9-2006

Information Hiding and Visibility in Interface Specifications

Gary T. Leavens
Iowa State University

Peter Müller
ETH

Information Hiding and Visibility in Interface Specifications

Abstract
Information hiding controls which parts of a class are visible to non-privileged and privileged clients (e.g., subclasses). This affects detailed design specifications in two ways. First, specifications should not expose hidden class members. As noted in previous work, this is important because such hidden members are not meaningful to all clients. But it also allows changes to hidden implementation details without invalidating correctness proofs for client code, which is important for maintaining verified programs. Second, to enable sound modular reasoning, certain specifications must be visible to clients. We present rules for information hiding in specifications for Java-like languages, and demonstrate their application to the specification language JML. These rules restrict proof obligations to only mention visible class members, but retain soundness. This allows maintenance of implementations and their specifications without affecting client reasoning.

Keywords
Information hiding, visibility, behavioral interface specification language, proof obligation, public, protected, private, client, JML language

Disciplines
Software Engineering | Theory and Algorithms

Information Hiding and Visibility in Interface Specifications

Gary T. Leavens and Peter Müller

TR #06-28
September 2006

Keywords: Information hiding, visibility, behavioral interface specification language, proof obligation, public, protected, private, client, JML language.

2006 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, Spec#, Eiffel, JML; D.3.3 [*Programming Languages*] Language Constructs and Features — Modules, Packages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques.

Submitted for publication

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Information Hiding and Visibility in Interface Specifications

Gary T. Leavens*
Iowa State University
Ames, Iowa, USA
leavens@cs.iastate.edu

Peter Müller†
ETH Zurich
Switzerland
peter.mueller@inf.ethz.ch

Abstract

Information hiding controls which parts of a class are visible to non-privileged and privileged clients (e.g., subclasses). This affects detailed design specifications in two ways. First, specifications should not expose hidden class members. As noted in previous work, this is important because such hidden members are not meaningful to all clients. But it also allows changes to hidden implementation details without invalidating correctness proofs for client code, which is important for maintaining verified programs. Second, to enable sound modular reasoning, certain specifications must be visible to clients. We present rules for information hiding in specifications for Java-like languages, and demonstrate their application to the specification language JML. These rules restrict proof obligations to only mention visible class members, but retain soundness. This allows maintenance of implementations and their specifications without affecting client reasoning.

1 Introduction

When following information hiding, clients (including subclasses) of each class are provided with the information they need to use that class, but nothing more [28]. This aids maintenance because hidden implementation details can be changed without affecting clients. However, information hiding and its benefits apply not only to code but also to other artifacts, such as documentation and specifications.

In this paper, we focus on formal interface specifications and correctness proofs. Formal interface specifications include contracts written in Eiffel [22], the Java Modeling Language (JML) [14], and Spec# [3]. We use JML examples for concreteness, but the rules we present can also be applied to Eiffel and Spec#. We mainly discuss JML since its syntax has visibility modifiers for specification constructs, such as invariants and method specification cases. These modifiers allow one to specify a class's public (non-privileged client), protected (subclass), package (friend), and private (implementation) interfaces [10, 13, 29, 30].

Our contribution is a set of rules for the modular use of visibility modifiers in specifications. Formalization allows us to precisely describe the subtle interactions between programs, specifications, and proofs and to prove soundness.

Our rules could also be applied to similar artifacts. For example, they could be applied to the weak (incomplete) specifications embodied in unit test cases and to the informal specifications embodied in documentation. Like formal specifications they could also be specialized for different visibility levels. For example, a unit test case could be marked as public, which would imply that changes to hidden implementation details would not affect its type correctness or meaning. Hence, it would not have to be changed when hidden details change. Similarly, a class could have documentation marked as protected, which describes how its methods affect its protected members.

Information hiding affects specifications in two ways. First, specifications should not expose hidden implementation details. Such details cannot be fully understood by all clients [23]. Also they should not be used in a client's correctness proof, since otherwise the proof would be invalidated when they change. For example, suppose method `add` of a class `BoundedList` has a public precondition `count < capacity`, where `count` and `capacity` are protected fields. Then non-privileged clients do not know what this precondition means exactly; for instance, they do not know whether `count` is the number of elements in the list (counting from one) or an array index (counting from zero). Such details are hidden from clients to enhance maintainability, which includes maintainability of correctness proofs.

