Fraunhofer Einrichtung
Experimentelles
Software Engineering

The Normal Object Form: Bridging the Gap from Models to Code

Authors:
Christian Bunse
Colin Atkinson

Accepted for publication in
Proceedings of the 2nd International
Conference on the Unified Modeling
Language 1999, UML'99

IESE-Report No. 035.99/E
Version 1.0
June 25, 1999

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Executive Summary

The value of graphical modeling within the analysis and design activities of object-oriented development is predicated on the assumption that the resulting models can be mapped correctly, optimally and efficiently into executable (normally textual) code. In practice, however, because of the large potential mismatch in abstraction levels, the mapping of graphical models into code is often one of the weakest and most error prone links in the chain of development steps. This paper describes a practical approach for addressing this problem based upon the definition of a restricted extension of the UML known as the Normal Object Form (NOF). The basic purpose of the NOF is to provide a set of UML modeling concepts which are "semantically close" to those found in object-oriented programming languages. Highly abstract UML models can then be mapped into corresponding executable code by means of a series of semantically small refinement (intra-UML) and translation (extra-UML) translation steps, rather than in one large (often ad hoc) step. This not only increases the chances of a correct and optimal mapping, but also significantly improves the traceability of UML constructs to and from code constructs, with all the associated advantages for maintenance and reuse.

Table of Contents

1 Introduction

The success of the Unified Modelling Language reflects a growing consensus in the wider software industry that "modeling is a good thing." However, models created in the earlier phases of development (e.g. analysis and design) are of little value unless they can be readily mapped into correct and efficient executable forms, which in today's technology means code in high-level object-oriented programming languages. Any problems in the transformation path from models to code not only have a negative impact on the quality of the delivered software system, but also hinder its future maintenance and/or reuse. Furthermore, they reinforce the widely held suspicion that modeling is "just paperwork" without any serious connection to the "real" business of code generation.

Ironically, the very richness and generality of the UML is something of an "achilles heel" in this regard. This is because power without control is dangerous. Developers attempting to use the UML have such a wide selection of modelling concepts at their disposal, ranging from low-level implementation oriented concepts to very high-level abstract concepts, that they can easily lose their way and end up facing a daunting gap to span in order to translate their models into code. Not surprisingly, the wider the semantic gap to be bridged, the greater the chance of mappings which are inadequate or incorrect.

It is not the UML per se which is really responsible for addressing this problem, but rather the methods which are intended to support it. However, few if any, of the current UML-oriented methods pay much attention to this issue. The books which define the leading methods at best include a chapter discussing "implementation issues", usually in a general and ad hoc way. Similarly, books on computer languages rarely spend more than a chapter discussing how features of the language relate to modeling concepts such as those in the UML. As a result, the mapping of graphical models into code is one of the most neglected links in modern software development processes.

Perhaps one reason for this is the widely held view that case tools have already solved this problem. Widely advertised capabilities such as "round-trip engineering" give the impression that at the press of a button a case tool can translate a rich UML model into a complete, optimal, executable program. However, although the code creation capabilities of modern case tools can be helpful when used appropriately, no tool is yet capable of handling all the nuances and trade-offs involved in creating an optimal and efficient implementation of UML models. The "one-size-fits-all" mapping schemes found in most case tool inevitably end up being incomplete or suboptimal for most situations.

Metamodelling approaches such as 'Design by Translation' [7] or metamodelling tools such as Platinum Technology's 'Paradigm Plus' represent a step forward over simple case tools since they enable the mapping scheme for each metamodel concept to be defined independently by the user. However, this technology still assumes that the user "knows" what mapping scheme to use. Without an underlying theory or methodology for mapping UML models into code, a user can just as easily define an inappropriate mapping in such a tool as they can apply the mapping by hand.

These problems all point to the need for a well-defined and flexible methodology for supporting the translation of UML models into executable code in a way that takes into account prevailing non-functional requirements. This paper describes an attempt to provide such methodological support. After first outlining the principles underling the approach in section 2, the bulk of the paper in section 3, describes the Normal Object Form, a restricted extension of the UML which aims to encapsulate and enhance the implementation-oriented elements of the notation. This is followed in section 4 by a description of an accompanying methodology to support the flexible and optimal implementation of NOF elements based on a modified form of design pattern.

2 Refinement versus Translation

There are a large number of different object-oriented methods to choose from when considering the use of the UML in a software development project, with an equally large number of different processes and modelling approaches. Despite their prima-facie differences, however, they all share the same underlying assumption that high-level "analysis" models will be developed during the early phases of development, and lower level executable code (i.e. the implementation) will result from the later phases. The terms "earlier" and "later" may no longer apply in a strict waterfall sense due to the prevalence of incremental processes, but the basic idea of progress depending on abstract models being transformed to concrete code is more or less universal.

