

Formal Definition and Refinement of UML's Module/Package Concept

Andy Schürr, Andreas J. Winter
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, D-52074 Aachen, Germany
[andy | winter]@i3.informatik.rwth-aachen.de
[http://www-i3.informatik.rwth-aachen.de/people/\[andy|winter\]](http://www-i3.informatik.rwth-aachen.de/people/[andy|winter]).

1. Introduction

After about 20 years of research and development, object-oriented (OO) modeling methods and notations have reached a certain degree of maturity and acceptance. They are no longer the occult science of a small number of OO gurus, but the widely accepted approach for analysis (OOA) and design (OOD) of software (hardware) systems. Popular OO methods - like Booch [3], OMT [17], or OOSE [9] - are nowadays used to develop systems of continuously increasing size and complexity. As a consequence, produced analysis or design documents may consist of hundreds or even thousands of elements (classes, relationships, etc.). Keeping these documents in a consistent state or reusing generic parts of one analysis/design document within another one is a nightmare without the existence of any module concept.

These problems should be familiar for software developers of the late 60ies. Well-known software engineering concepts like “abstract data types” [13] and “programming-in-the-large” [6] have been invented to overcome these problems. They lead to the development of modular programming languages like Modula-2 or Ada [22] and software design languages like HOOD [16] or EMIL [4]. For a long time these ideas didn't have any significant impact onto the development of OOA/OOD notations. The first generation of OO methods simply ignored the problem. Later on developed approaches offered more or less ad hoc solutions for partitioning analysis and design documents (diagrams) into surveyable pieces. OMT [17] offers for instance so-called “modules”, which allow for the decomposition of unmanageable diagrams into a number of related diagrams. But there are no means to construct exports or imports of modules. As a consequence, any two elements in two different diagrams with the same name have to be identified. And even more elaborate concepts like categories in Booch [3], collaborating subsystems with their contracts in Wirs-Brock [23], or subsystems with well-defined interfaces in ADM3 [8] do not study the interactions between information hiding, module boundary crossing associations and inheritance.

The proposal of a *Unified Modeling Language* (UML) as a successor of Booch, OMT, and OOSE, is in our opinion the first OO notation which addresses all facets of a state-of-the-art module concept [14]. Its modules, called *packages*, build shells around arbitrary types of diagrams (static structure diagrams, collaboration diagrams etc.). They offer an information hiding concept which was heavily influenced by the design of C++ [7]:

- (1) *Explicit import* relationships have to be used to access *public* elements of one package within another package,
- (2) *refinement* (generalization) relationships provide additional access to *protected* elements, and
- (3) *friend* relationships reveal even *private* (but not *implementation*) elements of one package to another package.

Furthermore, UML supports nesting of packages with visibility rules derived from nested scope rules of programming languages à la Modula-2 or Ada.

Our main problems with UML's module concept are as follows:

- (1) All concepts are defined in natural language only. This makes it sometimes really difficult to determine the precise semantics of introduced terms. Consider for instance a sentence like “*In this context, reference means that the referenced Element instance is visible ...*” on page 7 of the UML semantics definition [14]. Does “means” mean “required” or “is equivalent to the fact”?
- (2) Import, refines, and friend relationships between packages are indirect subclasses of Element and inherit its property to possess a Visible attribute with values from the ordered set $\{public > protected > private > implementation\}$. But we didn’t find a single line that explains the consequences of dependency visibilities for connected packages, although the usage of import and refines relationships with varying visibility values makes sense from the software engineering point of view.
- (3) Many strictly necessary constraints like “a client A of another (imported/referenced) package B should not add (own) an import relationship from B to another package C ” are either not part of the UML semantics definition or carefully hidden in its 103 pages.

This paper is our attempt to translate the natural language definition of UML’s module concept into twelve predicate logic formulas that give precise answers to points (1) and (2) above. Furthermore, we will suggest five additional formulas that address point (3) above. Within all these formulas we will use the following (all-quantified) variables:

$$P, P', P'' \in \text{Package is_a Element}, E \in \text{Element}, dep \in \{\text{exp_imports}, \text{refines}, \text{friend}\}$$

$$\varpi, \varpi', \varpi'' \in \{public > protected > private > implementation\}$$

2. Aggregation of Elements and Packages

Any UML document contains a number of top-level packages which represent the regarded system model. Each package defines a visibility shell around a number of elements, which are either (a) basic constructs of a certain type of diagrams or (b) nested packages or (c) dependencies between nested packages. A package A contains an element E if it owns or references (uses) E , which belongs to another package B . In the latter case, where A references E , E must be visible inside A . This situation is captured by the following five predicate logic formulas. Please note that formula (5a) considers only *visibility of nested package elements*. The interactions between visibility of elements and (explicit) import relationships as well as refinement and friend relationships are the subject of sections 3, 4, and 5, respectively.

