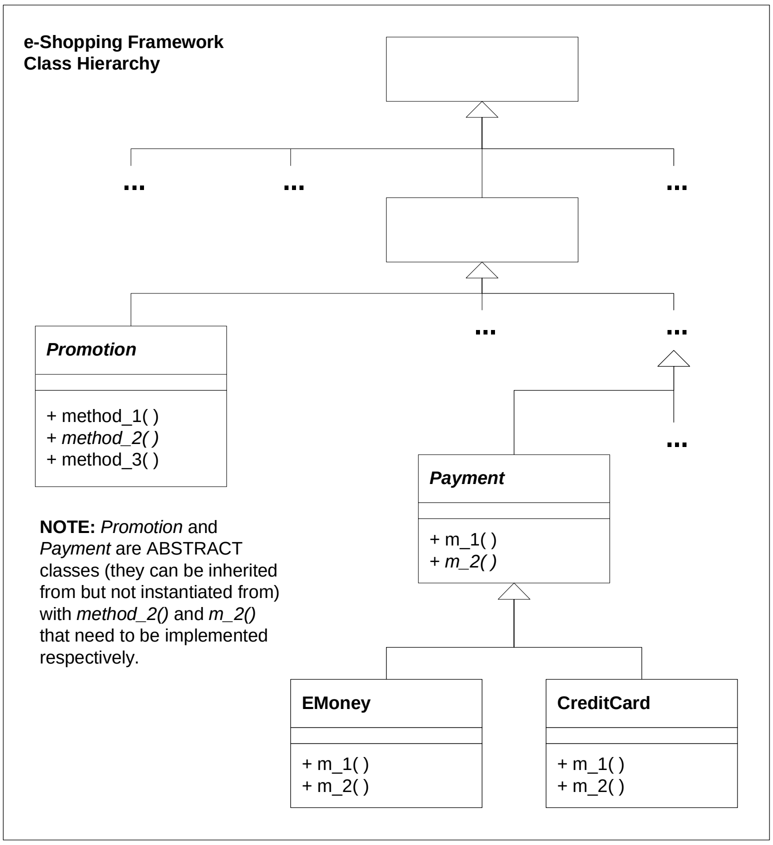

Figure 1. Example of a framework class hierarchy

Note that two of the classes (*Promotion* and *Payment*) are abstract classes that must be inherited from and implemented, which provide the *variation points* of the framework.

Note also that there is a difference between a *framework class hierarchy* and an *application framework*. The framework class hierarchy offers all potentially useful classes in an inheritance hierarchy. It may be compared to a domain specific class library with variation points to allow for further specialization of certain classes. The application framework, however, is “almost” an application – or, more precisely, an application class diagram except for the classes that will be replaced or further specialized in the final configuration. Figure 2 exemplifies an application framework derived from the e-shopping class hierarchy of

Figure **1**.

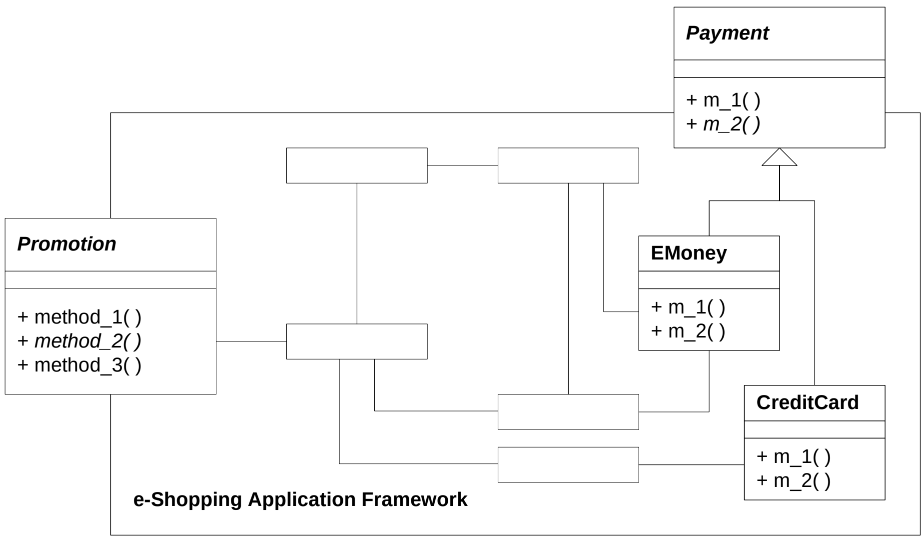

Figure 2. Example of an application framework with variation points

The application framework in Figure 2 is capable of allowing for two specializations (variation points): in the case of the abstract class *Promotion*, it would first be inherited from and its *method_2()* will then be implemented, generating a concrete class for the specific promotion need of an application. The framework also offers the abstract *Payment* class where two concrete implementations of the abstract class are already provided in the form of class EMoney and class CreditCard. Thus, when the application designer decides to use only electronic money as the payment method in the application, the EMoney class will be chosen instead of CreditCard as *Payment*. Figure 3 shows the same application framework that has generated a concrete application.