Two basic transformations take place in turning high-level abstract models into concrete executable code - *refinement* and *translation*. Refinement is a relation between two descriptions of the same thing, with one, the *abstraction*, containing less information than the other, the *realization*. In the context of software development, refinement can be viewed as a relation between two descriptions of a software entity, the abstraction or high-level description, and the realization or low-level description closer to implementation. Translation, in contrast, is the description of a given phenomenon in two different ways, but at the same level of abstraction. A classic example from every day life is the translation of a piece of text from one natural language (say English) to another (say German). If done correctly, the information content (i.e. the meaning) of both versions should be identical. In the context of software development, translation results in the description of a given software entity in two different ways (e.g. graphical and textual), but with the same information content.

2.1 Separation of Concerns

Although from a conceptual point of view these two ideas are clearly distinct, they are rarely distinguished in practical software development methods. On the contrary, most methods bundle them together into a series of "shopping list" style implementation guidelines which attempt to describe an "implementation" for each distinct modelling feature on a case-by-case basis. However, this approach leads to various problems -

1. *large semantic gap* - for high level modelling constructs with no direct programming language counterpart, trying to bridge the semantic gap to code in one large jump significantly increases the chances of errors or poor mappings,

2. *undocumented decisions* - even if a reasonable mapping is obtained when performing the mapping in one step, the decisions that it embodies are undocumented and thus unavailable for future maintainers of the system,

3. *loss of reuse opportunities* - although the refinement concepts applied within such a "single step" mapping are often applicable to various languages, bundling them up with language specific translations makes them unavailable for reuse in mappings to other languages,

4. *replication of information* - the previous problem (3) implies that implementation guidelines targeted to different languages often replicate language independent refinement concepts.

A concrete example of the problem in a UML context is shown in Figure 1, which depicts a UML association being mapped into C++. Such an association in the UML conveys a limited amount of information. It indicates only that instances of the two classes may be linked in some way at run-time, and says nothing about the precise nature of the links, nor how they should be implemented in a programming language such as C++. As a result there are usually many ways to implement them, depending on the exact properties required. In the case of a high-level association such as this, some of the major implementation considerations are which class should be the client and which the server, whether the server should be a data member of the client or just a local method variable, whether the server should be embedded with the client or just referenced by it and so on. By making all these decisions in one fell swoop, and bundling them altogether within the resulting code, they remain implicit and undocumented. This not only makes the code hard to check for correctness (i.e., is this the correct implementation of the association giving the prevailing non-functional requirements), but also difficult to understand. This, in turn, causes problems during maintenance when one cannot easily identify why the association was implemented in a particular way and thus what changes are acceptable.

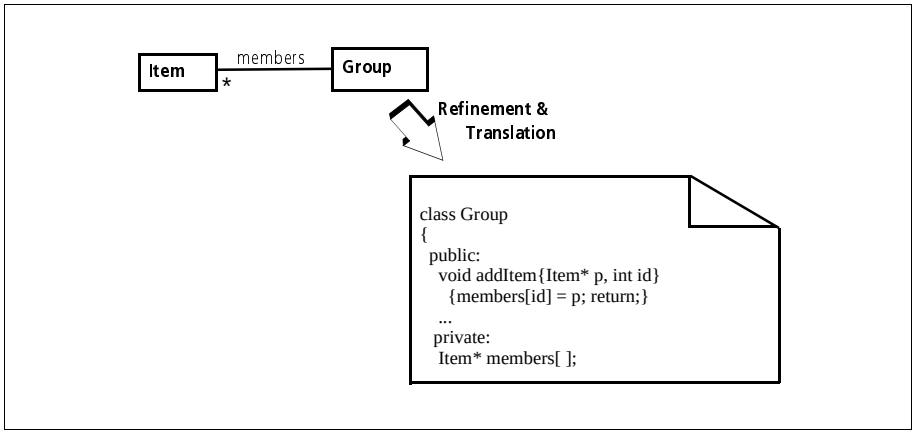

Figure 1: Example 1: Refinement & Translation

Following the time honored principle of "separation of concerns" we believe the only way to seriously address this problem is to cleanly decouple refinement and translation within the software development process. Instead of bundling both transformations together into a single step they should be performed independently. According to this scheme analysis and design models would be developed in the usual way as before, but before translation they would first be refined within the UML to a lower level of detail. Only when models have been obtained at the appropriate level would they be translated directly and straightforwardly into code. Applying this approach to the previous example, as illustrated in Figure 2, we obtain an additional description of the association, still in the UML but at a lower level of detail. The two most important benefits of this approach are that the refinement relationships are clearly visible as explicit UML constructs, and the size of the individual mappings steps (and thus the likelihood of error) is significantly decreased.