Second, to enable sound modular reasoning, certain specifications must be visible to clients. For instance, specifications of virtual (overrideable) methods must be visible to overriding subclass methods, otherwise the overriding method cannot respect behavioral subtyping [1, 5, 15, 21].

*Supported in part by the US NSF under grant CCF-0429567.

†Funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

Another example of the need to make some specifications visible is related to object invariants. Suppose a class `BoundedList` contains a private invariant to express that its protected `capacity` field is non-negative. Suppose further that an analysis (e.g., code review or verification) has shown that the program maintains this invariant. Now if the implementor of `BoundedList` strengthens the private invariant, for instance, to require a capacity of at least 1000 elements, the analysis would have to be repeated to show that the strengthened invariant is still maintained. The analysis must be repeated because methods in `BoundedList`'s package and in subclasses of `BoundedList` might break the invariant by assigning to the protected `capacity` field. Such a repeated analysis illustrates a maintenance (and modular reasoning) problem. Making the invariant protected would make it visible to clients that might break it, clarifying what proofs might break when it changes. Our rules enforce such visibility, for example by not allowing `BoundedList`'s private invariant to mention the protected `capacity` field.

In sum, there are two problems. The first is a maintenance problem, which motivates a lower bound on the visibility of the class members mentioned in a specification: they must be *at least as* visible as the specification itself. The second is a problem of sound modular reasoning, which motivates an upper bound on the visibility of the class members mentioned in certain specifications: they have to be *at most as* visible as the specification itself to allow hidden specifications to be changed without affecting client reasoning. Existing work [3, 14, 23] has focused on the first problem. The recognition and solution of the second problem are our main contributions.

Maintenance is greatly simplified if the effects of changing one part of a program or its specification are limited to a known set of classes. For example when refactoring, tools have to perform a dependency analysis for each change. Efficiency of such an analysis is aided by keeping some specifications hidden. For example, one should keep invariants that describe implementation hidden so that when either is changed, clients are not affected. For this reason, not all specifications should be public—some should be hidden.

We use the notion of visibility to determine the set of classes potentially affected by changing a specification construct or a type member such as a field or a method.[1]

Definition 1 (Visibility) *A type T* is visible in *a type S if (1) $T = S$, (2) T and S are in the same module (package or namespace), or (3) S explicitly imports T and T is public. A specification construct or a class or interface member m declared in type T* is visible in *a type S if T is visible in S and the visibility modifier of m grants visibility to S.*

For example in Java a default (package) access field f of class T is visible in a type S if and only if T and S are in the same package. Therefore, only types in T's package are potentially affected by changes to f (e.g., renaming it).

Our definition of visibility is more restrictive than Java's [8] since we require an explicit import, whereas Java permits fully-qualified type names. In particular, a public member of a type T is not visible in all types of the program but only in those that directly or indirectly import T. Our more restrictive definition improves maintainability because one does not have to inspect the full text of each type S to determine whether it uses T. However, the rules we present are also sound for Java's notion of visibility.

Outline. The rest of this paper is structured as follows. Sec. 2 addresses the first information hiding problem described above. The following sections address the second problem for method specifications (Sec. 3), model fields and data groups (Sec. 4), and invariants (Sec. 5). In each of these sections, we present general rules that enable sound reasoning and efficient proof maintenance. We also show how to enforce the rules in Java and JML.

2 Information Hiding

This section addresses the first information hiding problem described above: a specification should not expose implementation details that should be hidden. A specification construct that mentions a member f *exposes* f if the specification construct is visible to a class C but f is not.

Rule 1 (Information Hiding) *Every (class or interface) member mentioned in a specification construct must be visible wherever the specification construct is visible.*

We illustrate this rule using the JML specification in Fig. 1 of two-dimensional Cartesian points. The coordinates of a point are stored in the protected fields `_x` and `_y`. The private fields `_oldX` and `_oldY` are used in the `undo` method. The invariants restrict points to the upper right quadrant. The first invariant is protected and the second is private.

Following Rule 1, the protected invariant of class `Point` must not mention the private fields `_oldX` and `_oldY`, because these fields are not visible to subclasses of `Point`, unlike the protected invariant itself. (We will explain in Sec. 5 that it is also not admissible to mention protected fields in private invariants. Therefore, properties of the private fields must be expressed in a separate, private invariant.)

