Fraunhofer Einrichtung
Experimentelles
Software Engineering

Improving Quality in Object-Oriented Software: Systematic Refinement and Translation of Models to Code

Authors:
Christian Bunse
Colin Atkinson

Accepted for publication in
Proceedings of the 12th International
Conference on Software & Systems
Engineering and their Application,
ICSSEA'99

IESE-Report No. 036.99/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 reliable attainment of quality requirements is still a major weakness of the object-oriented development paradigm, with a significant proportion of object-oriented systems either failing to work correctly, or failing to do so in a way that meets non-functional requirements. One of the main reasons for this problem is the discrepancy between the various object-oriented languages/notations typically used during the course of an object-oriented project and the lack of well-defined mappings between them. This paper describes a practical strategy for addressing this problem, known as SORT, which is based on two fundamental tenets: distinguishing and strictly separating refinement from translation, and the definition of a common, core set of object-oriented implementation concepts to minimize the gap between object-oriented models and programs. When used in a systematic way, SORT can not only improve the quality of an object-oriented system, but can also significantly increase confidence that the desired quality levels have been attained. The approach is applicable to most major modeling and programming languages although, in this paper, we concentrate on the UML and C++.

Keywords: UML, Refinement, Translation, Patterns, Object-Oriented Development

Table of Contents

1 Introduction

The widespread adoption of object technology over the past decade has brought many advantages to the software industry, but increased quality is not normally viewed as one of them. A high proportion of object-oriented systems still either fail to work correctly, or do so in a way that falls short of certain non-functional requirements. Moreover, object-oriented methods still lag a long way behind leading structured methods, such as Cleanroom [9,2], when it comes to systematic quality attainment. This is reflected in the fact that in domains where reliability is a primary concern the object-oriented approach is still rarely used in its full generality[20].

Although requirements specification shortcomings account for many quality problems, a large proportion of defects are still introduced in the mapping of the requirements specification to executable code. It is tempting, therefore, to lay much of the blame for low quality in object-oriented systems on poor verification, since this is the activity which is supposed to assure the integrity of the mapping. However, in the case of object-oriented development this would be unfair. Verification techniques, such as inspection, can only work effectively when there is something to verify, but unfortunately this is rarely the case in real-world projects. A typical object-oriented development project employs several different representations of the system under development, written in various different languages and notations, and the mapping between them is rarely if ever well-defined. In fact, in many cases the semantics of the representations are not even well-defined. It is not so much verification per se that is the problem, therefore, but the lack of well defined development mappings to verify.

The recent advent of the UML as a standard modeling notation has had both a positive and negative impact on this problem. To the extent that it has obviated the use of different modeling notations in analysis and design it has been helpful. However, the increased number of modeling concepts at different levels of abstraction has also complicated the task of correctly and optimally mapping graphical object-oriented models to other object-oriented representations such as code. To cover both analysis and design, the UML is forced to include significantly more features than notations focused more specifically on either analysis or design. Some of these features are at a very high, abstract level, while others are at a very-low, concrete level similar to those found in object-oriented programming languages. In addition, the shear success of the UML means that many more projects will be faced with the problem of mapping graphical models into textual source code.

An obvious way of trying to help developers achieve correct and optimal mappings of UML-style models to other object-oriented representations is to define feature-by-feature mapping guidelines for various combinations of non-functional requirements. However, not only would such a "Mapping Handbook" be required for each target representation (e.g. programming language), but there would be significant replication of material in these "handbooks" because of the commonality of underlying concepts in mainstream object-oriented languages. Using such a brute force approach would miss many opportunities to exploit this commonality, with all the attendant disadvantages of lower portability, higher training costs and poorer maintainability etc.

CASE tools and meta-modeling approaches such as Shlaer/Mellor [18] help to a certain extent by automatically performing mappings of models to code based on in-built or user-defined mapping strategies. A commonly used phrase to describe the automated creation of textual code from graphical models, and back again, is "round-trip engineering." However, this "one size fits all" approach of CASE tools often leads to suboptimal mappings, or mappings that need to be "finished" manually. Meta-modeling tools have the disadvantages that the user still has to specify the required mappings, and thus in all likelihood will have to turn to the appropriate "Mapping Manual" in any case.

In this paper we present a practical approach to this problem, known as SORT, which seeks to identify and exploit the commonalities between object-oriented notations and languages in order to minimize inter-representation mappings (i.e. UML to code) and maximize intra-representation mappings (i.e. UML-to-UML). This not only increases the likely integrity of the overall mappings from requirements to code, but also significantly increases portability between different implementations.

The approach is based on two main principles. The first, described in the following section, is the principle of distinguishing and strictly separating refinement from translation, and the second, described in section 3, is the idea of identifying a core set of object-oriented features common to all important object-orientation representations languages. Practical packaging of the distinct translation and refinement guidelines is achieved in a style similar to pattern catalogues [5, 12]. Section 4 describes the overall method built around these principles and provides examples of inter-UML refinement patterns, and UML-C++ translation patterns. Finally, section 5 concludes.

2 Refinement Versus Translation

A modern software design process takes a high-level description of system functionality and creates a corresponding "implementation" describing the details of how the system actually works. Moreover, this description must be in a high level programming language that can be understood by a compiler. Thus, there are essentially two major transformations which take place in the creation of an "implementation" from a requirements specification: *refinement* and *translation*.