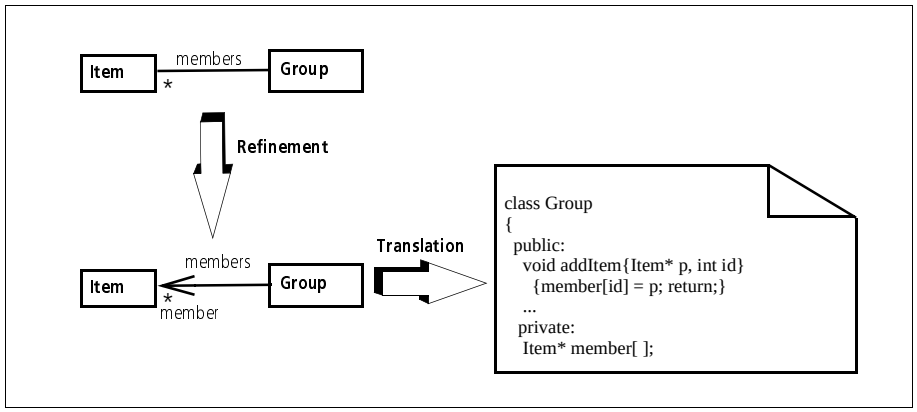

Figure 2: Example 2: Refinement then Translation

2.2 When to Refine and When to Translate?

The idea of separating refinement from translation in the implementation of high-level modelling concepts would seem to offer numerous advantages, but its practical realization begs one major question - to what level should "high-level" models be refined before they are ready for translation, and how can this level be identified. In other words, what exactly is "high -level". The goal of the Normal Object Form (or NOF) is to provide an answer to this question.

As illustrated in Figure 3 (which is a generalization of Figure 2) the purpose of the NOF is to define a set of UML modelling constructs which are "semantically close" to object-oriented programming features, and which can therefore be mapped into elements of a program in a manner that approximates translation (i.e without a significant change in abstraction level). We call this set the Normal Object Form, or NOF, because in a sense it represents a "normal" form, akin to

that used in relational databases, to which UML models must be 'reduced' before translation can begin. On might think of a UML model as being "normalized" by the application of refinement rules to prepare it for translation into code.

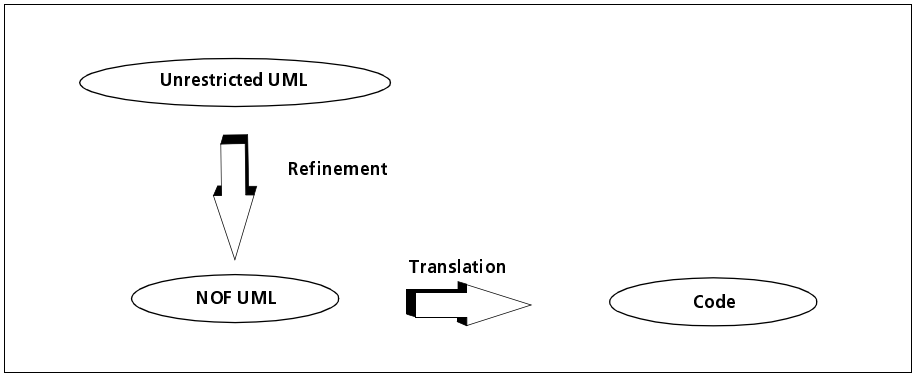

Figure 3: Role of the NOF

Notice that the definition given above does not mention a particular programming language, but instead refers to the "constructs of object-oriented programming". This reveals a second goal of the NOF which is to capture the concepts which are common to the majority of mainstream object-oriented languages, not those specific to a particular language. Only then will the true benefits of separating refinement from translation be available. As experienced programmer are aware, the underlying concepts of object-oriented programming are basically the same whatever language or implementation vehicle is actually used to apply them. For example, the various implementations options discussed above in the implementation of an association apply to most mainstream object-oriented programming languages. In fact, this core set of concepts lies at the heart of the universal applicability and success of object-oriented design patterns [4], [5]. In a sense, the NOF can be thought of as providing a UML embodiment of the common, core features of object-oriented programming.

This generality has a price, however. By positioning the NOF at a level to capture the common concepts of the main object-oriented programming languages (e.g. Java, C++, Eiffel, Smalltalk, Ada..), the mapping of NOF elements to any one particular languages is often no longer strictly translation in the sense defined above. However it is close enough for practical purposes. Hence, the use of the wording "..that approximates translation.." in the definition above. The positioning of the NOF in the abstraction hierarchy represents a trade-off between the need for generality across programming languages, and the need to approximate translation in the mapping to a particular programming language. In the followings section we provide an overview of the main concepts in the NOF.

3 The Normal Object Form

At first sight it might appear that a *subset* of the existing UML modeling concepts would be sufficient to satisfy the goal identified above. Certainly the UML already contains many low-level features which have a very close correspondence to object-oriented language concepts, such as classes, methods, packages etc. However, it turns out that there are also numerous other fundamental object-oriented programming features which are not represented directly within the UML. For example, the various implementation choices identified above in the implementation of an association are not supported very well using the current UML modelling constructs. Therefore, in defining the NOF we also found it necessary to add some additional concepts using the UML's in-built extension mechanisms (i.e. stereotypes, tagged values and constraints). More precisely, the definition of the NOF consists of three distinct elements:

– a subset of the predefined UML modeling features,

– additional modeling features, defined through the UML extension mechanism,

– constraints on the use of (1) and (2).