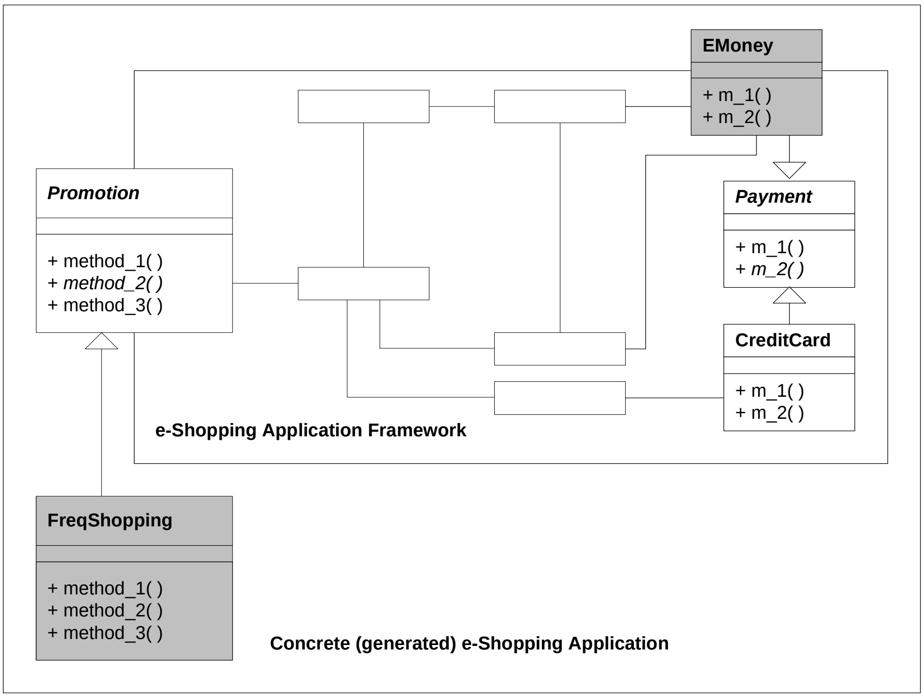

Figure 3. Example of an application framework that has generated a concrete application

Note that the issues related to representing variation points apply to product lines as well, though the examples we have given are for frameworks.

The technique for representing variations as demonstrated in Figure 2 and Figure 3 solve only certain issues. There are categories of issues that cannot be solved using these mechanisms. One such example is based upon a case that is inspired from a European project called "Single Point of Access for Citizens of Europe" abbreviated SPACE [2].

1.2 Example Case

The Maastricht Treaty granted EU citizens the right to move freely across boundaries within the Union (at that time called the "European Community"), which implied the possibility to move the citizens' personal and social data with them. As an example, a person from one of the treaty countries who wanted to travel to or settle in another treaty country would ideally still have access to his/her full medical data, his/her unemployment pay, his/her retirement, the aid he/she had right to etc. The essence of the SPACE project was to study the technical enablers for making this possible. Some of the essential problems represented by the SPACE project are the following:

– How do we integrate systems and their data across (system) boundaries when the systems in question are already designed, developed and in production (i.e.,

when they are no longer possible to influence at design and/or development time)?
- How do we represent (possible) changes in structural (static) relationships between systems and their data, while enabling integrity, avoiding unplanned and undesirable replication, redundancy etc?

The first question is a heavily studied and well-known issue with many partial answers as for example the answers offered through protocols like SOAP [3, 4], other XML-based technologies that enable exchange, storage and retrieval of semi-structured data [5-7], solutions derived from heterogeneous database systems as available in many database textbooks [8], and solutions that use a mix of these for integrating applications through integrating their data [9]. Our issue is the second question, but our answer will offer some answers for the first question as well.

