

Accessibility & Helper Types

Jan Dockx

Report CW 341, June 2002



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Accessibility & Helper Types

Jan Dockx

Report CW 341, June 2002

Department of Computer Science, K.U.Leuven

Abstract

We often encounter a need for formal specifications aimed toward users with more than public access. Here we embark on a search for a clear semantics for such specifications, by looking at the specifications of different accessibility levels as different types related through inheritance. This paper reports the first results of this approach.

Keywords : BISL, behavioral subtyping, object oriented programming, Java, accessibility, scope

CR Subject Classification : D.1.5, D.2.4.

Accessibility & Helper Types

Jan Dockx

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
3001 Leuven, Belgium
Jan.Dockx@cs.kuleuven.ac.be
<http://www.cs.kuleuven.ac.be/>

Version 1, Monday, 8 April 2002

Position paper for Workshop on Formal Techniques for Java-like Programs - FTJJP'2002, ECOOP 2002, June 11, 2002, Malaga, Spain

Version 2, Wednesday, 29 April 2002

CW Report; short position paper is published in the ECOOP 2002 Workshop Reader

Abstract. We often encounter a need for formal specifications aimed toward users with more than public access. Here we embark on a search for a clear semantics for such specifications, by looking at the specifications of different accessibility levels as different types related through inheritance. This paper reports the first results of this approach.

Introduction

More and more often we are confronted with the need to take formal specification beyond the public interface of a class. It has become clear to us that this is less trivial than can be assumed. A recent discussion on the applicability of type invariants to private methods in the JML mailing list [Leavens] triggered the research of which we want to report the initial findings here. The main idea is that a frame of reference is in order to clean up the thinking about protected, package and private specifications. By introducing virtual (in the English, not the C++ sense, although the C++ sense also applies) types for the different levels of accessibility, we are able to reason meaningful semantics for such less accessible specifications.

Java terminology will be used throughout the text, but we believe that this discussion is equally applicable to e.g., C++ or Eiffel. We have a BISL like the one presented by Meyer [Meyer 1991][Meyer 1997] in the backs of our head, but we believe the discussion applies equally to a model based specification language such as JML [Leavens, Baker & Ruby 2001].

Lead

Public users only see public methods. The specification of public methods can therefore only refer to other public methods and properties. If the specification of a public method would refer to protected, package accessible (or friend accessible), or private members, either encapsulation is broken or the public users cannot understand the specification, because it refers to unknown concepts. Users with more access, such as protected users, users in the same package, or users in the same compilation unit (private users), have access to more members. They see, respectively, protected, package accessible and private methods and properties, next to the public members.

This means in the first place that they can use the extra methods they see, in contrast to public users. Secondly, the users with more access will understand a specification that refers to more restricted members, even if it is the specification of a less restricted method. Users in the same package, e.g., would be able to interpret a specification of a public method that refers to package accessible methods. From experience, we know this is often desired. In the package accessible part, more properties exist than in the public part, and the effect of a public method on the package accessible properties often must be known by package users for them to be able to do their job. Often even more pressing is the need to know the preconditions that apply to package accessible properties. This immediately leads to the idea to extend the specification of methods with more restricted access with a specification aimed at the users with less restricted access. In other words, we would like to

- add a protected, package accessible, and private specification to public methods, next to the existing public specification;
- add a package accessible and private specification to protected methods, next to the existing protected specification;
- add a private specification to package accessible methods, next to the existing package accessible specification.

The extra postconditions need to be extensions of the existing postconditions: they can say more, but can never contradict the existing postconditions. The extra preconditions also cannot contradict the existing preconditions: the method may be applicable in more circumstances for a less restricted user, but never in less.