Refinement is the description of a given phenomenon at a lower level of detail. It is therefore a relation between two descriptions (e.g., specifications, designs, or code) of the same thing, but one, the *abstraction*, contains less information than the other, the *realization*. Refining an abstraction does not invalidate it, but merely complements it with a more concrete description[8]. In theory a refinement takes one arrangement of pieces and rearranges or expands it into another, often larger arrangement. This results in a tree (see Figure 1) of models/constructs connected via refinement relationships.

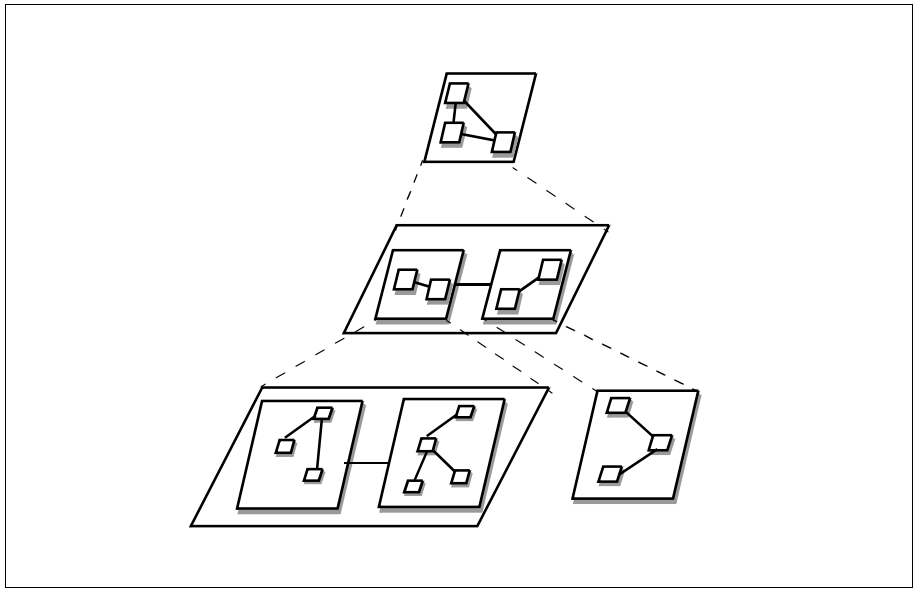


Figure 1: Refinement Tree [8]

Translation is the description of a given phenomenon in two different ways, but at the same level of abstraction. In contrast to refinement, translation describes a relation between two descriptions at the same level of detail. Both descriptions are of the same thing, and contain precisely the same amount of information,

but they use different languages. Any translation, whether formal or informal, has a set of possible inputs (the domain) and a set of possible outputs (the range). In the context of an object-oriented system implementation, the domain is the set of possible models that are created during the modeling/refinement process, and the range is the set of possible programs that could be created in the chosen language. As mentioned above, in the absence of formally defined mappings, the best that can be provided is a set of guidelines and heuristics for matching elements of a particular model to elements of a program.

As an analogy of the difference between refinement and translation, consider the following English sentence; "I spent the day at work.". This can be refined - "I travelled to work on the train, wrote an article and travelled home on the bus," or it can be translated into say German "Ich habe den ganzen Tag gearbeitet". Refinement and Translation can also be combined in one step - "Ich bin mit dem Zug zur Arbeit gefahren, habe den ganzen Tag an einem Artikel gearbeitet, und bin dann mit dem Bus nach Hause gefahren."

The essence, the question addressed by this paper is how best to get from the original form (which corresponds to a specification) to the final form (which corresponds to a software implementation). Unfortunately the majority of methods in practical use today, especially those of the object-oriented variety, handle this problem poorly. Most methods do not even explicitly recognize the difference between the two concepts, and when they do, they rarely provide methodological support for separating them. As a result, software engineers often find themselves trying to transform high-level analysis concepts directly into the low-level constructs of a programming language. In the natural language analogy given above, this corresponds to transforming the high level English sentence directly into the low-level German form.

Why is this a problem? Basically, it breaks two of the most fundamental and time-honoured principles for solving engineering problems -

Divide and Conquer - the reduction of a problem into a sequence of several smaller problems, each easier to understand and solve than the whole.

Separation of Concerns - the decoupling of unrelated aspects of a problem to enable each to be tackled individually using optimal techniques.

Failing to apply these principles by trying to perform complex refinement and translation tasks in one combined step effectively increases the size of the "intellectual gap" to be spanned in a transformation step. Not only does this increase the likelihood of errors, and thus quality problems, but it also leads to problems during maintenance because the rationale for the transformation is more difficult to understand.

An example of this problem in a typical object-oriented software development context is shown in Figure 2. At a high-level of abstraction an association between two classes can be represented as a simple line in a UML diagram. However, if this has to be mapped directly to code not only the notation has to be changed, but also many implicit decisions (i.e., refinements) have to be made. The example shows an implementation in which the class 'Group' maintains a set of items by an array of pointers. This simple example already shows the root of the problem. The implicit mapping decisions are documented solely by the code constructs themselves, making the code not only 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 leads to problems during maintenance when one cannot easily identify why the association was implemented in a particular way and thus what changes are acceptable.

