Instance Level Packages for UML

P.J. 't Hoen*, G. Busatto, and L.P.J. Groenewegen

Leiden University, Leiden Institute of Advanced Computer Science, P.O. Box 9512,
2300 RA Leiden, The Netherlands

Abstract. Current OO modelling languages like UML and programming languages like Java use some form of modules of classes to restrict the possible interaction between objects. Class visibility is however not sufficient to forbid all undesired interaction between objects. Objects instantiated from a class of a module can interact with other objects instantiated from the private classes of this module. In the paper we explain by means of an example why this can be a problem. We need not only the concept of private class, but also the concept of private object. We introduce instance level modules to encapsulate objects when we need multiple instances of a module of classes. We base our example modules and definitions on packages from the industry standard UML.

1 Introduction

When an OO model (or program) is constructed, a set of classes is the end result of an analysis of the problem domain. See [RBP+91, Mey97, Boo91, GJM91, UML97, RTF99] and [Str97, GM95] for details. When classes are conceptually related, or may not be accessible to all other classes of the model, they are placed into distinct modules. These class modules make the structure of the model more explicit and can be used for the desired encapsulation. Limited visibility of classes is used to ensure only authorised access exists between the various objects instantiated from the classes. Restricting visibility between classes using only class modules is however not sufficient to remove all undesired access between objects. We give a small example to illustrate our point.

Consider a simple class module *Bank* which represents a basic bank and a class *Customer* which models a customer of the bank. The bank manages the accounts of its customers, gives loans, reposesses houses and so on. The class module *Bank* for our example only contains two classes; one for the front office dealing with the outside world and one for the internal accounting affairs. The first class *frontoffice* is public, i.e. it is visible to the rest of the classes of the model. Specifically, an object of type *Customer* can access the front offices of a bank. The second class *accounting* for the internal affairs of the bank is private to the module. Only objects instantiated from classes representing the front office or accounting department are able to access the internal affairs of a bank. Thus, the accounting departments of a bank are hidden from an object of type *Customer*

* Please use the primary author's contact address: `hoen@wi.LeidenUniv.nl`

and a bank can keep up its professional businesslike appearance. Class visibility is sufficient to ensure that objects instantiated from classes not belonging to *Bank* can not access the accounting departments.

There is however a problem if we want to consider more than one bank. Our model or program then needs to include two (or more) representations of real life banks and we want to limit the possible interactions between the different institutes using our previously developed class module *Bank*. We get multiple sets of objects instantiated from the classes *frontoffice* and *accounting* where each separate set represents one of the banks. An instance of *Customer* can still only access the front offices of the various banks. An object representing a part of any bank can however access all objects from all other banks, including the objects which represent the accounting department(s) as any object of type *frontoffice* or *accounting* can access any object of type *accounting*. This is not a realistic representation of the cut-throat world of banking. The encapsulation rules for classes however make no distinction between the different instances of *Bank*. All objects instantiated from the classes of *Bank* have the same access right. Class visibility as commonly applied is only sufficient to ensure that objects instantiated from the private classes of a class module are not accessed by objects instantiated from classes outside of the class module.

An elegant solution to this problem is to use *object modules*, i.e. modules of objects. Analogous to class modules, object modules group objects. For the example, we place each separate set of objects which represents one bank into one separate object module. This is useful by itself as by placing these objects into modules we make the common role of these objects in our model explicit. More importantly, we use these same object modules to restrict further the visibility of the objects of the various accounting departments. Each object module encapsulates all objects of type *accounting*. By hiding within the object modules the objects which are instantiated from the private classes from the class module we ensure that there is no undesired access from the objects of one bank to the private objects of a *different* bank. Object modules add visibility constraints for objects not defined by the visibility constraints for the parent classes. We introduce the concepts of public and private objects. We define *instance level modules* as special object modules which model different instances of one class module.

In Section 2, we once more present the above scenario but now more formally using UML diagrams as defined in [UML97, RTF99] as UML is becoming a de facto standard for OO notation. For modules we use UML packages. This means we have class and object packages. In Section 3, we define *instance level packages*, a special type of object package and we show how the problem scenario from the introduction and Section 2 using these instance level packages is resolved. In Section 4 we make some concluding remarks.

2 Problem Scenario

In Section 1, we have sketched a problem scenario involving a customer and one or more banks. In this Section we model this same scenario using the notation for classes and UML modules, i.e packages. We first present the necessary classes and packages. We then present the scenario for one customer and just one bank. We extend this example by adding one bank to show the specific problems involved with two or more banks. Section 3 gives a general solution for this type of scenario.