However, Rule 1 is violated by `move`'s specification. This method's specification consists of two specification cases with different visibility modifiers. The specification cases are separated by the keyword **`also`**. The assignable clause in the protected specification case violates Rule 1 because it exposes the private fields `_oldX` and `_oldY`. The assignable

[1] This paper ignores the possibility of having multiple types in a single compilation unit and also does not consider nested classes.

```
public class Point {
  protected int _x, _y;
  private int _oldX, _oldY;  // for undo method
  //@ protected invariant _x >= 0 && _y >= 0;
  //@ private invariant _oldX >= 0 && _oldY >= 0;

  /*@   protected normal_behavior
    @    requires _x + dx >= 0 && _y + dy >= 0;
    @    assignable _x, _y;
    @    assignable _oldX, _oldY;  // illegal
    @    ensures _x == \old(_x + dx)
    @         && _y == \old(_y + dy);
    @ also
    @   private normal_behavior
    @    requires _x + dx >= 0 && _y + dy >= 0;
    @    assignable _x, _y;
    @    assignable _oldX, _oldY;  // legal
    @    ensures _oldX == \old(_x)
    @         && _oldY == \old(_y);
    @*/
  public void move(int dx, int dy) {
    _oldX = _x;
    _oldY = _y;
    _x += dx;
    _y += dy;
  }

  /*@ private normal_behavior
    @  assignable _x, _y;
    @  ensures _x == \old(_oldX)
    @       && _y == \old(_oldY);
    @*/
  public void undo() {
    _x = _oldX;
    _y = _oldY;
  }

  public void testUndo(Point p) {
    int initialX = p._x;
    p.move(1, 1);
    p.undo();
    assert initialX == p._x;
  }
  // Other methods and constructors omitted.
}
```

Figure 1. A JML specification for Cartesian points. Annotation comments start with an at-sign (@), and at-signs at the beginning of lines are ignored. Specification cases for a method, which start with a visibility modifier and `normal_behavior`, appear before the method's header. Preconditions are introduced by the keyword `requires`, frame axioms by `assignable`, and postconditions by `ensures`. Each specification case must be obeyed when its precondition holds.

clause in a specification case has to describe all locations that are potentially assigned by the method when that specification case's precondition holds. But mentioning such private fields in the assignable clause of a non-private specification case exposes these fields. This problem can be solved using data abstraction, as we explain in Sec. 4.

Rule 1 states a general requirement in terms of visibility. This rule is carefully written using "visible wherever" because enforcement is not as simple as it might seem. For example, in Java Rule 1 is usually satisfied if the members mentioned in a specification construct have the same or a less restrictive visibility modifier than the construct itself, for instance, when both the members and the specification construct are private. However, this is not always the case if the member has protected visibility.

Careful enforcement is needed in two kinds of examples. First, suppose a protected invariant of class C mentions a protected field f declared in a superclass of C. Then the invariant is visible to all classes in C's package, but f is (generally) not. Second, suppose a protected invariant of class C mentions a protected field f declared in another class D of C's package. Then the invariant is visible to all subclasses of C, but f is not. The same lack of transitive visibility can be caused by **`protected internal`** declarations in C# and Eiffel's selective availability of feature exports [6].

3 Method Specifications

This section begins our discussion of the second information hiding problem described in the introduction. We explain the visibility requirements that enable sound modular reasoning, starting with method specifications.

3.1 Visible Preconditions

Preconditions of methods lead to proof obligations for clients (i.e., callers) of the method, but clients cannot be required to satisfy hidden preconditions. For instance, since `move`'s protected precondition mentions the protected fields `_x` and `_y`, a client cannot use a conditional to check the precondition before calling the method. Sound modular reasoning prohibits a method implementation from relying on a precondition that is not satisfied by all clients.

In a language like Eiffel or Spec#, each method has one precondition, which defaults to **`true`**. Unsoundness in such a language is prevented by the following rule [6, Sec. 8.9].

Rule 2 (Visible Preconditions) *Each method's precondition must either be* **`true`** *or have the same visibility as the method itself.*

In JML, a method specification may have more than one specification case, each with its own visibility and precon-

dition [12, 13, 31, 32]. Each specification case is a self-contained contract; that case's contract must be obeyed whenever its precondition holds. This allows method specification to be broken up into more easily understood parts.