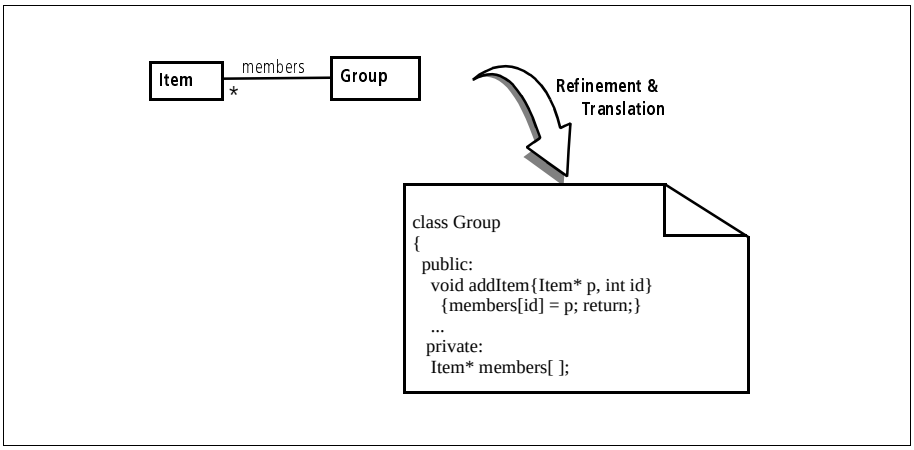


Figure 2: Example 1: Refinement & Translation

The only way to really tackle this problem is to cleanly decouple refinement and translation within the software development process. Instead of performing both activities within one step they should be performed sequentially. Analysis and design models are developed in the same way as before, but these should first be refined within the UML to a level of detail needed for implementation, with the associated refinement relationships explicitly documented. Only when models at the appropriate level of detail have been obtained, they can be translated directly and simply into code.

Applying this approach to the previous example, as illustrated in Figure 3, we obtain an additional description of the association, still in the UML but at a lower level of detail. The most important benefit of this approach is that the refinement relationships are clearly visible as explicit UML constructs.

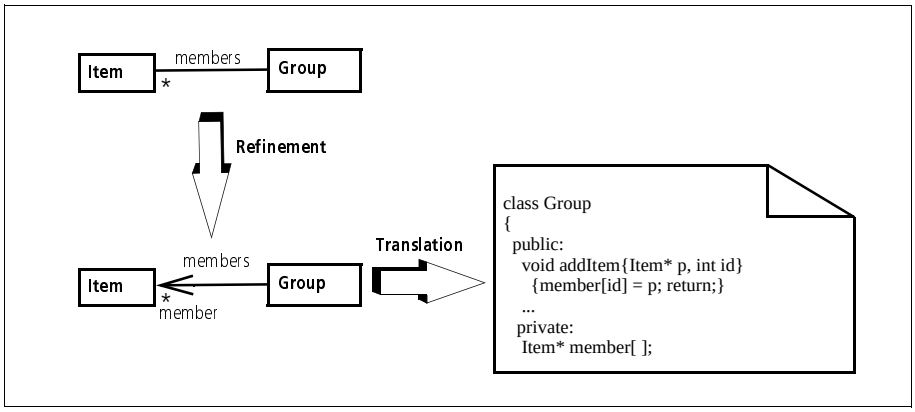


Figure 3: Example 2: Refinement & Translation

3 Normal Object Form

Explicitly separating refinement and translation in the way explained seems like a reasonable idea, but it begs one major question - to what level of detail should the refinement proceed until translation can start? Answering this question is crucial for refinement and translation to be effectively separated in practice. We address this problem by applying another time-honoured engineering principle

exploitation of commonality - identifying identical and similar aspects of a problem domain to avoid solving the same problem twice.

The basic idea is to define a set of UML modeling concepts which correspond to the core constructs of object-oriented programming so that they can 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.

The word "set" was chosen carefully in the preceding paragraph. At first sight it might appear that a *subset* of the existing UML modelling concepts would best satisfy the goal identified above, but on closer analysis this turns out not to be so. The UML certainly contains many low-level features which have a very close correspondence to core object-oriented language concepts, such as classes, methods, packages etc. However, there are also numerous fundamental object-oriented programming features which are not represent directly within the UML. Therefore, in defining the NOF we also found it necessary to add additional concepts to those predefined in the UML, using UML's in built extension mechanisms (i.e. stereotypes, tagged values and constraints).

More precisely, the definition of the NOF consists of three distinct parts:

1 a subset of the predefined UML modeling features

2 additional modelling features, defined through the UML extension mechanism

3 constraints on the use of (1) and (2)

Strictly speaking this makes the NOF a restricted extension of the UML.

Another important question in the definition of the NOF is which of the UML diagram types should be affected. In principle it would be possible to define subsets and extensions of all the diagram types in the UML, but in practice this is unnecessary. This is because by definition the goal of the NOF is to support the "as is" description of the final implementation. In other words, by the time a system is described in the NOF, all language independent refinement decisions should have been made. The NOF therefore only needs to support those concepts which are directly manifest in typical object-oriented programs. However, some UML diagrams, such as use case diagrams, explicitly focus on the description of concepts (use cases) which are by definition far away from detailed implementation considerations, and others, such as interaction diagrams and state diagrams, describe characteristics of a system which are not directly manifest in object oriented programs. For example, although the states that make up a statechart model of a class play a major role in determining the implementation of the class, they are not directly visible in its source code, but instead are indirectly manifest through its instance variables. In other words, information from the behavioural models of the UML is essentially imparted into a system through refinement steps, and cannot be directly "translated" in features of a typical object-oriented programming language.