Based on the original scenario from Section 1, we define a class *Customer* and a Package *Bank*. The customer can add and withdraw money from his account through the front office of the bank. The internal administration of the bank is done by the accountancy department of the bank.

In [UML97, RTF99], the notation for classes and packages is defined as:

– A class is drawn as a solid-outline rectangle with 3 compartments separated by horizontal lines. The top name compartment holds the class name and other general properties of the class; the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations. Both of the last two compartments may be suppressed.
– A package is shown as a large rectangle with a small rectangle (a "tab") attached on one corner (usually the left side of the upper side of the large rectangle). It is a manila folder shape. The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol ('+' for public, or '-' for private).

The concept packages for UML is very general as packages can contain just about any model element UML cares to define. We call packages restricted to contain only classes *class packages*.

Figure 1 gives the problem scenario in a UML static structure diagram. The diagram includes the class *Customer* and the class package *Bank* with two classes *frontoffice* and *accounting*. The class *Customer* can use the public class *frontoffice* to access his money but not the private class *accounting*.

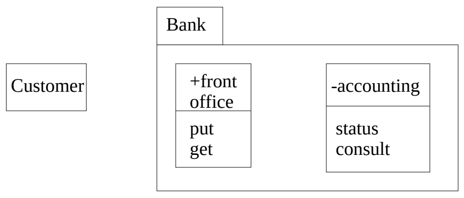

Fig. 1. The *Bank* and the *Customer*

We first instantiate the classes to concrete objects where we describe how one instance of *customer* interacts with the objects of just one bank. Objects in UML are usually drawn in a separate object diagram. For UML, an object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of an instance of the model at a point in time. An object represents a particular instance of a class. The object notation is derived from the class notation by underlining instance-level elements.

In Figure 2, one customer c can choose from either of the two front offices f_1 or f_2 of the same bank. The front offices of this bank share the accounting department a. The customer c can however not access a as the parent of a, the class *accounting*, is hidden from the parent of c, i.e. the class *Customer*. This type of constraint is enforced by the encapsulation of classes by class packages.

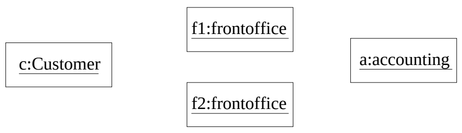

Fig. 2. A *customer* and a bank with two front offices

Consider however the case where we want to model two distinct banks. We then have two sets of objects where the first set represents one bank and the second set the other. We add as objects an f' : *frontoffice* and an a' : *accounting* to represent the second bank. The objects from the set $\{f_1, f_2, a\}$ represent our original bank while the objects from the set $\{f', a'\}$ represent the newly added second bank. We now however have the undesired situation of Figure 3. An object f_i : *frontoffice* or a : *accounting* from the first bank can access the internal accounting department a' : *accounting* of the second bank. Specifically, objects from the first set can access the objects representing the internal accounting departments of second bank. We would like there to be a “firewall” indicated by the dotted line which prevents the access from f_i or a to a'.

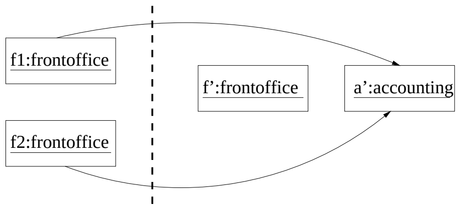

Fig. 3. A second bank

The next Section defines this firewall by introducing *instance level packages*. These are special packages of objects which represent the instances of a class package.

3 Instance Level Packages

In the previous section, we claim that we solve the problem of the incorrect visibility of the objects instantiated from the private classes of a class package by introducing special packages of objects. Before we substantiate this claim, we first make some general remarks on the use of packages and objects in UML. We then give a solution for the example scenario. Based on this example, we give the desired definition of *instance level packages*.

In [UML97, RTF99], a package is a grouping of any model elements. We have already seen in Section 2 how packages are used to group and encapsulate classes and this is the main role of packages in UML. The objects themselves in UML only play a secondary role. The use of object diagrams is fairly limited, mainly to show examples of data structures. Although this is not done, i.e. the possibility is largely ignored, UML allows for packages to group and encapsulate objects. We call packages which only contain objects *object packages*. There are useful opportunities for object packages as is shown below.