A method's caller must establish the precondition of some visible specification case (and cannot use non-visible cases). Thus the effective precondition for a client is the disjunction of the preconditions in the specification cases visible to that client. If none are visible, the effective precondition is **true**. Thus the meaning of a JML method specification as a whole automatically satisfies Rule 2. It follows that a specification case whose visibility is less than that of the method itself can only describe a special case in the method's behavior, and cannot limit calls to the method. A method implementation may assume only the effective precondition for the method's own visibility, since that is all that the least privileged clients must establish.

In the specification of `move` (Fig. 1), both specification cases, and hence their preconditions, are not public and therefore are not visible at all calls to `move`. Thus, the effective precondition of `move` for non-privileged clients (that are neither subtypes of `Point` nor in the same package) is **true**. This effective precondition is trivially satisfied by non-privileged clients. However, non-privileged clients are not able to use the postconditions of the hidden specification cases, which are not visible to them.

A correctness proof for `move` may assume only its effective precondition, **true**. Since this precondition does not prohibit calls in which `_x + dx < 0` or `_y + dy < 0`, it is not sufficient to reestablish the invariants at the end of the method. An attempt to verify `move` would find this problem. While not needed for soundness, such problems can be avoided by writing at least one specification case with the same visibility as the method. Another guideline is that one should make sure that the method's effective precondition is at least as strong as the disjunction of the hidden preconditions. The specification in Sec. 4 follows these guidelines.

Hidden specification cases are useful for specifying implementation details for privileged clients [29]. In particular, protected specification cases are useful for verifying calls to overridden methods (super-calls). The benefit for maintenance is the ability to change implementation details and the corresponding hidden specification cases without affecting the correctness proofs for non-privileged clients.

3.2 Behavioral Subtyping

The general idea of behavioral subtyping [1, 21] is the requirement that an overriding method has to obey the specifications of the overridden methods [5, 12, 15]. It allows one to reason about dynamically-bound methods in a modular way. In this subsection, we clarify the requirements of behavioral subtyping in the presence of visibility modifiers.

```
public class ScreenPoint extends Point {

  /*@ also
    @ protected normal_behavior
    @  requires   _x + dx < 0;
    @  assignable _x, _y;
    @  ensures    _x == 0;
    @ also
    @ protected normal_behavior
    @  requires   _y + dy < 0;
    @  assignable _x, _y;
    @  ensures    _y == 0;
    @*/
  public void move(int dx, int dy) {
    if(_x + dx >= 0)  _x += dx;
    else              _x = 0;
    if(_y + dy >= 0)  _y += dy;
    else              _y = 0;
  }

  // constructors omitted.
}
```

Figure 2. `Point`**'s subclass** `ScreenPoint` **overrides the inherited** `move` **method and provides additional specification cases that describe how the method behaves for arguments that do not satisfy the precondition of the inherited protected specification case.**

Overriding subtype methods cannot be required to obey supertype specifications that are not visible in the subtype.

Rule 3 (Behavioral Subtyping) *Every overriding method $S.m$ inherits those specification cases of each overridden supertype method $T.m$ that are visible to type S. The implementation of $S.m$ must satisfy the specification cases for m given in S as well as all inherited specification cases.*

For instance, method `move` of class `ScreenPoint` in Fig. 2 cannot see the private specification case of the overridden `move` method (Fig. 1). Therefore, it might not obey this specification case. That actually happens in our example, as the overriding method does not update the `_oldX` and `_oldY` fields. So callers of the `move` method must not use its private specification case unless they know what implementation will be bound dynamically to the call.

This is illustrated by method `testUndo` in Fig. 1, where `p` might be an instance of an arbitrary subtype of `Point`. If `p` is a `ScreenPoint` object, then the call to `move` would not save the value of `_x`. Then the following call to `undo` would not restore the value of `_x` correctly, and so the assertion would fail. This shows that it is not sound to use a private specification case to reason about dynamically-bound calls even if the specification case is visible at the call site.

If method `testUndo` would create the object `p` using `Point p = new Point()` instead of taking it as parameter, then we would know which implementation the call of `move` is bound to and it would, therefore, be sound to use its private specification to reason about the call. This observation is generalized by the following lemma. For simplicity, we ignore invariants here, but we discuss them in Sec. 5.