Due to the lack of space the following formulas cannot be explained in detail. We do hope that almost all of them are self-explanatory as soon as the role of ϖ variables is clear. Terms like

$$P \text{ owns } \varpi E \text{ or } P \text{ references } \varpi E \text{ or } P \text{ contains } \varpi E \text{ or } \dots$$

have to be interpreted as “package P owns/references/contains/... the element E with the associated visibility $\varpi \in \{public > protected > private > implementation\}$ ”. Please note that the concepts *owns*, *references*, and *sees* (visibility) are defined in UML, whereas *contains* and *offers* are our own inventions in order to keep formulas as simple as possible.

- (1) *Owner of element is unique (page 4 of [14]):*
 $P \text{ owns } \varpi E \wedge P' \text{ owns } \varpi E \rightarrow P = P'$
- (2) *Elements (from other packages) may be referenced if visible (page 7 of [14]):*
 $P \text{ references } \varpi E \rightarrow P \text{ sees } \varpi E$
- (3) *Contains relationship is union of owns and reference relationship:*
 $P \text{ contains } \varpi E \leftrightarrow P \text{ owns } \varpi E \vee P \text{ references } \varpi E$
- (4) *Offers relationship is transitive closure of contains relationship (page 9 of [14]):*
 $P \text{ offers } \varpi E \leftrightarrow P \text{ contains } \varpi E \vee \exists P': P \text{ contains } \varpi P' \wedge P' \text{ offers public } E$
- (5a) *Visibility of elements (of nested packages) is determined as follows (page 9 of [14]):*
 $P \text{ sees } \varpi E \leftrightarrow P \text{ offers } \varpi E \vee \dots$

The most important consequence of the definitions above is that a package sees all public elements of nested packages (visibility is transitive), where nested packages are either locally defined packages or imported (referenced) packages. A surrounding package has no possibility to hide public elements of nested packages at its own interface, except by owning or referencing nested packages themselves with a lower visibility priority.

3. Export/Import of Packages

We have seen that nesting of packages gives surrounding packages access to the public elements of enclosed packages. Therefore, UML says that owns and reference relationships establish a kind of *implicit import*. Using implicit imports only a package would never be able to reference elements at the interface of sibling packages (a package may only reference visible packages, and reference relationships were our only means in section 2 to make foreign packages visible). Therefore, UML introduced the concept of *explicit import* as a dependency relationship between packages that belongs to the common surrounding package of the related client (target) and server (source) package. These import dependencies have their own visibility attributes. A *public import* represents for instance a kind of interface import, which is visible for all clients of the surrounding package. An *implementation import*, on the other hand, is an always hidden import, which represents local analysis or design decisions.

The following four formulas are our attempt to formalize imports and exports of packages. Please note that packages do not have any means to define sets of exported elements explicitly. Their *public/protected/... exports* are always implicitly determined as their sets of public/protected/... visible offered elements. As a consequence, packages do not only export own elements, but also referenced elements from other packages. This takes from client packages the cumbersome burden to import all those elements of other packages which are used in the interfaces of already imported packages.

- (6) *Implicit import are all indirectly owned or referenced elements (page 8 of [14]):*

$$P \text{ imp_imports } \varpi E \leftrightarrow \exists P': P \text{ contains } \varpi P' \wedge P' \text{ offers public } E$$
- (7) *Import is the union of explicit and implicit import (page 8 of [14]):*

$$P \text{ imports } \varpi E \leftrightarrow P \text{ exp_imports } \varpi E \vee P \text{ imp_imports } \varpi E$$
- (8) *Export is set of all offered nonimplementation elements (page 10 of [14]):*

$$P \text{ exports } \varpi E \leftrightarrow P \text{ offers } \varpi E \wedge \varpi > \text{ implementation}$$
- (5b) *Visibility of elements across packages is extended as follows (page 8 of [14]):*

$$P \text{ sees } \varpi E \leftrightarrow P \text{ offers } \varpi E \vee (\exists P': P \text{ exp_imports } \varpi P' \wedge P' \text{ exports } \varpi E)$$