In Figure 4, in order to resolve the problems of the scenario, we place the two banks into distinct object packages. One object package contains the objects f_1, f_2 and a of the first bank while the second object package contains the objects f' and a' of the second bank. All objects of type *accounting* are encapsulated by the object packages. These objects have the visibility specifier private. The objects representing the front offices are left public. According to the visibility rules for package elements, only object f' is visible outside of the package *SecondBank*. Objects which represent the first bank can no longer access a' which is private for the second package. The customer c, if located outside of the two packages, is not affected an can still open an accounts at any of the offices of the two banks. The objects packages extend the encapsulation of class packages. By extending encapsulation we mean that our specific object packages from Figure 4 ensure that for objects not only the visibility between the parent classes is considered, but that also the visibility of objects belonging to different sets of objects instantiated from the same classes of one class package is considered.

The concept of private classes is quite common. The class *accounting* is such a class of the class package *Bank* in Figure 1. We now also have the concept of a private object. Object a' of the object package *SecondBank* for Figure 3 is such a private object.

As we already stated, it is already possible in UML to group objects into packages: a package can contain any model elements. A package can also encapsulate its contained model elements. A package in UML is mainly a syntactical tool to group and structure the classes of a model. By restricting ourselves to a class package concept, we get a concept that is sufficiently clear to be able

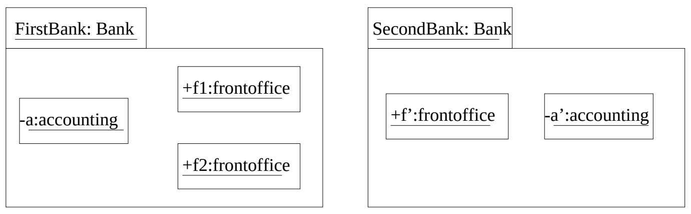

Fig. 4. The two banks in packages

to define a meaningful semantics for it. We define object packages which are instances of class packages.

In our application of packages, we have given a type to the packages of objects. For example, we have introduced *FirstBank* : *Bank*. This is not defined in [UML97, RTF99]. We define: a package of objects O is an *instance* of a package of classes C when:

– All objects of O are instances of classes of C. The objects contained by O are only instances of the classes contained by C.
– The visibility specifier of an object $o : c$ of O is equal to the visibility specifier of c for C. Public classes of C are mapped to public objects of O like private classes of C are mapped to private objects of O.

When O is an instance of C we define that O is an *instance level package* and that $O : C$. The package O is not just any object package but a very specific package which represents an instance of a class package.

With instances of class packages defined, we can distinguish between *different* instances of the same package. Object packages group the different sets of instantiated objects and allow the modeller to regulate the visibility of objects instantiated from one class but which represent different parts of a model.

Note however that we do not claim that using our definition of instance level packages is the only way to instantiate the classes of a package. For example, if for the problem scenario we only want to have *one* instance of the package *Bank*, i.e. only one bank, then it is not a problem to instantiate the classes and leave the objects ungrouped. The visibility rules for classes are sufficient for preventing instances of *Customer* to access the instances of *Accounting*. Also, if the modeller wants for the different instances of *Bank* to access directly their respective accounting departments, then by all means, go ahead and instantiate the classes of *Bank* as unencapsulated sets of objects. Otherwise, the modeller *can* use instance level packages as we have defined to limit the visibility of objects.

We envision the new use of object packages in UML in two specific ways. Object packages either play a role for the object diagram or for the class diagram.

For the object diagram, instance level packages can be used to give examples of multiple instances of packages of classes. We have used this for our problem scenario to model how two different banks can communicate. Alternatively, packages of objects can be used to group related objects which represent different model parts. These same packages can be used to encapsulate private objects. Class packages can be found from these object packages much in the same as classes are found which capture the shared properties of a set of objects. The discovered class packages then represent the common role of a set of different types of objects.

In their second role, instance level packages play an indirect role for the class diagram. We have seen for the example of Section 2 that we want to have the possibility to discern the different instances of one class package. For this purpose we have defined instance level packages. We need to be able to indicate in the class diagram that certain packages can only be instantiated to these instance level packages. For this purpose we add the stereotype *distinct*. Stereotypes represent one of the built-in extensibility mechanisms of UML and a stereotype in UML represents a usage distinction of a model element. A package of classes with the stereotype *distinct* can only be instantiated to instance level packages. The objects instantiated from the classes of the package P with stereotype *distinct* may only occur in the object diagram in packages of objects O where $O : P$. The names of stereotypes in UML are put between << and >>. The general presentation of a stereotype is to place the keyword string above the name of the element. For Figure 5 we once more give the package of classes *Bank* and we indicate that the package may only be instantiated to distinct instance level packages.