As we noted above, the ability to represent planned and/or ad-hoc changes in structural relationships is instrumental for other areas like product line architectures and frameworks as well. Thus, an answer to the second question is really an answer to a range of issues. As a matter of fact, we have already seen one potential answer to the second question using frameworks, but a framework that builds upon existing constructs does not provide a satisfying model for the (simplified and re-worded) specification below:

A person is employable when he/she is off-age and not under education or in military or civil service. An employable person can register at the unemployment agency if the person is yet unemployed or unemployed after a period of employment. This is not valid for the disabled, whom are registered at the aid agency instead.

Valid reasons for unemployment after a period of employment may be being fired, quitting, being disabled, the company being bankrupt or the company being dissolved of other reasons.

Note that Person, Unemployment Agency, Aid Agency and Company referred to may really be representing four distinct information systems, each part of a separate piece of software, possibly in their own distinct organization and/or country as SPACE implies, i.e.:

- Person may represent a national registry system or equivalent, if such exists, or person registered in company and/or other agency.
- Company may represent a company's information system or some public registrar's information about companies.
- Unemployment Agency and Aid Agency may represent their own internal information systems respectively.

The specification above will usually be modeled using classical structural diagrams (at least three distinct ones in this case, one for each of the three "systems" as in Figure 4) that would not allow for change or behavior made visible at the structural level at design time (if there was indeed a need during design of individual systems) or integration time:

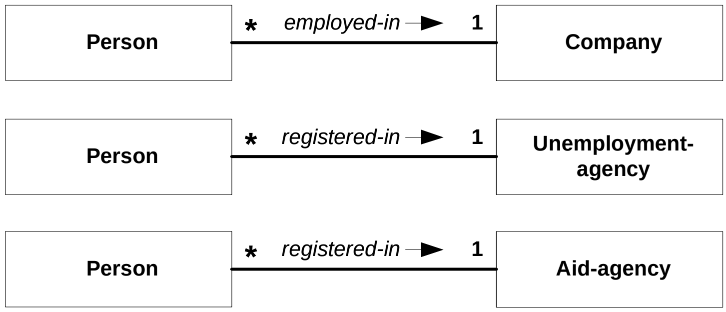

Figure 4. Models of three distinct systems where (semantically shared) *person* participates throughout *person*'s life-cycle

2 Scope of an Association and its Representation

Person is shared in the systems represented in Figure 4 and a potential threat to integrity in general, and a potential semantic headache in case the three "systems" are to be integrated. It would have been desirable to model the system so that the role of person is visible from the start (or at least at integration time), as well as how the three systems relate to each other. Since the same person is not necessarily part of all three systems all the time, spaghetti-like connecting lines and heavy-duty constraints are needed to model the system with existing modeling constructs.

To be able to offer a better modeling construct, let us fist note the constraints: a person's existence in one of the three systems depends upon certain events and/or conditions as noted in Figure 5:

States with their events and guards (conditions) are inherent to a *class* in major OO modeling languages like the UML, implying a statechart per class. This information reflects the behavior of the instances of the class. It is information that is usually hidden at the level of the structural model, but we obviously need just these conditions/events at the structural level if we want to express the *conditional* participation of person objects in associations in accordance with Figure 5.

One potential solution is to use constraints (for example OCL) at the structural modeling level, reflecting the conditions and events governing the participation of a person in the various associations. This is doable, but there are concerns.

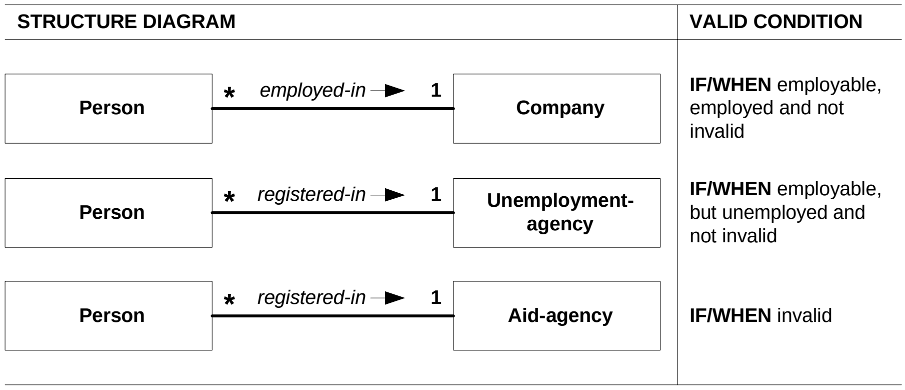

Figure 5. Conditions/events under which one of three distinct systems are valid for any given *person* at any given time

One concern is that OCL is not a diagrammatic modeling language (DML) like the diagramming languages of UML and other DMLs. This is valid for most formal constraint languages. Though there are efforts to create diagrammatic notations for OCL (see for example [10], pp. 125-129, and [11]), they are still not fully developed.