For these reasons the NOF is actually only embodied within three of the eight diagram types defined in the UML

- static structure diagrams
- implementation diagrams
  - component diagrams
  - deployment diagrams

since these collectively embody the "as is" implementation of a system.

3.1 Example NOF Elements

It is not possible in a paper of this size to provide a complete and detailed description of the NOF and its elements. However, in this section we provide a brief overview of some of its major elements, by addressing in turn the three separate parts of the definition.

3.1.1 UML subset.

At its core, the NOF contains those elements of the UML which embody the fundamental elements of object-oriented systems such as classes, attributes, links, associations and inheritance. Rather than list all the all the basic elements which are included, it is actually more interesting to identify some of the major con-

cepts which are not deemed appropriate for the NOF. Some of the features of static structure diagrams which do not correspond directly to programming level concepts 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 [17] that an association class is not the same as a class connecting two other classes, no existing object-oriented language supports any other implementation [15] (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 associations.
- *Qualifier*. 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.

3.1.2 Additional 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. One simple extension is that of a *thread*. Active classes/objects are not part of the NOF, since they characterize the abstract behaviour of objects in a concurrent environment. However, threads, which provide one common way of realizing active classes, are an important implementation concept (e.g., Java). Thus, we define the stereotype <<thread>> to directly depict threads.

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).

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

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, 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>>), apersistent 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.

A good example of how NOF features are tailored towards implementation concepts is provided by the 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.

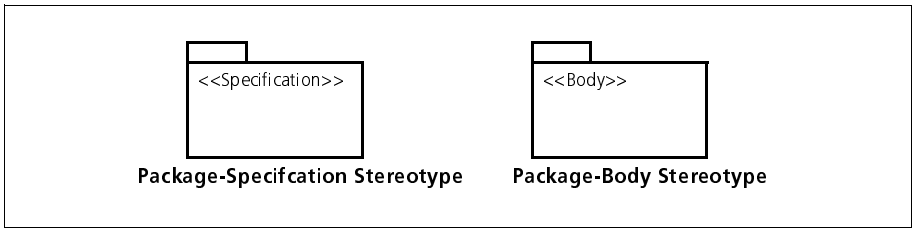


Figure 4: Package Stereotypes in UML

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).

3.1.3 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 modeller to decide when and where certain pieces of information should be displayed. While this flexibility is fine during high-level modeling phases, it becomes a problems in the later implementation phases when the "as is" implementation needs to be described.This is because from an individual diagram is not possible to determine whether the absence of specific markings is because 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.

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

3.1.4 Example

As an example of the application of NOF consider again the example from Section 2. Figure 3. shows how it is possible to provide a more detailed UML model of the association before it is translated into equivalent C++. Using the NOF stereotypes introduced above it is possible to define the precise properties of the chosen implementation, as shown in Figure 5. The stereotype <<Standard>>, indicates that the implementation employs a clientship relationship which is:

- *Detached,* because Group uses a pointer to access the "Item" objects.

- *Permanent*, because this pointer is data member, and thus exists for the lifetime of "Group" object and not only for the duration of a single methods execution.
- *Proper*, because Group only uses the official public interface of "Item".
- *Direct*, because there are no intermediate objects.

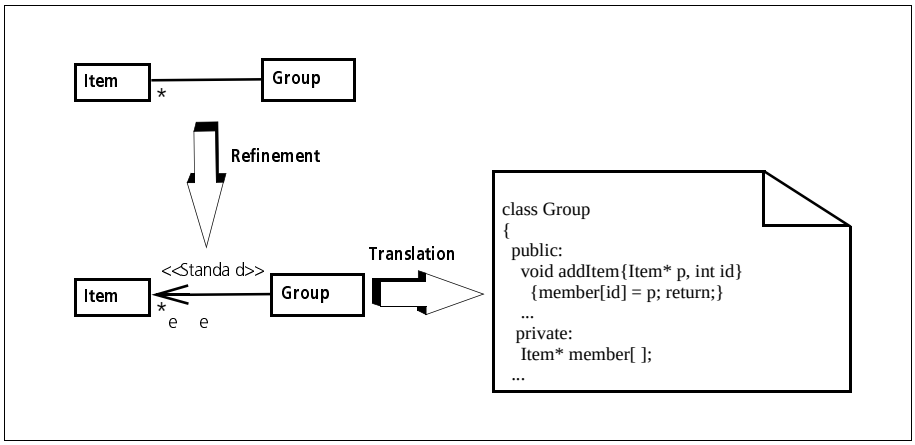


Figure 5: Refinement & Translation by using the NOF

4 The *SORT* Approach

In order to be of practical value, the ideas introduced in the previous sections need some form of 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.