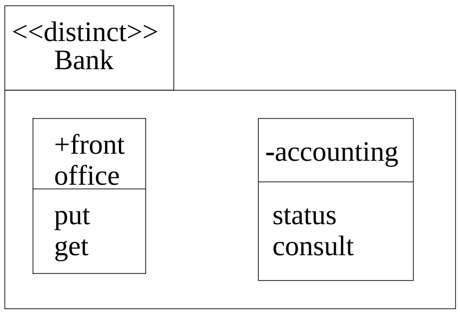

Fig. 5. *Bank* can only be instantiated to instance level packages

Note that we do not claim that object packages *have* to exist for a program which implements our example scenarios for *Bank*. Our object packages need not necessarily be directly implemented at run time. We however *do* want to express the possible legal interactions of the objects which represent different banks/instances of one class package. For our example, our instance level packages capture the fact that the accounting departments of each bank should be private to the bank. A program should ensure that an object of one bank does not access the accounting department of another bank. This *can* be done using packages at runtime.

Up to now, we have kept our examples and first definition simple by ignoring the possible hierarchical nature of packages; a package in UML can contain sub packages. In general, a package may have as elements either non hierarchical model elements like classes or (sub)packages. These elements of a class package are respectively called *class elements* and *(class) package elements*. We call a package which contains package elements a *complex* package. A package element of a complex package is called a *nested* package. Likewise, for a complex package which structures and encapsulates objects, we have *object elements* and *package elements*.

With the above terminology, we can define an instance of a complex class package.

A complex object package O is an *instance* of a complex class package C iff

– each object element of O is an instance of a class element c of C and the visibility specifier of o is equal to the visibility of c.
– Each (complex) object package element op of O is an instance of a (complex) class package element cp of C and the visibility specifier of op is equal to the visibility of cp.

When a (complex) object package O is an instance of a (complex) class package C, then $O : C$ where O is an *instance level package*.

The next Section has some concluding remarks.

4 Conclusions

We present some results and future work.

4.1 Results

We have shown that class visibility imposed by modules can be insufficient to define properly for all cases when objects may access each other. This is a problem if we consider instantiating the classes of one module to two or more distinct sets of objects where each set of objects collectively represents a distinct thing. For our example we presented a module of classes which represents a bank at the type level. We get multiple sets of objects at the instance level if we want to model more than one bank; one set of objects for each separate bank. With the existing visibility, the objects of one bank can however access the objects instantiated from the public *and* private classes of any other bank. This is not the intention of the modeller as this means that one bank can access all objects of another bank including the objects instantiated from private classes which should be private to the bank if they are only intended for internal use within the other bank. This is a problem which occurs for any OO modelling language which addresses the visibility of objects solely in terms of visibility of classes. Specifically, this is a problem for UML and current programming languages such as C++ and Java. We need not only the concept of private classes, but also the concept of *private objects*.

To solve the above problem we define *instance level packages* for UML. These are special packages of objects. Any given instance level package is an instance of one package of classes. All objects contained by this instance level package are instances of the classes of this package of classes. Furthermore, all private classes are instantiated to private objects. For our bank example, each instance of the bank is placed in a separate instance level package which encapsulates the objects of a bank that are intended solely for local use. These are precisely the objects instantiated from the private classes of the bank. By giving semantics to packages of classes, we can discriminate between the different instances of one package.

Taking into account possible instances of packages of classes allows the modeller to work with objects on a more fine-tuned level. During the development of a model, objects are often used for giving example snapshots of a system under development. Objects are also used to model the existence of specific instances of classes for the final implementation. When not using packages of objects, the grouping of objects is left implicit and only class visibility determines the visibility of objects. The use of instances of packages makes the grouping of objects more explicit and allows a modeler to fine-tune the visibilities of objects belonging to different sets of objects instantiated from the classes of a package.

4.2 Future Work

There are three definite targets for future work using the concepts of instance level packages.

A first candidate is the addition of instance level packages for SOCCA. SOCCA (Specification of Coordinated and Cooperative Activities) is a graphical formalism and associated method for object-oriented modelling which is under active development at Leiden University. SOCCA allows for a precise and detailed specification of the communication and synchronisation between the modelled classes and objects. SOCCA, based on [EG94], has been formalised for the type level in [DGSK$^+$99] using the formal language Z [Spi92]. UML-like packages for SOCCA are currently being formalised in [tHDG$^+$99]. As a followup to [DGSK$^+$], instance level packages for SOCCA will be defined.