In a sense therefore, the NOF can be viewed as a restricted extension of the UML.

Another major consideration in the definition of the NOF was how many of the UML's eight distinct diagram types are actually affected? In principle, all of them could be, because they all have a bearing on the properties of the software system, and thus ultimately on the way it should be implemented. In practice, however, it turns out that information in many of the diagrams types is usually "folded into" other types as part of the refinement process. For example, the information provided by use case diagrams, and the sequences diagrams that usually accompany them, ends up in the methods of the classes from which the system will be constructed.

The same is also true for the UMLs' statechart diagrams. These describe characteristics of a system which although important are not usually directly visible in object-oriented programs. The constraints they define are instead indirectly manifest within the instance variables and methods of the class concerned. In general, the information from the UML's dynamic models is essentially "folded" into the static structure models through refinement steps, and cannot be directly "translated" into a programming language.

In practice, we find that only two of the main diagram types have a significant role in describing the "as is" implementation of an object-oriented program -

- Static Structure Diagrams (i.e., Class Diagrams and Object Diagrams)
- Implementation Diagrams (i.e., Component Diagrams and Deployment Diagrams)

Information from the other UML diagrams is generally folded into these as part of the refinement process. Since static structure diagrams alone account for well over half the UML modeling concepts, the NOF still embraces a large fraction of the predefined UML modeling elements.

3.1 Class Diagrams

In this section we describe how the NOF impacts the two major kinds of static structure diagrams - class diagrams and objects diagrams.

Class Diagram Subset

At its core, the NOF contains those elements of UML class diagrams which embody the fundamental elements of object-oriented programs such as classes, attributes, associations and inheritance. Since this list is rather extensive in a paper of this size it is more illuminating to look at the major concepts which are not deemed appropriate for the NOF since they have no directly counterpart in object-oriented programming languages. These include:

- *Specialized Compartments*. These compartments are used to show specialized abstract properties of a class (e.g., responsibilities, business rules, etc.) By definition therefore, such compartments play their major role in the analysis phase to help developers understand the domain, but do not play an important role in implementation. Consequently they are not included in the NOF. The only compartments which are acceptable in the NOF are those for attributes, operations and exceptions.
- *Association Class*. Association classes describe an association that is also a class. Although it is stated [6] that an association class is not the same as a class connecting two other classes, no existing object-oriented language supports any other implementation [5] (i.e., a dictionary class is used). Consequently association classes are not part of the NOF.
- *Class-in-state*. Classes with a state machine may have many states. The class-in-state modeling element describes a state that objects of that class can hold. Due to its close relation to activity diagrams, it is another way to accomplish the same goal as dynamic classification. Therefore it not necessary as a part of the NOF.

– *Dependency.* All dependencies which describe historical connections between elements (e.g., <<trace>>) do not influence the implementation of a system and are therefore not part of the NOF.
– *Derived Elements*. These are not part of the NOF because they are used for the purpose of clarity and do not provide additional semantic information.
– *Metaclass/object and Powertype*. Metaclasses are classes whose instances are also classes, whereas powertypes are metaclasses whose instances are subclasses of a given class. They are typically used to construct metamodels and thus, can be removed from the NOF.
– *N-Ary Associations*. N-ary associations are associations between three or more classes. However, just as for association classes no currently existing object-oriented language provides direct support for such associations; they have to be "simulated" using multiple binary associations.
– *Qualifiers*. These are used to partition a set of objects connected with an object via an association. Qualifiers are not part of the NOF for two reasons. First, due to their definition as attributes of an association (see also association class). Second, they are clearly analysis elements, which model an important semantic situation, but do not influence the general strategy for implementing an association.

Additional Class Diagram Elements

Although the UML is a powerful tool for describing object-oriented software systems it is not possible to describe all properties of programs entirely in the UML subset within the NOF. Certain extensions (i.e., new elements) are needed. The NOF includes legal extensions to the UML defined using the in-built extension mechanism. Most of the UML extensions in the NOF occur in connection with associations. This is because the fundamental implementation variations for associations are not fully supported in the present version of the UML. Although associations can vary in many ways at the analysis and design level, such as in their arity (e.g. binary, ternary, etc.) and their multiplicity (e.g. one-to-one, one-to-many, many-to-many), at the implementation level there are far fewer variations. All inter-object relationships are essentially implemented by the same basic mechanism: one object holding a pointer to, or the value of, another object. Even the implementation of associations by 'Relation Tables' makes use of these mechanisms, by implementing the table as a class in its own rights which routes the communication.

Following ION [1], we call this basic relationship between classes "*clientship*". Clientship is an asymmetric relationship; the client needs to be aware of the identity of the server class, but the server requires no knowledge of the client class. All clientship relationships are therefore represented with a UML navigation arrow indicating the direction of the client/server relationship. As with all

program-level relationships, clientship implies a compilation (and thus static) dependency. In total there are four different, orthogonal properties of clientship relationships, each with two possible values. Each possible value for each property has a corresponding UML association stereotype:

1 *Attached vs. Detached:* One of the most important characteristics of a clientship relationship is whether the client holds a reference to the server or whether it holds the actual state (i.e., the value) of the server. When the client holds a reference to the server the clientship is said to be *detached*, whereas when the client actually holds the state of the server the clientship is said to be *attached*.