Another concern is that we will be using two distinct DMLs and a constraint language to be able to express structure as well as events and conditions. That is an unnecessary cognitive load. It would in addition be a source of error since it requires event/condition information to be maintained in the constraint language and in the statechart. A sufficiently intelligent computer-based modeling tool could help, but then we would have to introduce precise transformation rules back-and-forth between constraints and statecharts.

There is an easier and obvious alternative: we simply pull the relevant class' state information up to the structural diagram level as demonstrated in Figure 6. A brief explanation of the notation and a definition follows:

- Each class is assumed to have a default state. As an example, the person is by default *employed*, whereas the company is by default *active*.
- Not all classes have to have their states visible as in the case of unemployment agency and aid agency.
- Not all states of a class need to be visible.
- Which classes carry which states to the structure diagram is derived from the specification. The same information is also available in Figure 5, which offers the conditions/events that make the association valid for each association.
- We use short bars (called *transition bars*) cutting the transition line for differentiating between a transition and an association.
- We use the exclamation mark to designate an event.
- Events/conditions can be combined in logical expressions as in the case of !*quit* $\vee$!*fired*.

– One can scope names (also events/conditions) using a prefix notation as in the case of *Company*!*dissolved* ∨ *Company*!*bankrupt*. Note why this is necessary: a company going bankrupt or being dissolved is a condition for the persons in the company being unemployed. Naturally, the association between these persons and the company ceases to exist, since the company has no associations to any other classes in its dissolved or bankrupt states.
– Note that *Company!dissolved* designates an event that a Company instance has received and triggered upon, which (in this case) a Person instance also may react to.
– There are no statechart-style guards (conditions) shown in Figure 6 but they are allowed. See [10] pp. 259-271 for a more detailed discussion of statechart transition syntax and semantics.
– We distinguish between associations and transitions by using thinner arcs for transitions (in addition to the transition bar) and thicker straight lines for associations.

Based upon the notation and concepts presented up to this point, a somewhat more precise definition of the *scope of an association* would be "the conditions/events under which an association is valid".

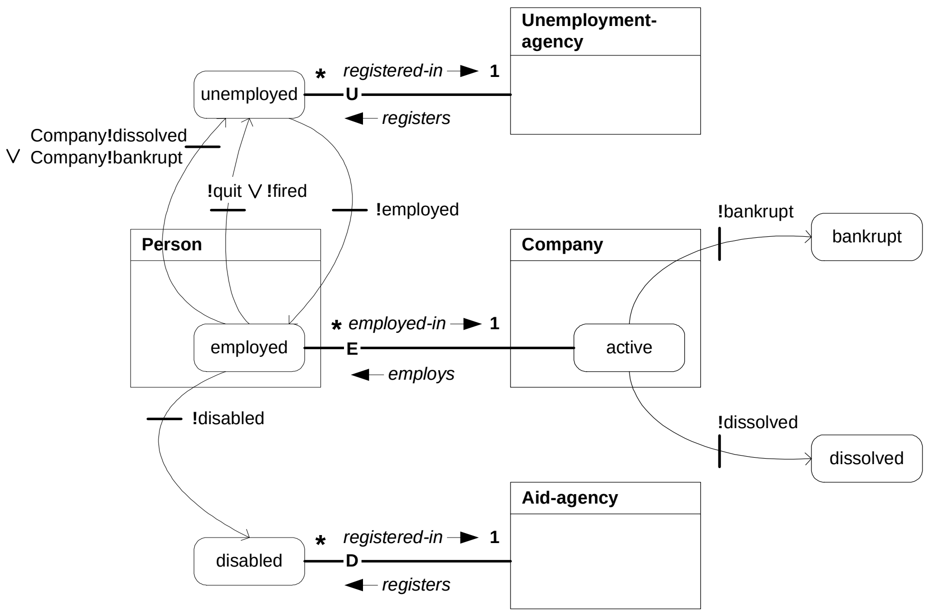

Figure 6. Integrating structural models of three distinct systems through using state information at the structural level

The associations are between a class in any one of its given states and another class (can be the same class) in any one of its states. A class that does not need its states at the structural model level will participate in an association as usual. As an example, the unemployment agency will register persons in their unemployed states regardless of the state of the unemployment agency.