In view of the success of the pattern cataloguing approaches pioneered by Gamma [12] and Buschmann [5], we document our refinement and translation guidelines in a similar style. However, there is a subtle differences between the patterns defined in *SORT* and those of Gamma and Buschmann in that the latter essentially capture good (i.e. useful) object-oriented structures/behaviours, whereas *SORT* patterns capture good (i.e. useful) mappings between object-oriented structures/behaviours.

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 OO 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.

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 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.

4.1 Refinement Patterns

Refinement patterns define sound refinement mappings from higher-level (i.e. non-NOF) UML model elements to UML-NOF elements. In general a refinement pattern defines how to remove higher-level elements (static structure diagrams), how to use NOF specific elements in NOF diagrams (static structure diagrams), and how to enforce the NOF modeling constraints (all diagrams).

To keep track (traceability) of refinements, the refinement relationships between elements must be made explicit by using the UML stereotype <<refinement>>. This enables developers to traverse the refinement tree in order to identify those places where further work is needed. However, as mentioned above, simply defining the mapping is not sufficient. If there is more than one pattern for a specific construct it must be possible to choose the pattern which is most suitable for a given context. An essential part of a refinement pattern is therefore a list of the primary quality factors which the refinements influence (e.g., performance) and the level of this influence. For example, a refinement that implements a model element in a way that minimizes space may well have a detrimental influence on the final system's performance, whereas a refinement that maximizes performance may have a detrimental influence on space usage.

Providing quantitative measures, on the influence a pattern has on quality factors, can be potentially misleading. On the one hand they allow patterns to be chosen on an objective basis, but on the other hand such measures are difficult, if not impossible, to collect and may rely heavily on the definition of the underlying metric. Therefore *SORT* adopts a qualitative scale which provides a subjective influence estimate represented by one of the following symbols: "--, -, o, +, ++", with "--" being the worst case and "++" being optimal support respectively.

Additional help in selecting appropriate patterns is provided by dividing the complete collection of patterns into different categories. These categories, organized in terms of typical areas of modeling, simply serve to reduce the amount of patterns a developer has to scan. In general refinement patterns can be categorized into the following pattern families:

- **Fundamental** patterns describe the refinement of basic object-oriented concepts.
- **Interaction** patterns describe refinements of how objects work with each other.
- **Aggregates** patterns describe refinements of object structures (e.g., inheritance).
- **Architecture** patterns describe refinements of system architecture constructs.

Another important factor influencing the usability of refinement patterns is their overall design. The patterns informational content, which has to be communicated to a developer, must be structured in such a way that relevant information can be found easily. Our own experiences with the application of patterns in software projects show that forms of the kind illustrated in [7], are best suited for this purpose. This personal experience is also supported by findings in psychological research. In accordance with these findings, refinement patterns are presented using one-page forms divided into three horizontal areas. Having a standard presentation template of this form also helps developers define their own patterns by providing an exact definition of the information needed. An example refinement pattern, describing the refinement of an unidirectional association, is shown in Figure 6.

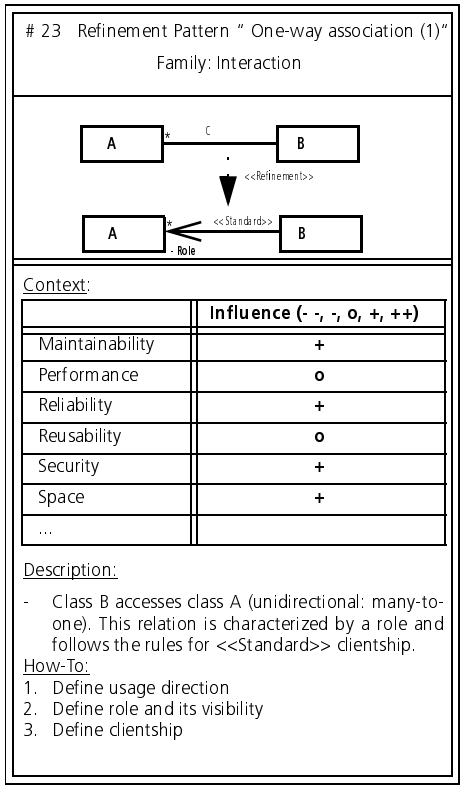


Figure 6: Refinement Pattern (1)

- The top area is used for storing administrative information (i.e., unique pattern number, pattern name, and pattern family). These are used help developers find a specific pattern quickly, and to support the use of tools for maintaining the complete set of patterns.
- The area in the middle of each pattern contains a graphical description of the actual refinement. This description depicts how a specific UML construct

(e.g., a unary association) can be refined to a UML-NOF construct (e.g., a clientship relationship) by presenting both constructs and their refinement-relationship.

- Finally, the bottom area characterizes the context in which the pattern is applicable, augmented by textual descriptions of the refinement itself. The context characterization provides a subjective estimate of the pattern's influence on various quality factors. In addition the refined construct (i.e., the UML-NOF construct) and the refinement itself are explained textually, to allow for better understanding.

In the context of the work presented here we used the UML documentation [17], standard books on patterns [5, 12], various books on software development with the UML [10, 13, 14, 15, 16], as well as our own software development experiences to capture the required knowledge for defining a first set of refinement patterns.