The second candidate is to use instance level packages within patterns [GHJV94]. The various patterns are defined in terms of classes and examples are used to show how a pattern should be instantiated. Instance level packages can be used in three ways within the pattern community if the classes of a pattern are contained within a package. First of all, instance level packages can group the objects of a pattern. This gives a clear boundary between the objects belonging to the instance of the pattern and the rest of the model. The instance level package serves as an aggregate for the objects of the pattern. Secondly, instance level packages can serve to distinguish between the objects belonging to different instances of the same pattern. Lastly, instance level packages are still very general. Each class of a package can be instantiated to any number of objects of an instance level package. We can however refine the definition of an instance level package if a pattern requires for a class belonging to the pattern to be instantiated a fixed number of objects. This type of constraint can be incorporated into the definition of the instantiation of the pattern represented by the package of classes. We then only allow for special instance instance level packages with the correct number of objects.

A third topic for future research concerns possible further constraints on the object package structure, and ways to specify such constraints in a model. For example, once that we know that there exist instance level packages of type *Bank*, how do we determine/constrain how many of such instance level packages there are, and which objects belong to which packages?

A solution to a similar problem in the context of graph struturing is illustrated in [BEMW], in which a graph structuring concept, called *graph packages*, is presented. Without going into further details (for which you can refer to [BEMW]), we sketch how the concepts presented there can be transfered to the object package case. The main idea is that instance level packages can be determined by the information contained in the objects they have to group. In our example, *frontoffice* and *accounting* objects which have the same value for attribute *swiftcode* shold be in the same *Bank* package. Their visibility can be determined from the visibility of the corresponding classes with respect tothe *Bank* class package. The number of existing *Bank* instance packages can be determined by the number of different existing swift codes.

More generally, this method, which we call *descriptive* approach, assumes that we extend our modeling language and define some kind of predicates, that allow to describe the structure of instance level packages. Such predicate can use information about object types, about possible links between objects, as well as values contained in object fields. We envision to obtain at least the following advantages from the descriptive approach: Firstly, the package structure is defined in the model in a declarative way, making the reasons for the structuring (e.g. , same swift code) explicitly visibible. Secondly, as far as possible implementations of object packages are concerned, by using predicates we do not need to represent instance level packages in the running system, since it is sufficient to add code to check that access restrictions between objects are not violated. Such additional code for checking access restrictions can be systematically obtained by the declarative definition of the packages. In our example, each time a *frontoffice* object requires to access an *accounting* object, a piece of code can be added, to check whether the *swiftcode* fields in both objects have the same values.

We think the descriptive approach is an attractive solution for modeling, designing, and implementing instance level package structures, and we aim for an implementation in the near future.

References

[BEMW] Giorgio Busatto, Gregor Engels, Katharina Mehner, and Annika Wagner. A framework for adding packages to graph transformation approaches.

Presented at TAGT98, Paderborn, 16-20 November 1998.

[Boo91] Grady Booch. *Object oriented design with applications.* Benjamin/Cummings series in Ada and software engineering. Benjamin/ Cummings Pub. Co., Menlo Park, CA, USA, 1991.

[DGSK$^+$] J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, P.J. 't Hoen, and G. Engels. A formalisation of SOCCA using Z; part 2: the instance level concepts. Manuscript. This document continues the formalisation of the modelling language of the SOCCA OO method in the formal specification language Z as started in [DGSK$^+$99]. It captures the "dynamic" aspects of SOCCA, i.e. the meaning of SOCCA language elements at the instance level (objects in execution).

[DGSK$^+$99] J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, P.J. 't Hoen, and G. Engels. A formalisation of SOCCA using Z; part 1: the type level concepts. Technical Report 1999–03, Leiden Institute of Advanced Computer Science, February 1999. Available on the web as `http://www.wi.leidenuniv.nl/TechRep/1999/tr99-03.ps.gz`.

[EG94] Gregor Engels and Luuk P.J. Groenewegen. SOCCA: Specifications of coordinated and cooperative activities. In A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 71–102. Research Studies Press Ltd. / John Wiley & Sons Inc., 1994. Taunton 1994.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley professsional computing series. Addison-Wesley, 1994.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, Inc., Englewood Cliffs, 1 edition, 1991.

[GM95] James Gosling and Henry McGilton. *The Java Language Environment: a white paper.* Sun Microsystems, oct 1995.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, Inc., New York, N.Y., second edition, 1997.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[RTF99] Uml specification v. 1.3. Technical report, OMG Unified Modeling Language Revision Task Force (UML RTF), June 25 1999.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[tHDG$^+$99] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, and G. Engels. SOCCA extended with uml like packages. Technical report, liacs, April 1999.

[UML97] Unified modeling language 1.0. Technical report, Rational Software Corporation, January 13 1997.