2. *Permanent vs. Transient:* Another important characteristic of a clientship relationship is how long the class has visibility of a particular instance of the server class. If the client holds a reference to, or the value of, the server in its main data structure, the clientship is said to be *permanent*. If, on the other hand, the client has visibility of a server object only for the duration of a single method, the clientship is said to be *transient*.

3. *Proper vs. Intimate:* Normally, a client class only has access to the "official", publicly visible methods of the server. This is termed *proper* clientship. Most object-oriented languages also allow client classes to be given privileged access to the server. This is termed *intimate* clientship.

4. *Direct vs. Indirect:* The final property of a clientship relationship is whether the client holds visibility of the server, or whether the client relies on a second server for visibility of the first. The first situation is known as *direct* clientship, and the second *indirect* clientship.

Individual clientship relationship between two objects, or their corresponding classes, must make a choice between all four binary attributes. However, showing all four stereotypes, in full, on a clientship arrow would unduly clutter a NOF class diagram. Therefore, as well as defining default properties (detached, permanent, proper, direct) the NOF defines stereotypes corresponding to meaningful permutations of the four orthogonal characteristics (see Table 1).

In general there are 16 different possible combinations of the four binary clientship properties, but some of them may not necessarily fit well together from a programming point of view. A typical example is a clientship relationship characterized by 'Attached, Permanent, Proper, Indirect'. Such a combination defines a relationship where the client contains the server and all intervening objects which, in the worst case, may lead to really large objects.

As mentioned previously, the fundamental elements of object-orientation are naturally present in the NOF, such as classes, objects, attributes, links, associations and inheritance. However, the NOF has to supports these concepts at the

level of detail in which they appear in an object-oriented program. Thus, for example, when a class appears in a NOF diagram, its precise NOF implementation stereotype must be defined - that is, whether it is an abstract class (<<abstract>>), a persistent class (<<persistent>>), a template class (<<template>>), or a utility class (<<utility>>). If a class is not marked as belonging to any of these categories in a NOF diagram, this indicates that a decision has been made to implement it as a normal class.

Combination of Characteristics	stereotype	Description
Attached, Permanent, Proper, Direct	<<embedded>>	Client holds value of public parts of the server in main data-structure directly.
Attached, Permanent, Intimitate, Direct	<<private>>	Client holds value of public/private parts of the server in main data-structure directly.
Attached, Transient, Proper, Direct	<<public local>>	Client holds value of public parts of the server for method execution directly.
Attached, Transient, Intimitate, Direct	<<local>>	Client holds value of public/private parts of the server for method execution directly.
Detached, Permanent, Proper, Direct	<<standard>>	Client holds direct reference to public parts of the server in main data-structure.
Detached, Permanent, Proper, Intimitate	<<Dictionary>>	Client holds indirect reference to public parts of the server in main data-structure.
Detached, Permanent, Intimitate, Direct	<<Friendship>>	Client holds direct reference to public/private parts of the server in main data-structure.
Detached, Transient, Proper, Direct	<<Parametric>>	Client holds direct reference to public parts of the server for method execution.
Detached, Transient, Intimitate, Indirect	<<Parametric Dictionary>>	Client holds indirect reference to public parts of the server for method execution.
Detached, Transient, Intimitate, Direct	<<Parametric Friend>>	Client holds direct reference to public/private parts of the server for method execution.

Table 1: Possible Clientship Relationships

A good example of how NOF features are tailored towards implementation concepts is provided by the new stereotypes for packages. The package concept in the UML is a highly general concept intended to model any kind of grouping of elements whether logical or physical. Packages can of course be used to model modules in a program (in the case of Ada and Java these are even called packages).

However, in contrast with the logical packages of UML, the implementation packages of languages like Ada and others are typically divided into two parts to support the principles of information hiding. The specification part usually contains the elements of the packages which are intended to be exported to other

parts of the program, while the body contains those parts which should be hidden. This concept is supported directly in the NOF in the form of <<Specification>> and <<Body>> stereotypes as illustrated in Figure 4. The same stereotypes can also be applied to classes, where the specification relates to the interface and the body to the inner details (e.g., method implementations).