Lemma 1 *If a method $T.m$ satisfies Rule 3 and $T.m$'s implementation and the implementations of overriding subtype methods satisfy their specifications, then the following properties hold for each call $x.m(\ldots)$, where $P_i^{T.m}$ and $Q_i^{T.m}$ denote the precondition and postcondition of $T.m$'s i-th specification case including specifications inherited from supertypes of T according to Rule 3.*

(i) If the call $x.m(\ldots)$ is dynamically bound, the compile-time type of x is T, and a precondition $P_i^{T.m}$ that is visible to both the caller and subclasses of T holds in the pre-state of the call, then the corresponding postcondition $Q_i^{T.m}$ holds in the post-state of the call.

(ii) If the call $x.m(\ldots)$ is statically or dynamically bound to an implementation $S.m$ and a precondition $P_i^{S.m}$ that is visible to the caller holds in the pre-state of the call then the corresponding postcondition $Q_i^{S.m}$ holds in the post-state of the call.

Proof Sketch: Property (i) follows from the fact that the call is bound to an implementation $S.m$, where S is a subtype of T. By Rule 3 $S.m$ inherits the specification cases of $T.m$ that are visible to S. So the specification case $(P_i^{T.m}, Q_i^{T.m})$ is inherited as $(P_j^{S.m}, Q_j^{S.m})$. By assumption $T.m$'s implementation and the implementation of $S.m$ satisfy their specifications. Thus, since $P_i^{T.m}$ $(= P_j^{S.m})$ holds in the pre-state of the call, $S.m$ establishes $Q_i^{T.m}$ $(= Q_j^{S.m})$ in the post-state.

Property (ii) is a trivial consequence of the fact that $S.m$'s implementation satisfies its specification. ∎

For JML, this lemma generally means that calls to a virtual method can only use that method's public and protected specification cases. Private and default access specification cases are useful to reason about calls to non-virtual (private, final, or static) methods, constructors, and also virtual methods if the caller knows the exact type of the receiver.

Rule 3 facilitates maintenance. Private and default access specification cases of a method can be changed without re-verifying all overrides in subclasses.

4 Model Fields and Data Groups

In this section, we explain how model fields and data groups can be used to write specifications that follow information hiding and explain the related rules. We restrict the presentation to model fields of single objects, that is, objects whose representation does not depend on the states of referenced objects. Aggregate objects introduce complications related to aliasing that are orthogonal to the concerns of this paper. We have shown how to address these complications elsewhere [24, 25].

4.1 Writing Client-Visible Specifications

As we just discussed, clients cannot use the private specifications of `move` and `undo` to reason about dynamically-bound calls (Rule 3). Non-privileged clients also cannot use the protected specification case of `move` (Rule 2). This means that the specifications given in Figs. 1 and 2 are not useful for reasoning by non-privileged clients. However, Rule 1 does not allow these specification cases, as written, to be made public, because they mention non-public fields.

A standard way to fix this problem of expressing specifications in a client-visible way is by using data abstraction [9]. JML supports data abstraction through model fields [4, 16, 19]. A model field is a specification-only field whose value is determined by applying an abstraction function to an object state. In Fig. 3, we use JML model fields to provide a corrected specification for class `Point` (compare to Fig. 1). The value of each of the public model fields `x` and `y` is determined by a simple lookup of the corresponding protected field `_x` or `_y`. This relation between model and concrete fields is specified by two represents clauses. Following Rule 1, these represents clauses are protected since they mention protected fields. Analogously, the model fields `oldX` and `oldY` are used to abstract from the corresponding concrete fields `_oldX` and `_oldY`.

Each model field is associated with a data group [17]. A *data group* is a set of locations; it serves two purposes. First, modular verification requires the ability to determine whether a field update affects the value of a model field g even if the represents clause for g is hidden [16]. The data group of g is a superset of all fields whose update might affect g's value. Therefore, an update of a field outside g's data group is known not to affect g's value. For this to be sound, the evaluation of g's represents clause can only access fields within its data group. Second, to enable information hiding in assignable clauses, a method m is allowed to assign to a field f (perhaps via a method call) if m's assignable clause directly mentions f or if m's assignable clause mentions a model field g, and f is in g's data group.