4.2 Translation Patterns

Translation patterns perform a similar job to refinement patterns, but for translations instead of refinements. Although the two are strictly separated, however, the relationship between the leaves of the refinement tree and the associated programming constructs must be clear. In other words, translation patterns are not solely oriented towards implementing a NOF modeling element, but also towards continuing the process of developing code from models seamlessly. Then, and only then, it is possible for a developer to traverse a software system from early analysis to code.

Like refinement patterns, translation patterns are usually not applicable in all circumstances. Usually there are several applicable translation patterns for a given NOF element, because the context (e.g., non-functional requirements) plays a major role in which translation is the most suitable (e.g., a system needing high-speed will be implemented in a different way than a system where security and safety are the major concerns). Consequently a translation pattern has to provide the same context information to allow developers to choose the pattern most suitable for their needs. As for refinement patterns, we use subjective estimates ("--, -, o, +, ++") as to what extent the translation pattern influences certain quality factors. These can differ from those of related refinement patterns. Translation patterns are also presented using the same template as refinement patterns (see Figure 7).

The patterns developed to date are aimed at translating NOF-UML models to C++ in the domain of information systems. The knowledge required to define these patterns was derived from different sources, such as books on OOP [6] or OO programming languages [19, 11], and practical experiences of OO develop-

ers. Each pattern was then developed with a view to its application context and target language.

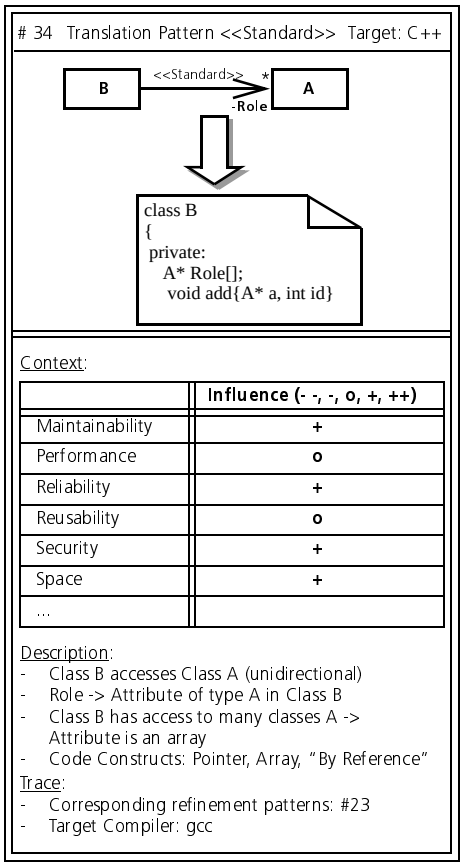


Figure 7: Translation Pattern (1)

4.3 Applying Refinement and Translation Patterns

The collective set of refinement patterns and translation patterns can be seen as a knowledge archive for developers, which they can use to choose the pattern most suitable for their specific needs. In practice the *SORT* process of refining models to Normal Object Form, and then translating them to code, starts which the identification of the model elements which are not in Normal Object Form. For each of these constructs a set of patterns, generally applicable for the specific construct type, is collected. The developer is then responsible for selecting and applying specific patterns from this set, based on the patterns context information and the prevailing non-functional requirements. This procedure is continued until the complete model is refined. The subsequent translation of the refined model is done in the same manner by selecting the set of applicable translation patterns for each NOF construct, selecting the most suitable one

given the prevailing context information, and applying this pattern. By repeating this procedure a developer will finally end up with an implementation of the original model adapted to the non-functional requirements of the system. More importantly, the exact sequence of decisions which the developer took in getting from the high-level analysis models to the detailed implementation is documented, and their rationale explicitly described (traceability).

SORT provides a high-level of flexibility because a developer can individually combine refinement and translation patterns based on the provided context information. An example of this flexibility is shown in Figure 8. The example describes the implementation of an association between classes by applying two different refinement patterns (see Figure 6 and Figure 9 respectively) to refine the construct to the NOF level, and by applying two translation patterns (see Figure 7 and Figure 10 respectively).