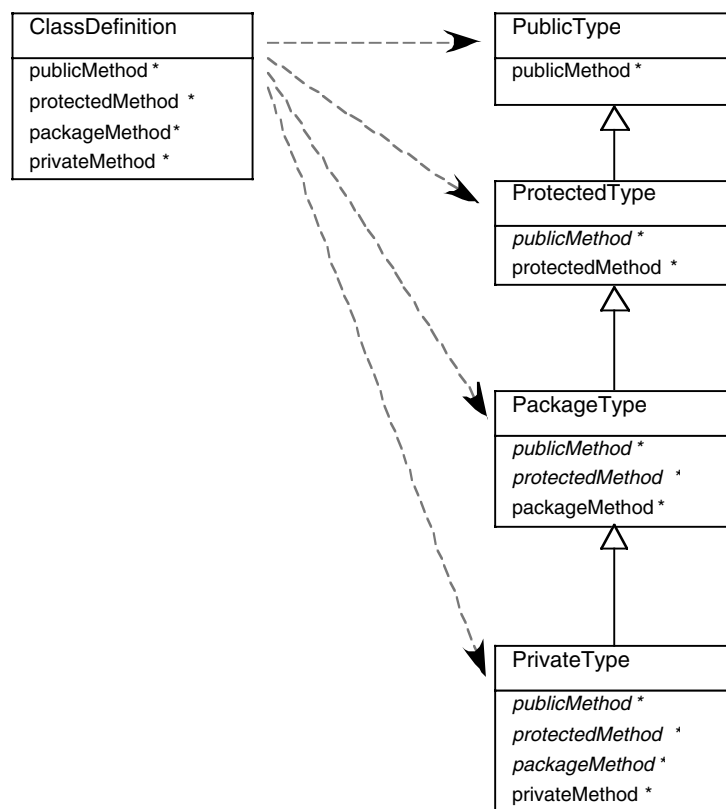
The latter sounds very much like *behavioral subtyping* [Liskov, Wing 1994]. It suggests that we should look at the more restricted parts of a class as *subtypes* of the less restricted parts. The intention of this paper is to examine where such an approach leads us.

Accessibility Types

In this view, a Java class would introduce not 1, but 4 types implicitly: a public type, a protected type, a package accessible type, and a private type. Java interfaces need not be discussed, since they can only offer methods of the same accessibility as the interface itself. C++ class would introduce a public type, a protected type, a private type, and an implementation type, and several different types for each combination of friend declarations. The impact on C++ will not be discussed further.

The *accessibility types* are related through subtyping, as shown in the figure.

ProtectedType inherits members defined in PublicType, with their specification, PackageType inherits members available in ProtectedType and so on. Subtypes can add new members, and redefine the contract of existing members in compliance with behavioral subtyping.



Implementations with different Accessibility and In-Module Overwriting

In this view, it is possible to implement some methods in a more general accessibility type than the final PrivateType. E.g., a public method, which is defined abstract in PublicType, could be implemented in there. Obviously, such implementation can only use members defined in PublicType. Other public methods might only get an implementation in more specific implementation types, ultimately in PrivateType. Such early implementations represent a much-used form of encapsulation within the implementation of a class. These methods are independent of representation and implementation choices made in less restricted implementation types, which results less in-module dependency and higher in-module stability.

This view puts forward a possibly interesting new concept: *in-module overwriting*. A more accessible method, which can be implemented completely in terms of other methods of the same or broader accessibility, is often implemented nevertheless in terms of less accessible methods or properties for

reasons of efficiency. Algorithms can often be made more performant if they can use otherwise hidden representation properties. The cost is lower in-module stability. The same is often true if a recursive algorithm needs to be replaced by a more efficient iterative alternative. The above view suggests that the more efficient implementation can be considered as overwriting the more general implementation. If later lower-level types change, first of all it is clear which implementation are touched by this change. Secondly, in-module stability is ensured, because the higher level, less efficient, though correct implementation is available as a fallback.

Another possible benefit of this separation in different implementation types is a clear separation in the implementation of the code responsible for the specifications in the different accessibility types. Suppose a public method has extra, behavioral subtyping compliant specifications in the protected, package accessible and private accessibility types. The public implementation type could possibly provide an implementation for the public specification, while the protected, package and private implementation types extend the implementation to do what the extra specification requires. This implies the use of a `super`-call between different implementation types. On the other hand, we know from regular object oriented programming and experiences in aspect oriented programming that a simple `super`-call often does not lead to the desired result. Most often a complete rewrite is needed to fulfill both the original and extra contract.

This way of working could be applied by hand in existing programming languages through a creative use of documentation.

Constructors

Constructors pose a particular problem when using accessibility types. Constructors cannot be sensibly defined in the accessibility types, since they are all abstract. Only the final private type can be instantiated. On the other hand, a method defined in the final private type is not visible to any user, and thus cannot be called. Furthermore, constructors are not inherited in most object-oriented languages.

In practice, constructors can be defined with any level of accessibility, and depending on the actual situation, it is often possible to implement them in `PublicType`, `ProtectedType`, or `PackageType`, and certainly in `PrivateType`.

As a solution we propose to consider constructors as instance initialization methods. They behave exactly like regular instance methods, including inheritance, within the accessibility and implementation type hierarchy, with the exception that public or package users cannot call them directly, and protected and private users can only call them at the start of another constructor. We also consider initialization code to be part of the constructor implementation. This makes that the state of the instance is exactly known when execution of the constructor starts. Furthermore, type invariants do not necessarily apply at the start of constructor execution.