In JML, data group membership is declared by **`in`** clauses. The membership relation is transitive. The clause **`in`** `oldX` in the declaration of the concrete field `_oldX` declares it to be a member of `oldX`'s data group. This membership allows `_oldX` to be accessed in the evaluation of the represents clause for the model field `oldX`. It also permits methods that mention `oldX` in their assignable clause to as-

```
public class Point2 {
  //@ public model int x, y;
  protected int _x; //@ in x;
  protected int _y; //@ in y;
  //@ protected represents x <- _x;
  //@ protected represents y <- _y;

  //@ public model int oldX, oldY;
  private int _oldX; //@ in oldX;
  private int _oldY; //@ in oldY;
  //@ private represents oldX <- _oldX;
  //@ private represents oldY <- _oldY;

  //@ public  invariant x >= 0  &&  y >= 0;
  //@ private invariant _oldX >= 0 && _oldY >= 0;

  /*@ public normal_behavior
    @  requires x + dx >= 0 && y + dy >= 0;
    @  assignable x, y, oldX, oldY;
    @  ensures x == \old(x + dx) && y == \old(y + dy);
    @  ensures oldX == \old(x) && oldY == \old(y);
    @*/
  public void move(int dx, int dy) {
    // implementation like in class Point
  }

  /*@ public normal_behavior
    @  assignable x, y;
    @  ensures x == \old(oldX) && y == \old(oldY);
    @*/
  public void undo() {
    // implementation like in class Point
  }

  // other methods and constructors as for Point.
}
```

Figure 3. A variant of class `Point` using model fields. Model fields allow specifications to be visible to clients without violating information hiding.

sign to the concrete field `_oldX` without explicitly mentioning the concrete field. This shows how data groups enable information hiding in assignable clauses. In particular, the assignable clause of `Point2`'s `move` method no longer needs to expose the hidden fields `_oldX` and `_oldY`, which was noted as illegal in Fig. 1.

Public model fields allow us to describe the behavior of `move` (and `undo`) publicly, without violating information hiding. The public specification of `move` has several major advantages. First, it is useful for all clients. Second, its precondition is now strong enough to prove that it preserves `Point2`'s invariants. Third, its public specification case is visible to subclasses and so can be used to reason about dynamically-bound calls. Finally, its specification is simpler, since it no longer has two specification cases.

Nevertheless, the specification of `Point2` allows one to change implementation details, such as the names of the hidden concrete fields, without changing the public specifications. Changing such hidden field names would only entail changes in the hidden represents clauses.

4.2 Rules for Model Fields

Leino and Nelson [16, 20] developed a modular verification technique that supports model fields. In this section, we adopt their rules and show how to enforce them in JML.

Updating a concrete field f might implicitly change the values of all model fields whose data group contains f. A method m that updates f has to describe these implicitly modified model fields in its assignable clause. To allow m to determine the set of such model fields, they must all be visible in m. This leads to the following rule.

Rule 4 (Authenticity) *If the data group of a model field g contains a concrete field f, then g must be visible wherever f is visible.*

In our example, the declaration of `oldX` is visible wherever `_oldX` is. Therefore, any method that might update `_oldX` can also see the declaration of `oldX`.

Rule 4 allows one to verify the assignable clause of a method m by only considering visible fields. The benefit for maintenance is that hidden fields can be added or removed without affecting the proof for m's assignable clause.

The semantics of assignable clauses allows a method m to modify a field f if f is contained in the data group of a field g mentioned in m's assignable clause. Therefore, to determine whether a call to m potentially modifies f, one must be able to determine whether f is in g's data group by only considering visible declarations. Hence the following.

Rule 5 (Data Group Membership) *Wherever two fields f and g are visible, one must be able to determine whether f is in g's data group.*

In our example, suppose a non-privileged client of `undo` wants to determine whether the method potentially modifies `oldX`. To such a caller, both `x` and `oldX` are visible, and therefore by Rule 5 any **`in`** clauses that connect the two must also be visible. Since there are no such **`in`** clauses, the client can correctly conclude that `oldX` remains unchanged.

Lemma 2 *In JML, both Rules 4 and 5 automatically follow from context conditions and Rule 1.*

Proof Sketch: By definition, the field f is in g's data group if and only if there is sequence of fields g_i, where $g_0 = g$, $g_n = f$, and for all $i \in \{0, \ldots, n-1\}$ g_i is mentioned in the **`in`** clause of the declaration of g_{i+1}. We show by induction on n that Rules 4 and 5 are satisfied.