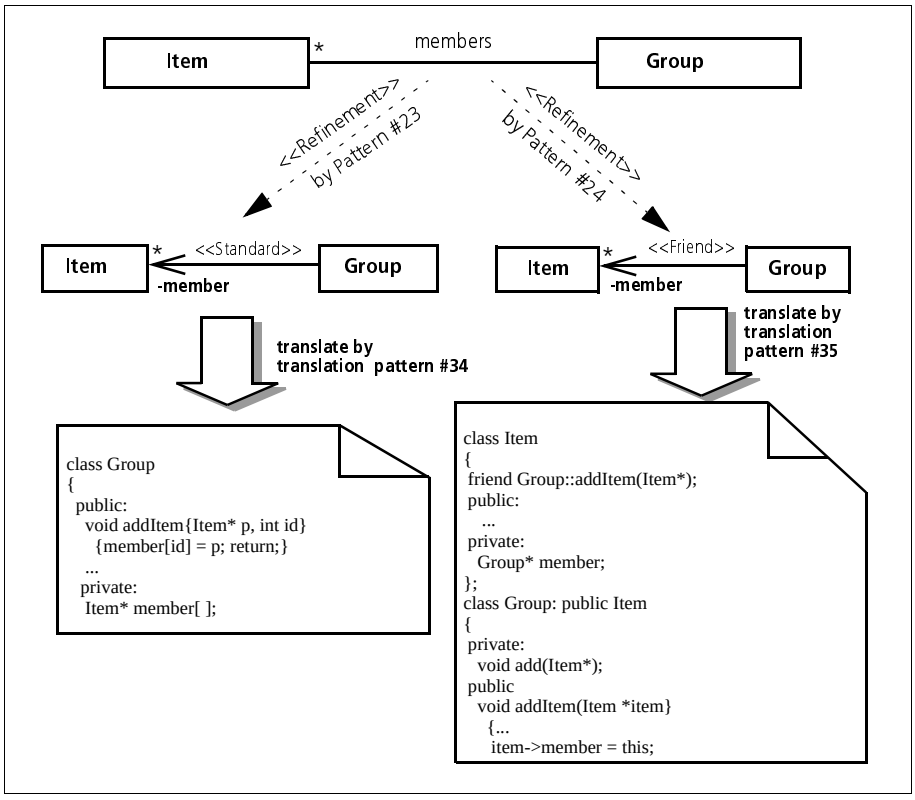


Figure 8: Example Application

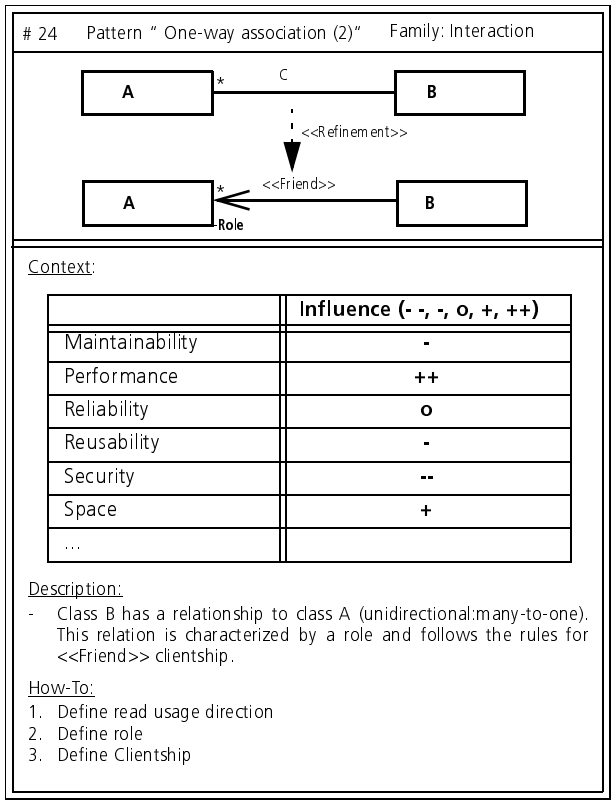
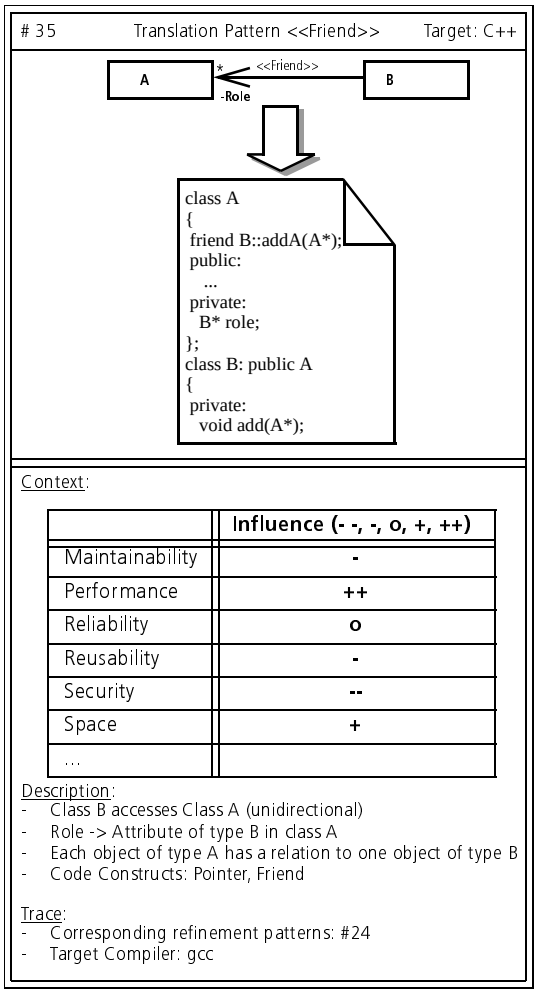


Figure 9: Refinement Pattern (2)

The example shows how non-functional requirements drive the refinement and translation activities. If a developer is developing a software system where reliability and security is a concern he will probably choose the left path by applying the refinement pattern #23 and translation pattern #34. This gives an implementation based on an array of pointers which allows access to the public parts of 'Item' objects. This kind of implementation is reliable because adding/removing pointers is easy, and the program can simply walk through the array, one element at a time, using the pointer to invoke methods of the referenced object. Otherwise he may choose the right path (patterns #24 and #35) provided that it is acceptable that the relationship violates the principle of information hiding (security) but allows a fast traversion even from the one to the many side.



Context:

	Influence (- -, -, o, +, ++)
Maintainability	-
Performance	++
Reliability	o
Reusability	-
Security	--
Space	+
...	