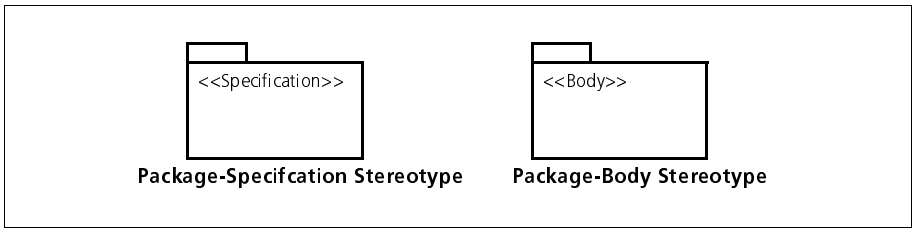

Figure 4: Package Stereotypes in UML

Class Diagram Constraints

Most of the UML modeling elements can be used at different levels of abstraction. For example in the analysis phase an operation can be specified by a simple name whereas during detailed design it can be specified in more detail (e.g., types, parameters, etc.). In general, the UML allows the modeler to decide when and where certain pieces of information are used. While this flexibility is valuable during high-level modeling phases, it becomes a problem in the implementation phases when the "as is" implementation needs to be described.

Class Diagram Constraints
1. All attributes of a class must have visibility markings. In other words, it must be clear whether they are to be implemented as public, private or protected members (in the case of C++).
2. A method must have a visibility marking, a list of parameters (if existing), and a return type
3. A parameter is specified in the following form: name:type=default-value
4. Each class must have at least a constructor and destructor method.
5. The methods of a class have to be grouped by using one of the following stereotypes <<constructor>>, <<destructor>>, <<update>>, etc.
6. The parameter of a template class must be bound to an actual value to be meaningful.
7. A clientship relationship has to be augmented with multiplicity markings.
8. A clientship relationship has to be augmented with roles.

Table 2: Selection of Constraints

From an individual diagram it not possible to determine whether the absence of specific markings is due to the fact that they have simply been omitted or

because the corresponding decisions remain to be made. A general goal of the NOF is to rule out such ambiguities by providing constraints which clearly specify the level of detail to be represented. Due to the size of the UML and the limited space within this paper we cannot present the full set of constraints. However, Table 2 presents a short selection to give an overview of their nature.

3.2 Object Diagrams

The basic purpose of an object-oriented program is to create a set of objects which, at run-time will interact in such a way as to satisfy the needs of the users. The UML provides the basic mechanisms needed to describe this aspect of a system's implementation in the form of an object diagram, but not entirely in an appropriate form. This is because the way in which object diagrams are used in analysis is not appropriate for describing the "as is" implementation of a program. In analysis, object diagrams are generally used to depict a *typical* set of named instances and a *typical* set of links in order to provide an illustration of the kind of object structures that are meaningful for the application under development. There is no real sense of precisely when the object structure exists, and from whose perspective it is defined.

In a running object-oriented programming, however, the set of objects in existence at any one point in time is constantly changing. Moreover, the absolute names of objects, if they have any meaning, are known only to the run-time system. Individual objects only know about (and are able to communicate with) other object through the names of their instance variables. The predefined UML object diagram features also neglect another important aspect of a running system known as the *creation tree*. Apart from the outermost object (or main program) all other objects in a running program have to be created by some other object, and failing to ensure this takes places properly is one of the major sources of errors in object-oriented programs.

To rectify these problem, the NOF makes two main enhancements to the basic object diagram concepts in the UML -

– the first is to identify two distinct kinds of object diagrams - *snapshot* object diagrams which describe a group of objects and their links at a particular instance in time, and *history* object diagrams, which describe the same information accumulated over a period of time.
– the second is to make every object diagram relative to one specific object in the diagram known as the *root.*

These two enhancements are closely related, because the information contained in each kind of object diagram is defined relative to the root object. In the case of a snapshot diagram, the instance of time represented by the diagram corre-

sponds to a particular state of the root object, and in the case of a history diagram, the presented information is accumulated with respect to the root object. More specifically, a history object diagram shows an accumulation of all objects which the root object is linked to during its lifetime.

The NOF defines various stereotypes and constraints to support and enforce this usage of object diagrams as illustrated in Figure 5. This examples illustrates the set of links which are meaningful for a user of a library who is currently borrowing a book. A user who has a no books on loan would have a different set of links.