4. Refinement of Packages

The previous two sections introduced the “classical” modularization concepts of programming languages like Modula-2 (as defined in the ISO/IEC 10514-1 standard [18]), i.e. the construction of export interfaces for packages (with varying degrees of visibility inherited from C++), nesting of (local) packages, and the establishment of visible or hidden import relationships between packages. These import relationships permit access to public interface elements of server packages, only. The remaining two visibility values (for interface elements) — *protected* and *private* — are only useful in combination with refinement (generalization) and friend relationships between packages. The concept of friends is the subject of the following section, whereas the concept of refining/generalizing packages will be explained here.

The main motivation for introducing the *refinement (subtype)* relationship between packages is that the important OO concept of *inheritance* should not only be available for defining single classes, the basic elements of static structure diagrams, but also for defining and refining arbitrarily complex subdiagrams.

It is not at all difficult to come up with a precise definition of the consequences of refinement (generalization) relationships for the visibility of package elements as well as with a formal definition of the constraint “*generalization relationships do not build cycles*”. It is far more difficult to translate the meaning of sentences like “... *an instance of the subtype is substitutable for an instance of the supertype*”. The latter constraint cannot be defined for packages in general, but must be studied for each language of UML diagrams, separately. Such a precise definition of the term “substitutability” is not part of UML. Therefore, our formulas will only take the consequences of public refinement relationships for the visibility of (public) interface elements of related packages into account. For further details concerning the formal treatment of subtyping from an algebraic point of view the reader is referred to [5] and from a type-theoretic point of view to [12]. It is an open question whether similar constraints have to be added for the case of non-public refinement relationships and interface elements.

- (9) *SubtypeOf relationship is transitive closure of refines relationship:*

$$P \text{ subtypeOf } P' \leftrightarrow \exists P'', \varpi: P \text{ refines } \varpi P'' \wedge P'' = P' \vee P'' \text{ subtypeOf } P'$$
- (10) *Refines (generalization) relationship is acyclic (page 36 of [14]):*

$$\neg(P \text{ subtypeOf } P)$$
- (11) *Public export of refining package has refined package's export (page 38 of [14]):*

$$P \text{ refines public } P' \wedge P' \text{ exports public } E \rightarrow P \text{ exports public } E$$
- (5c) *Visibility across package boundaries is extended as follows (page 8 of [14]):*

$$P \text{ sees } \varpi E \leftrightarrow \dots (* \text{ see Def. (5b) } *)$$

$$\vee \exists P', \varpi', \varpi'' \geq \text{protected}: P \text{ refines } \varpi' P' \wedge P' \text{ exports } \varpi'' E \wedge \varpi = \min(\varpi', \varpi'')$$

Please note that formula (5c) above is just an extension of formula (5b) of the previous section. It takes refinement relationships with different degrees of visibility into account, ranging from a kind of *public subtype inheritance* to pure *implementation inheritance*. It says that a refining (subtype) package sees all public elements of the refined (supertype) package as public elements (if the refinement relationship is public, too). It states furthermore that a refining package sees all protected elements of the refined package as $\varpi \leq \text{protected}$ visible elements (if the refinement relationship is ϖ visible, too). It is an open question, whether it makes sense to have four different visibility cases for refinement relationships, instead of the usual distinction between interface preserving subtype inheritance and the hidden inheritance of implementations.

5. Friends of Packages

Last and least we have to deal with the C++ concept of friends, whose value for object-oriented analysis and design tasks seems to be doubtful, if compared with modularization concepts of programming languages like Modula-3 or Java. These languages allow for the definition of an arbitrary number of interfaces for different categories of clients of a single module (package). Nevertheless, we present the two additional formulas for friend relationships for reasons of completeness. *Friend relationships* may only exist in parallel to explicit or implicit import relationships (but seemingly not parallel to refinement relationships). They may be used to break another informing hiding wall around the contents of a server package and give access to all public, protected, and private elements of a package (but not to its implementation elements). The corresponding two formulas look like follows, with the second formula being another extension of the formula (5b) and (5c) of sections 3 and 4, respectively.