Description:
- Class B accesses Class A (unidirectional)
- Role -> Attribute of type B in class A
- Each object of type A has a relation to one object of type B
- Code Constructs: Pointer, Friend

Trace:
- Corresponding refinement patterns: #24
- Target Compiler: gcc

Figure 10: Translation Pattern (2)

The example shows that the application of *SORT* results in various different implementations of a modeling construct based on different context factors. The number of different implementations can increase further if new refinement and/or translation patterns come into play.

By applying *SORT,* a modeling construct can be first refined in various different ways and then be translated in various different ways. Consequently the 'possible' refinements and translations build up a tree (see Figure 1), and the task of the developer is to choose the appropriate path for this tree. This flexibility enables developers not only to understand and document the relationships between their models and code better, but also to fulfil the specified non-functional requirements.

5 Conclusion

With the rapid rate of innovation in object technology, one might have expected to have seen significant improvements in the quality of object-oriented systems. In practice, however, this has not happened due to the primitive techniques still largely dominating the "implementation" phase of object-oriented development methods. Leading edge technologies such as distributed objects, components and the UML will never achieve their full potential until this problem is addressed. Even the most notable development in this field, the "implementation through translation" approach popularized by Shlaer and Mellor, essentially sidesteps the problem by focusing more on performing, than on defining, the mapping between models and code. Like the automatic code generation facilities of CASE tools, therefore, such meta-modeling approaches have great difficulty in helping developers tailor implementation strategies to best meet their needs.

In an ideal world software systems would be written in formal languages, with formally verified mappings between levels, but for object-oriented development this is a long way off in practice. *SORT* offers a practical approach to this problem which provides effective assistance in the mapping of UML analysis and design models to code. This is achieved through the application of three time-honoured strategies for managing complexity:

1 *Reducing the size of the individual steps.* A sequence of several smaller steps is easier to understand than one big step (e.g., in fact, this is the approach taken in mathematics to prove a theorem in terms of "smaller" parts, which then together show the correctness of the whole theorem). With the *SORT* approach the task of implementing graphical models is broken into individual refinement and translation substeps.

2 *Separating concerns.* Developers can concentrate on single activities and do not have to worry about several things at once. These single steps facilitate a more systematic refinement of graphical models to code-level abstractions before starting translation.

3 *Identifying and exploiting commonality.* The *SORT* approach simplifies multiple implementations of a single model (i.e. porting). In general, developing a new implementation of a model (e.g., using another programming language) requires a large amount of effort. With *SORT* the reimplementation need only to deal with the retranslation of the NOF models.

As a by product, the NOF also provides a partial solution to a major problem with which users of the UML are now faced. The UML provides a large number of different modeling constructs which may be used to model a given structure and/or behaviour, but provides no guidance as to what constructs should be used. The NOF provides a first step towards addressing this problem by identify a restricted extension of the UML which is optimized for a specific phase of the development process: namely, the detailed design phase.

Currently we are planning to evaluate the work presented here in the context of industrial case studies. Recent empirical evidence has shown that quality guidelines can have a major impact on the quality of object-oriented systems [3, 4]. Therefore, further work will concentrate on incorporating such quality guidelines into *SORT* to ensure that refinement and translation patterns represent best-practice knowledge for developing high-quality software.

References

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

[2] Shirley A. Becker and James A. Whittaker. *Cleanroom Software Engineering Practices*. Idea Group Publishing, Harrisburg, USA; London, UK, 1997.

[3] Lionel Briand, Christian Bunse, John Daly, and Christiane Differding. An experimental comparison of the maintainability of object-oriented and structured design documents. *Journal of Empirical Software Engineering*, 2(3), 1997.

[4] Lionel C. Briand, Christian Bunse, and John W. Daly. An experimental evaluation of quality guidelines on the maintainability of object-oriented design documents. In Susan Wiedenbeck and Jean Scholtz, editors, *Proceedings of the Empirical Studies of Programmers: Seventh Workshop ESP7*, pages 1–19. ACM Press, October 1997.

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

[6] P. Coad and E. Nicola. *Object-Oriented Programming*. Prentice Hall, 1993.

[7] Peter Coad, David North, and Mark Mayfield. *Object Models - Strategies, Patterns, & Applications*. Computing Series. Yourdon Press, second edition, 1997.

[8] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison Wesley, October 1998.

[9] M. Dyer. *The Cleanroom Approach to Quality Software Development*. John Wiley and Sons, Inc., 1992.

[10] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit*. John Wiley & Sons, 1998.

[11] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc, 1996.

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

[13] Bernd Kahlbrandt. *Software-Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language*. Springer-Verlag, 1998.

[14] Craig Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 1998.

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

[16] Jesse Liberty. *Clouds to Code. A Case Study in Application Development with UML, Design Patterns and C++*. Wrox Press Ltd., 1998.

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

[18] 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.

[19] Bjarne Stroustrup, editor. *The C++ Programming Language*. Addison Wesley, second edition, 1993.

[20] Bruce F. Webster. *Pitfalls of Object-Oriented Development*. M & T Books, 1995.

Document Information

Title:	Improving Quality in Object-Oriented Software: Systematic Refinement and Translation of Models to Code
Date:	June 25, 1999
Report:	IESE-036.99/E
Status:	Final
Distribution:	Public