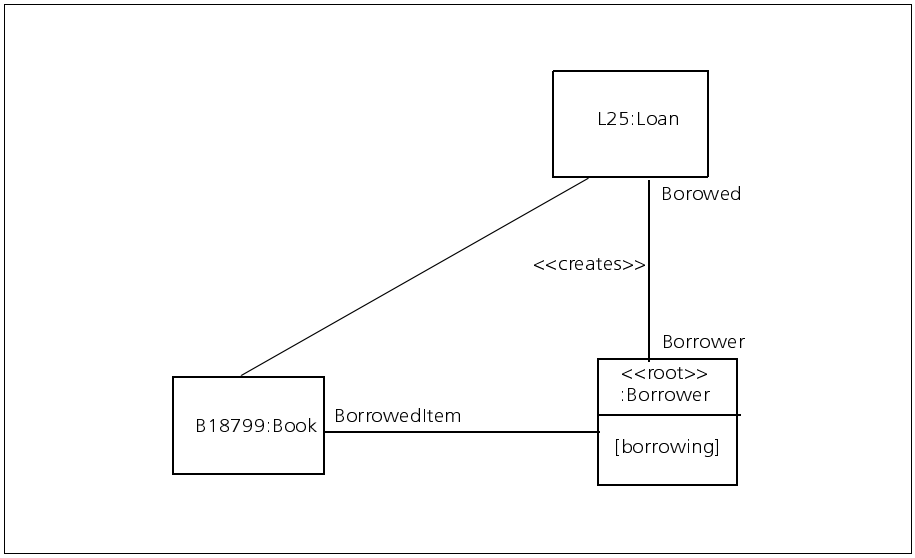

Figure 5: Snapshot Object Diagram

Figure 5 illustrates most of the important object diagram stereotypes and extensions defined in the NOF. First, it shows the use of the <<*root*>> stereotype to show that the anonymous object *:Borrower* is the root of this particular object diagram. Second, it illustrates the use of the stereotype <<*creates*>> to indicate that this object is responsible for generating the *Loan* object *L25*. Although this object has a name, the *Borrower* does not know this name. It is only able to access the *Loan* object through the role name *Borrowed*, which corresponds to the name of the instance variable in the class defining borrowers. Thirdly, the presence of the state definition *[borrowing]* in the body of the object indicates that this diagram is a snapshot diagram depicting the links which *Borrower* has when it is in the *borrowing* state. Although the UML does not intend state to be represented within object diagrams, the approach in the NOF is consistent with the notation for object state defined for activity diagrams.

4 SORT

In order to be of practical value the NOF needs appropriate methodological support. This is the goal of *SORT* (**S**ystematic **O**bject-Oriented **R**efinement and **T**ranslation). *SORT* provides a practical technique for leveraging the NOF, and the concept of refinement/translation separation, by packaging useful refinements and translations [2].

In view of the success of the pattern cataloguing approaches pioneered by Gamma [4] and Buschmann [3], SORT refinement and translation guidelines are packaged in a similar style. However, there is a subtle differences between the patterns defined in *SORT* and those of Gamma and Buschmann. Whereas the latter essentially capture good (i.e. useful) object-oriented structures/behaviors, *SORT* patterns capture good (i.e. useful) mappings between object-oriented structures/behaviors.

Two forms of patterns are recognized in SORT: *refinement patterns,* which describe "good" refinements within the UML for reaching structures at the implementation level specified by the NOF, and *translation patterns*, which describe the "good" mapping of UML-NOF models to a specific object-oriented programming language (e.g., C++). The latter are similar to "idioms" [5] in that they are language specific, however, as mentioned above they represent more of a mapping guideline than a useful programming practice. An example of each of the two pattern forms can be found in Figure 6.

Of course, there is rarely a single pattern which provides the best mapping (refinement or translation) of a given structure under all circumstances. Generally, there are several potential mappings, and the one which is most appropriate in a particular context depends on the associated non-functional requirements (e.g. performance needs, space limitations, reliability etc.). Therefore patterns have to provide a context description which allows a developer to choose the one most suitable for his/her particular needs. Providing such information allows *SORT* to offer a level of context sensitivity which is impossible in automated mapping tools, while at the same time still being reasonably systematic. Developers are told precisely what to refine or translate, how to perform these refinements/translations, and when to perform the relevant activity. At the same time, however, they can tailor the particular refinements and translations in different ways that depend on the relevant context factors. More detailed information on SORT and its patterns can be found in [2].