- (12) *Friend relationships require parallel import relationship (page 8 of [14]):*

$$P \text{ friend } \varpi P' \rightarrow P \text{ imports } \varpi P'$$
- (5d) *Visibility across package boundaries is extended as follows (page 8 of [14]):*

$$P \text{ sees } \varpi E \leftrightarrow \dots (* \text{ see Def. (5c) } *)$$

$$\vee \exists P', \varpi', \varpi'' \geq \text{private}: P \text{ friend } \varpi' P' \wedge P' \text{ exports } \varpi'' E \wedge \varpi = \min(\varpi', \varpi'')$$

6. Further Consistency Constraints

The developed formal definition of the UML package concept shows that the original natural language definition seems to be free of any contradictions, but is not all complete. As a consequence, we had to add a number of own assumptions, without knowing (until now) whether these add-ons respect the original intentions of the three amigos Booch, Jacobson, and Rumbaugh. These add-ons were kept as minimal as possible within the previous sections, thereby postponing the definition of a number of important *additional constraints* to this section. All these constraints, which will be presented below, are—in our opinion—not a matter of debate, but should be part of UML. It is for instance useless to see a dependency without seeing its source or target package (13). Furthermore, a package should never be able to define additional server packages or supertype packages for not locally defined packages (14). Otherwise, it would be possible to extend and modify the implementation of imported packages, thereby violating the very principle of information hiding. Last but not least it makes no sense that a package refines one of its locally defined packages (15) or that a supertype package needs (depends on) one of its subtype packages (16, 17).

(13) *Visibility of import/refines dependency is less equal visibility of related packages:*

$$P \text{ dep } \varpi P' \rightarrow \exists P'', \varpi', \varpi'':$$

$$P'' \text{ owns } \varpi \text{ dep} \wedge P'' \text{ offers } \varpi' P \wedge P'' \text{ offers } \varpi'' P' \wedge \varpi \leq \min(\varpi', \varpi'')$$

(14) *Source of import/refines dependency belongs to owner of dependency:*

$$P \text{ dep } \varpi P' \rightarrow \exists P'', \varpi': P'' \text{ owns } \varpi \text{ dep} \text{ and } P'' \text{ owns } \varpi' P$$

(15) *A package should neither import nor refine its offered (own) packages:*

$$P \text{ dep } \varpi P' \rightarrow \neg \exists \varpi': P \text{ offers } \varpi' P'$$

(16) *Needs relationship is transitive closure of imports and refines:*

$$P \text{ needs } P' \leftrightarrow$$

$$\exists P'', \varpi: (P \text{ refines } \varpi P'' \vee P \text{ imports } \varpi P'') \wedge (P'' = P' \vee P'' \text{ needs } P')$$

(17) *Package may not refine a package which is an (in-)direct client of itself:*

$$P \text{ refines } P' \rightarrow \neg(P' \text{ needs } P)$$

7. Summary

This paper presented a very compact definition of the UML notation's modularization concept and suggested a number of useful extensions. These extensions compensate obviously existing incompletenesses of UML's natural language definition in [14] or represent additional policies for the definition of import, refinement, and friend relationships between packages. The precise definition of such a OOA/OOD module concept is not an isolated activity at our department, but an integral part of the following projects:

- (1) Some years ago, people at our department developed the module interconnection language EMIL, which offers different types of modules, nesting of (sub-)systems, import/export relationships between modules and (sub-)systems, inheritance for abstract data type modules as well as genericity in the sense of generic Ada packages [11, 4].
- (2) Our formal background are logic-based graph rewriting systems [19]. They are used—in the form of the visual specification language and environment PROGRES [21]—to define graphical software engineering languages and to prototype tools for these languages. Currently, we are busy to specify and prototype a significant subset of UML in PROGRES.
- (3) Resulting graph rewriting specifications for complex languages like UML tend to be too large to be written down as a single unstructured document. Therefore, we are now starting to develop a module concept for PROGRES, which will be similar to the module concepts of EMIL and UML [20].