Type Invariants & Helper Methods

Normally we expect all methods to respect all type invariants when they hand over control of execution. When the new entity in control of execution looks at this instance, it expects to see a stable state, i.e., a state conformant to the type invariants. Handing over control happens at the end of a method, but also when the method implementation calls other methods to do part of the work. Working with accessibility and implementation types strongly suggests that all methods should respect all possible type invariants. Type invariants can be introduced in `PublicType`, `ProtectedType`, `PackageType` and `PrivateType`. If these types are required to be behavioral subtyping compliant, type invariants in more accessible subtypes can only strengthen type invariants introduced in less accessible types. Invariants in `PrivateType` are often called representation invariants, describing representation choices hidden from external users, which all implementations must respect.

On the other hand, our implementations frequently use *helper methods* that do not respect type invariants. When implementing a complex algorithm, it is considered good practice to isolate consistent parts of the algorithm in separate methods, with their own contract, to do part of the work for you. They act as lemmas, in that they encapsulate part of the complexity of the algorithm, which can be reasoned about locally. Reasoning about the main algorithm becomes easier. This is one of the earliest forms of encapsulation introduced in programming, often referred to as *abstraction*. These methods often are only relevant to the one method in question, and are written for a specific task, part of the larger task of the main algorithm. They are defined with private accessibility, and they often do not adhere to public, protected, package or private type invariants, nor do they depend on them being true initially. That is

often impossible, since they are conceived to do *part* of the work of the main algorithm, which as a whole will satisfy type invariants as part of its contract, and possibly depend on the type invariants being true initially. Such helper methods clearly violate behavioral subtyping.

A first idea for a solution to this problem is to exempt private methods from adhering to type invariants. This seems logical, since it solves the problem, and only influences the private users of the class. In essence, this moves `PrivateType` from the bottom of the accessibility hierarchy to a position next to the other accessibility types, possibly inheriting from a yet not defined empty super type. This clearly is no proper solution, since there has to be a bottom type to instantiate in any case, and the problem would move there. The result would be a hierarchy of all abstract types, with no final implementation. In other words: even if type invariants are removed from the specification of methods, their implementation still needs to satisfy them.

After some consideration, the solution seems simple. The heart of the matter is that these helper methods *are not part of the types* introduced by the class. Any other solution will result in a violation of behavioral subtyping. We propose to add *helper types* next to the accessibility types. Helper methods can be defined here, together with the data structures they work on. It is impossible to merge the helper types into the actual class and its accessibility types via inheritance in any way. Finally, the bottom concrete type will feature all type invariants introduced in any of the accessibility types, and if the methods of the helper types are mixed in, they will need to be overwritten to comply with them to keep everything behavioral subtype compliant. The solution is to use the helper types via delegation. E.g., if you would like to introduce private helper methods to do house holding of an array data structure, you should create a helper type for that data structure and its house holding routines, possibly without any type invariants. The private type then can introduce an instance variable that refers an instance of the helper type, and gets access to the helper methods in that way. Different helper types might be introduced for separate issues, and a graph of helper instances can be used where issues overlap.

It might be worthwhile to introduce these private helper types explicitly. Java offers premium support for helper types via nested classes. On the other hand, specification syntax might be introduced to make the helper types virtually, e.g. a label to be applied to the methods and variables that belong to the helper type (`@helper myDataStructureHelperType`).

Subclasses

The second figure shows how the accessibility types would behave when a subclass is defined outside the package. The subclass can add methods of any accessibility and can strengthen the contract of existing methods. Public users of the subclass see both the `PublicType` and the `PublicSubType`; protected users see the both public types plus `ProtectedType` and the extended `ProtectedSubType`. Package users only see `PackageSubType` and private users only see `PrivateSubType`. `PublicSubType` thus extends `PublicType` and `ProtectedSubType` extends `ProtectedType`. The package and private accessibility types are not related through inheritance. When the subclass would be defined in the same package, `PackageSubType` would extend `PackageType`.