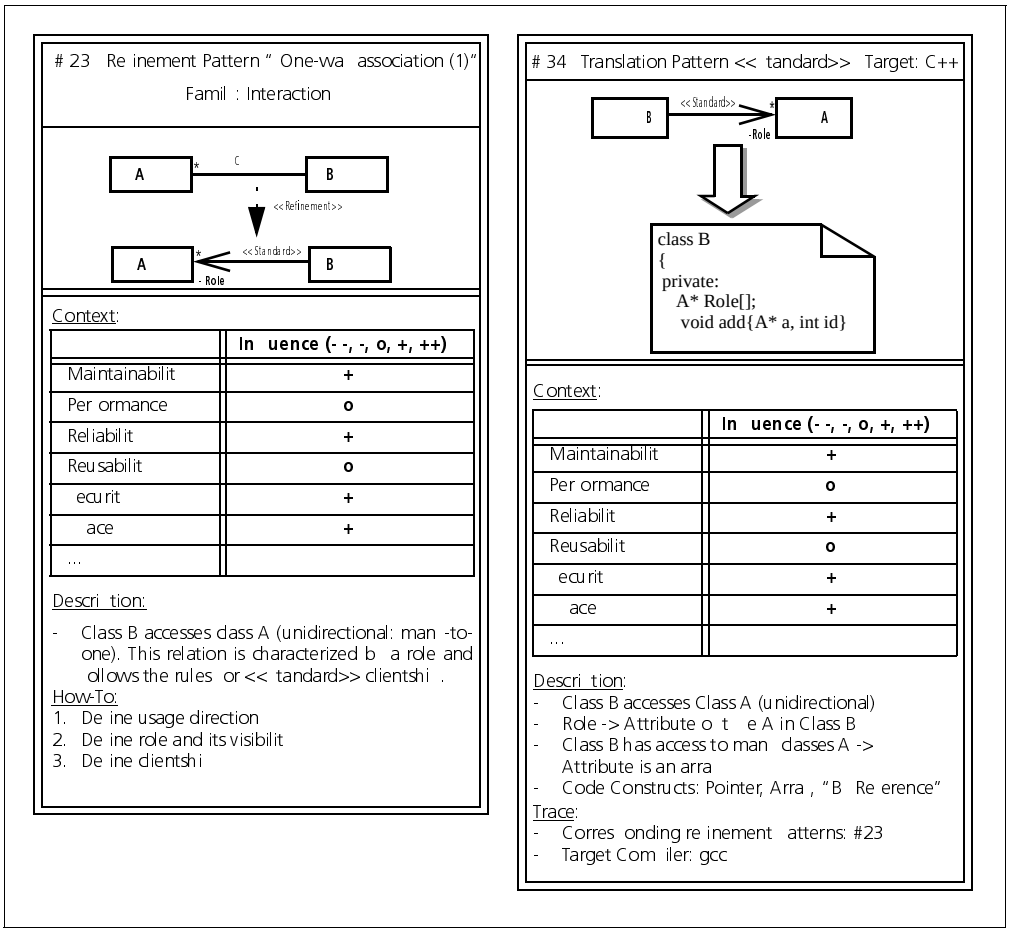

Figure 6: Pattern Examples

5 Conclusion

The phenomenal interest in the Unified Modelling Language provides a real chance for the software engineering profession to increase the amount of modelling work which is performed in the competitive software development industry, and through this to increase quality standards. However, this chance will be lost if developers are given no effective and practical link between UMLs diagrams and the executable implementations of their systems.

This paper has described a strategy for addressing this need which is based on three time-honored and fundamental engineering principles. The first is the principle of "separation of concerns". The basic tenet of the approach is to cleanly distinguish and separate two independent issues in transforming high level modes into executable code - namely, refinement and translation. The second is the principle of "exploiting commonality". The approach is explicitly aimed at exploiting the common, core concepts of object-oriented languages, and through this to define a set of general refinement patterns which are independent of language idiosyncrasies. The third is the principle of "divide and conquer". By clearly separating and capturing individual refinement and translation steps, the approach divides a single, "semantically large" mapping step into series of intellectually smaller steps. This not only serves to document the steps, but significantly improves the likelihood that the overall mapping will be correct.

These ideas are embodied within a restricted extension of the UML known as the Normal Object Form, and are supported by a methodology for their practical application known as SORT. One of the main advantages of the SORT approach is that it "doesn't care" where the original UML diagrams come from. In other words, the SORT approach is independent of, and usable with, any of the mainstream UML development methodologies. The ultimate goal of this work is to define sufficient refinement patterns to enable any UML compliant diagram to be normalized into NOF form ready for translation into code.

Another important property of this approach is its "tool friendliness". As discussed in the introduction, by trying to support the single-step implementation of all UML modelling concepts, no matter what their level of abstraction, case tools may actually be doing developers a disservice. This is because they typically have to apply a "one-size-fits-all" mapping strategy which fails to document the rationale for the mapping, and often fails to provide the best mapping for the circumstances in hand. The SORT approach, based on the NOF, promises to improve this situation by enabling tools to concentrate what they do best, namely context independent translation between different representations of

the same concept. The subtle, context sensitive aspects of the implementation process can then be left to humans.

In the long term, formally defined languages and notations may become available which will enable the translation of graphical models to executable code to be performed with formal rigor and 100% accuracy. Until that time however, we believe the approach outlined in this paper, which is currently under development at the Fraunhofer Institute for Experimental Software Engineering, provides one of the most practical ways of introducing some rigor into the implementation phase of object-oriented development, and in providing a reasonable degree of traceability and verifiability between graphical models and the object-oriented programs which implement them.

References

[1] Colin Atkinson and Michel Izygon. *Ion a notation for the graphical depiction of object-oriented programs.* Technical report, University of Houston-Clear Lake, 1995. available via WWW at: http://ricis.cl.uh.edu/atkinson/ion.

[2] Christian Bunse and Colin Atkinson. Improving quality in object-oriented software: Systematic refinement and translation of models to code. Submitted to the *Fourteenth Annual Conference on Object-oriented Programming Systems, Languages & Applications (OOPSLA)*, 1999.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley and Sons, 1996.

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

[5] Richard C. Lee and William M. Tepfenhart. *UML and C++. A Practical Guide To Object-Oriented Development*. Prentice Hall, 1997.

[6] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.

[7] Sally Shlaer and Stephen J. Mellor. The shlaer-mellor method. Pages on the WWW which can be found at: http://www.projtech.com/smmethod/smmethod.html, 1998.

References

Document Information

Title:	The Normal Object Form: Bridging the Gap from Models to Code
Date:	June 25, 1999
Report:	IESE-035.99/E
Status:	Final
Distribution:	Public