To summarize, neither UML's module concept itself nor the considerations presented here concerning its formal definition and necessary modifications are restricted to a single object-oriented analysis and design method. On the contrary, the presented *module concept may be added to other analysis, design, or specification languages* or it may even build the basis for a separate module interconnection language. This is due to the fact that presented formulas make no assumptions about the semantics of basic elements in packages. It is their exclusive purpose to explain the impact of packages and relationships between packages on the visibility of package elements. As a consequence, this paper complements the rapidly growing number of publications which have either the formal definition of module interconnection languages (architecture styles) or certain OO diagram types as their main topic. Both categories of papers assume either very simple visibility rules (compared with the visibility rules introduced here) or neglect this aspect at all due to the absence of a module concept. Examples of the first category are for instance the VDM-SL definition of the Modula-2 standard [18] or the Z definitions of various data flow or event based architecture styles [1, 15]. Examples of the second category are formal definitions of (subsets of) OMT [10] and Fusion [2].

References

- [1] Abowd G., Allen R., Garlan D.: *Formalizing Style to Understand Descriptions of Software Architecture*. ACM Transactions on Software Engineering and Methodology, 4(4):319–364, 1995.
- [2] Bates B., Bruel J., France R., Larrondo-Petrie M.: *Guidelines for Formalizing Fusion Object-Oriented Analysis Models*. In Proc. CAiSE'96, number 1080 in Lecture Notes in Computer Science, pp. 222–233. Springer Verlag, Berlin, 1996.
- [3] Booch G.: *Object-Oriented Analysis and Design*. Benjamin Cummings Series in Object-Oriented Software Engineering. Benjamin Cummings, Redwood City, CA, 1994.
- [4] Börstler J.: *Programmieren-im-Großen: Sprachen, Werkzeuge, Wiederverwendung*. Dissertation (RWTH Aachen), TR UMINF 94.10, Department of Computer Science, Umeå University, Sweden, 1994.
- [5] Breu R.: *Algebraic Specification Techniques in Object-Oriented Programming Environments*. Number 562 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1991.
- [6] DeRemer F., Kron H.: *Programming-in-the-large versus Programming-in-the-small*. IEEE Transactions on Software Engineering, 2(2):80–86, 1976.
- [7] Ellis M., Stroustrup B.: *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1994.
- [8] Firesmith D. G.: *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach*. John Wiley, New York, 1993.
- [9] Jacobson I.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA, fourth edition, 1994.
- [10] Jonckers V., Verschaeve K., Wydaeghe B., Cuypers L., Heirbaut J.: *OMT*, Briding the Gap between Analysis and Design*. In Proc. FORTE'95, pp. 39–55. Chapman & Hall, 1996.
- [11] Nagl M.: *Softwaretechnik: Methodisches Programmieren im Großen*. Springer-Verlag, 1990.
- [12] Palsberg J., Schwartzbach M. I.: *Object-Oriented Type Systems*. John Wiley, New York, 1994.
- [13] Parnas D.: *A Technique for Software Module Specifications with Examples*. Communications of the ACM, 15:330–336, 1972.

- [14] Rational Software Corporation: *UML Semantics, Version 1.0*. <http://www.rational.com>, 1997.
- [15] Rice M., Seidman S.: *A Formal Model for Module Interconnection Languages*. IEEE Transactions on Software Engineering, 20(1):88–101, 1994.
- [16] Robinson P. J.: *Hierarchical Object-Oriented Design*. Prentice Hall, Englewood Cliffs, MA, 1992.
- [17] Rumbaugh J., Blaha M., Eddy W. P. F., Lorenzen W.: *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [18] Schönhacker M., Pronk C.: *ISO/IEC 10514-1, the Modula-2 standard: Changes, Clarifications, and Additions*. ACM SIGPLAN notices, 31(8):84–95, 1996.
- [19] Schürr A.: *Logic Based Programmed Structure Rewriting Systems*. Fundamenta Informaticae, XXVI(3/4), 1996.
- [20] Schürr A., Winter A. J.: *Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems*. Technical Report AIB 97-3, RWTH Aachen, Germany, 1997.
- [21] Schürr A., Winter A. J., Zündorf A.: *Graph Grammar Engineering with PROGRES*. In Schäfer W., Botella P. (eds.): Proc. 5th European Software Engineering Conf. (ESE-EC'95), volume 989 of Lecture Notes in Computer Science, pp. 219–234. Springer Verlag, Berlin, 1995.
- [22] Wiener R. M., Sincovec R.: *Software Engineering with Modula-2 and Ada*. John Wiley, New York, 1984.
- [23] Wirfs-Brock R., Wilkerson B., Wiener L.: *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, NJ, 1990.