We refer to the type of diagram depicted in Figure 6 as a conditional structure diagram (CSD).

3 Comments and Conclusions

The clear advantage of a CSD is that it shows all associations together, providing a proper overview. It does this using a behavioral DML (statechart in this case) that is integrable with the structure diagram DML. It also offers a mechanism by which an association's validity is scoped, offering variation not only in classes as in frameworks depicted in Figure 1, Figure 2 and Figure 3, but also in associations. The usual variation points – abstract classes – can still be used together with scoped associations, increasing the power to express change and the conditions of change in a structure diagram (as for example in an architecture) considerably.

When designing this notation we have reverted to principles implicit in disciplines like linguistic pragmatics and communication: a statement about a thing is always understood (and hence formulated) with respect to the receiver's (assumed) understanding of the context of the thing, its nature, its capabilities etc, which also includes the thing's state as we interpret it.

Obviously, an object's (thing's) state may then effect its participation in an association, which has been assumed to be static. Since we know that each class (representing a number of objects) offers a statechart (representing the behavior of the objects of this class), we already had the mechanism for putting into practice the pragmatic principle: all we had to do was to make these states visible at the class-level.

Note that our design in a sense "unifies" the structural and the behavioral in a DML suite like UML to some degree. This would not have been possible if notations overlapped and the same symbols were used in the various DMLs of UML to mean various things. This points out the importance of notational consistency when designing DMLs – and especially suites of DMLs that may eventually need to be integrated in some manner.

One issue related to the CSD needs to be mentioned: CSD will not scale well unless it allows for diagramming at several levels of abstraction/detail that also can be distributed across pages. We know that statecharts allow for super-states and sub-states. Many structural modeling notations allow for similar breakdown paths, but a UML V1.5 class (for which the specification is available at [12]) does not break down to classes, and cannot be structurally decomposed in any other manner either, except through aggregation and composition that often need to be shown with the class. This issue seems to be addressed to certain degree (but primarily in relation to components) in UML V2.0 as implied in Björkander and Kobryn [13].

Note that the technique presented here is an experimental notation tried out only in small scale, in few cases, and in academic set-ups. It should not be forgotten that the notation still needs time to mature.

References

1. Fontoura, M., W. Pree, and B. Rumpe, *The UML Profile for Framework Architectures*. Component Software Seriers, ed. C. Szyperski. 2002: Addison-Wesley (imprint of Pearson Education).
2. Holmes, P., et al. *SPACE: An architecture for coordinated intra-european assembly and exchange of citizen data*. in *Proceedings of CE98: Advances in Concurrent Engineering.* 1998. Tokyo.
3. W3C, *SOAP home site*. 2004: http://www.w3.org/TR/soap/.
4. W3C, *SOAP primer*. 2004: http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.
5. W3C, *XML home site*. 2004: http://www.w3.org/XML/.
6. W3C, *XML Query home site*. 2004: http://www.w3.org/XML/Query.
7. Bourret, R., *XML Database Products*. 2004: http://www.rpbourret.com/xml/XMLDatabaseProds.htm.
8. Garcia-Molina, H., J.D. Ullman, and J.D. Widom, *Database Systems - The Complete Book*. 2002: Prentice Hall.
9. Andersen, E.P. *A UML-based Architectural Framework for Web-based Information Systems*. in *Proceedings of ICSSEA'2000: 13th Int'l Conf. on Software Systems Engineering and Applications, December 5-8*. 2000. Paris.
10. Akkøk, M.N., *Towards the Principles of Designing Diagrammatic Modeling Languages: Some Visual, Cognitive and Foundational Aspects*, in *Institute of Informatics, Mathematics and Natural Sciences Faculty*. 2004, University of Oslo, Dr.Scient. (PhD) Thesis, ISSN 1501-7710.
11. Gil, J., J. Howse, and S. Kent, *Formalizing Spider Diagrams.* Proceedings of the IEEE Symposium on Visual Languages (VL99), 1999: p. 130-137.
12. OMG, *Unified Modeling Language Specification V1.5, March 2003*. 2003: http://www.omg.org/docs/formal/03-03-01.pdf.
13. Björkander, M. and C. Kobryn, *Architecting Systems with UML 2.0.* IEEE Software, 2003. **20**(4): p. 57-61.