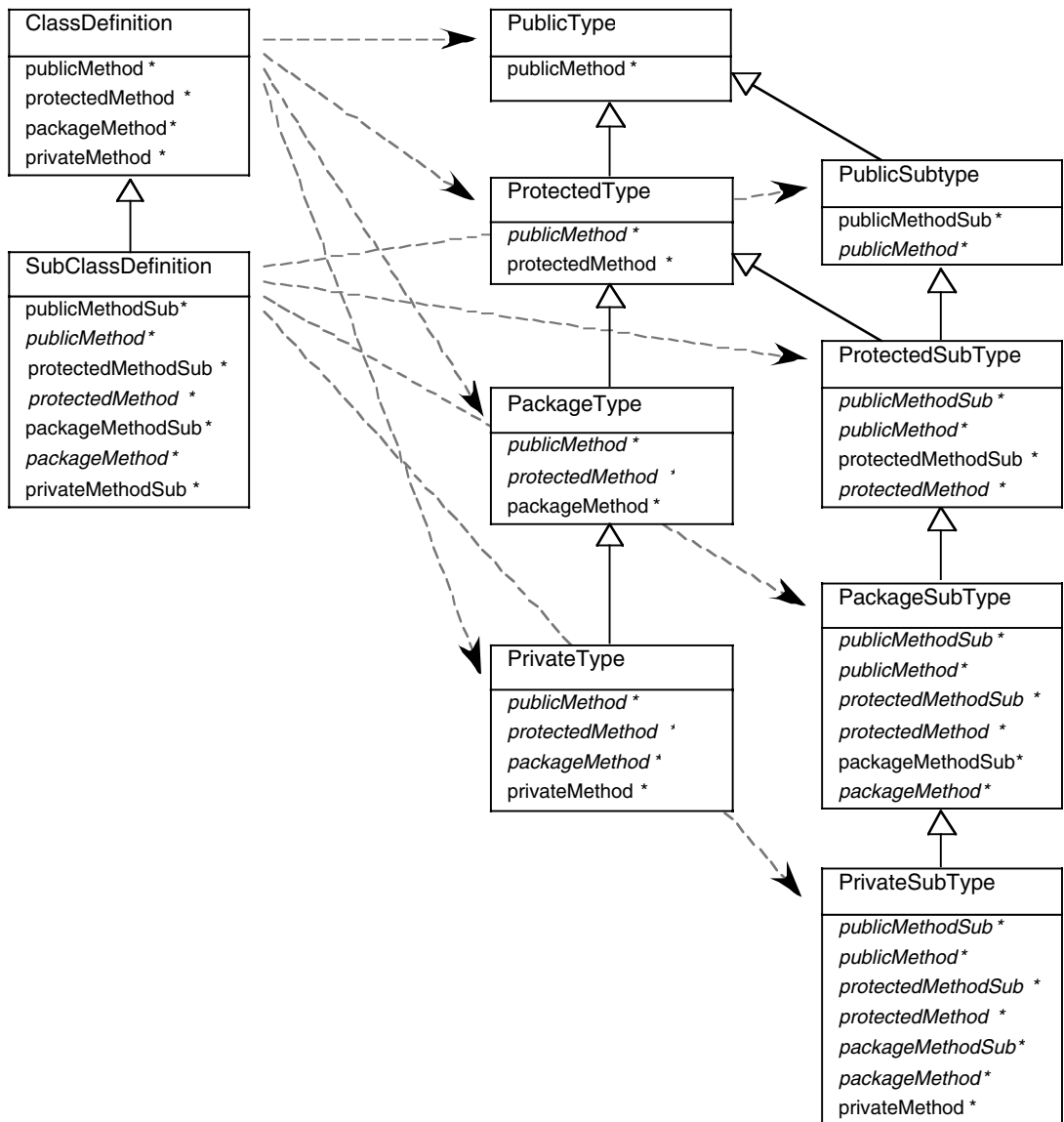
We encounter multiple specification inheritance here. A public method defined in the superclass can have a stronger specification in `ProtectedType`, and also in `PublicSubType`. It is clear that, because of behavioral subtyping, these 2 contracts need to agree. An implementation in `ProtectedSubType` or lower needs to implement both contracts.

Type Invariants

The immediate result is that method implementations in the subclass cannot see package accessible or private type invariants. They won't depend on them, but they also cannot be required to uphold them. In other words, the package and private implementation of the super class needs to take care of them once and for all.

Overwriting Protected Methods

One of the most intriguing aspects about specifications with broader accessibility is what happens with protected specifications in the subclass. The protected methods, or protected specifications of public methods of the superclass are presented to implementers of the subclass in 2 roles. On the one hand, these implementers are the protected users we talked about earlier. They can call the public and protected methods defined in the superclass in their own implementations. When doing this, the implementers



depend on the contract specified for these methods in the superclass, or actually in `PublicType` and `ProtectedType`. On the other hand, they can also strengthen their specification. The subclass can extend the specifications of `PublicType` in `PublicSubType` and `ProtectedSubType`, and it can extend specifications of `ProtectedType` in `ProtectedSubType`. The basis of the contract paradigm is that these 2 roles are completely separated, and yet here they come together in 1 entity. This gives rise to interesting conflicts, which have surfaced often in our work in object-oriented frameworks.

It happens quite often that the specification of a public method (`PublicType`) is not deterministic. Subclasses are required to fill out the missing bits. This is the heart of object-oriented software development. If such methods are abstract, there are little problems with it in the context discussed here. Quite often though, the superclass offers a *default implementation* that can be used by a subclass, or overwritten, as desired. Take a method for which specified that it can throw a `NotSupportedException`. Subclasses can choose to actually throw the exception, but they can also choose to do something meaningful described by the postcondition of the method. The implementer of the superclass might decide to provide a default implementation that actually throws the exception, taking some of the burden of the subclasses that desire that behavior. The problem arises when we want to tell the implementers of the subclass about this default implementation. In essence, we want to tell them that the method is implemented in the superclass with a postcondition `false`. Normal behavior can never make that postcondition true, so an exception must be thrown. Obviously, this specification cannot be made public. It is to be communicated only to the implementers of the subclass. This postcondition can strengthen the original, non-deterministic postcondition in `ProtectedType`. Now the implementer of the protected type knows what the implementation actually does. The protected postcondition is no less than the strongest postcondition of the actual implementation. But, because this specification also is inherited into `ProtectedSubType`, the implementation in the subclass also needs to uphold it. We have effectively specified that all subclasses need to throw the exception. That was not the intention.

If we decide to apply the accessibility types approach, we can only conclude that this practice is not allowed in good object oriented programs. If you want to offer default behavior to subclasses, you can provide a separate protected method, but the original method should be abstract. Subclasses then can use the default behavior by calling the second method in their implementation of the original method. The second method is in practice final, as its specification will be, or will be very close to, the weakest precondition/strongest postcondition contract. This result is not exactly what we hoped for.

Another kind of default implementation is a public method in an abstract superclass, which has an implementation that only partially reaches its contract. The intention is for subclasses to call the `super-method`, and do whatever else is needed to complete the contract before or after that call. When we use the accessibility types approach, this practice is forbidden, and rightfully so. What the implementation actually does should be communicated to the protected users, but this cannot be specified in `ProtectedType`. It does less than the contract in `PublicType` requires, and that would violate behavioral subtyping. We are happy with this result, because this technique has proven difficult to handle.

Note that the use of template and hook methods [Pree 1995]. The template method specifies a behavior in `PublicType`, and will call protected hook methods in its implementation. These methods have a `ProtectedType` contract, and together they can be proven to reach the public contract of the template method. The hook methods need to be abstract, as shown higher, for the subclasses to be implemented, potentially with the aid of other protected methods offered by the superclass.

The conclusion of this study actually is, that when the approach of accessibility types is used, there is no way to communicate the actual implementation of the method in the superclass to the implementers of the protected type. If the subclass needs to strengthen the contract, be it the public or protected specification, of the `super-method`, it needs to do so from scratch, and it cannot depend on the implementation of the `super-method` in any way.

References

[Leavens]

Gary T. Leavens: **JML**
<http://www.cs.iastate.edu/~leavens/JML.html>

[Leavens, Baker & Ruby 2001]

Gary T. Leavens, Albert L. Baker & Clyde Ruby; **Preliminary Design of JML: A Behavioral Interface Specification Language for Java**
<ftp://ftp.cs.iastate.edu/pub/leavens/JML/prelimdesign.pdf>

[Liskov, Wing 1994]

Barbara H. Liskov & Jeannette M. Wing: **A Behavioral Notion of Subtyping**
ACM Transactions on Programming Languages and Systems, Vol. 16, Nr. 6; November 1994; p. 1811-1841

[Meyer 1991]

Bertrand Meyer: **Design by Contract**
Advances in Object-Oriented Software Engineering, Ed. D. Mandrioli, Bertrand Meyer; Prentice Hall; Englewood Cliffs, N.J.; 1991; p. 1-50

[Meyer 1997]

Bertrand Meyer: **Object Oriented Software Construction**; 2nd Edition
Prentice Hall; Upper Saddle River, NJ; 1997; 1254 pages; ISBN 0-13-629155-4

[Pree 1995]

Wolfgang Pree; **Design Patterns for Object-Oriented Software Development**
Addison-Wesley Publishing Company; Wokingham, England; 1995; ISBN 0-201-42294-8