For the base case ($n = 1$), the **in** clause of the declaration of a field f mentions g. Since the **in** clause has the same visibility as f, Rule 1 guarantees that g is visible wherever f is, which implies Rule 4. Since the **in** clause of f is visible wherever f is, Rule 5 is also satisfied.

For the induction step ($n > 1$), the **in** clause of the declaration of a field f mentions a field g_{n-1}. Since the **in** clause has the same visibility as f, Rule 1 guarantees that g_{n-1} is visible wherever f is. The inductive hypothesis implies that g is visible wherever g_{n-1} is and thus also wherever f is. Therefore, Rule 4 is satisfied.

Since the **in** clause of f is visible wherever f is, one can determine whether f is in the data group of g_{n-1} if both fields are visible. The inductive hypothesis implies that one can determine whether g_{n-1} is in the data group of g if both fields are visible. Therefore, Rule 5 is satisfied. ∎

5 Object Invariants

Object invariants describe consistency criteria for objects. In this paper, we use a visible state semantics [26]; that is, invariants have to hold in all visible states. The notion of a visible state is determined by the specification language. For instance, in JML the visible states are the pre- and post-states of all method executions except for methods that are explicitly declared as helpers (see Sec. 5.2).

As with model fields, we only consider invariants of single objects in this paper. See our earlier work [18, 26] for a discussion of invariants for aggregate objects. Moreover, we do not consider method calls in invariants.

Recall that each JML invariant has a visibility modifier.

5.1 Dependencies

We say that an invariant *depends* on a field f if the value of f is used to determine whether the invariant holds. This happens if the declaration of the invariant mentions f or a model field that has f in its data group. For instance, the first object invariant of class `Point2` (Fig. 3) depends on `x` and `y` because these model fields are mentioned in the invariant and also on `_x` and `_y` because these fields are in the data groups of `x` and `y`, respectively.

In languages such as Java, C++, and C#, the object invariant of an object o leads to proof obligations for the methods of o and also for client methods. This is because clients can directly assign to fields of o without using one of o's methods. For instance, privileged clients of class `Point2` may directly assign to the protected fields `_x` and `_y`. Such assignments potentially violate invariants depending on these fields. Therefore, these invariants must be visible to the clients in order to prove that they are preserved.

Rule 6 (Visible Invariants) *If an object invariant depends on a field f then the invariant must be visible wherever f is.*

Rule 6 can be enforced by a simple syntactic check on the fields mentioned in an invariant declaration.

Lemma 3 *Rule 6 follows from Rule 4 and the requirement that an invariant that mentions a concrete or model field g is visible wherever g is visible.*

The proof is a trivial case analysis, and so we omit it.

Following Rules 1 and 6, the fields an invariant directly mentions must be visible to exactly the same classes as the invariant itself. Suppose an invariant I mentions a field f. If f is visible to a class where I is not visible, Rule 6 is violated. By contrast, if I is visible to a class where f is not visible, Rule 1 is violated. In JML, this means that a private invariant can only mention private fields, as private fields and invariants are limited to the class in which they are declared. The situation is analogous for default access.

However, the situation is not so simple for other visibility modifiers. We discussed the problems of the `protected` modifier in Sec. 2. Perhaps surprisingly, there is also a problem with public invariants, when one considers subtyping. Rule 6 prevents a subtype S of T from depending on a field f of T, even if S's invariant is public, because by Def. 1, S and its invariant are in general not visible in T. Therefore, by Rule 6, the invariant of the subtype S must not depend on fields of the supertype T. Modular verification requires that one can reason about a type T without knowing all of its subtypes [15, 16, 24]. The problem is that methods of T or its clients may update f and, thereby, violate the invariant of S. In other words, adding S to a program requires re-verification of T and its clients, which is a maintenance problem. This problem of object invariants is not directly related to information hiding because it is neither solved by making all concrete fields private nor by making all invariants public. Therefore, instead of discussing this problem further, we refer the reader to other works that discuss this problem [16, 24, 26] and solutions [2, 29].

It is important to understand that Rules 1 and 6 do not prevent public invariants from *depending* on private fields. By using model fields, it is possible for an invariant to depend on a field without syntactically mentioning it. Therefore, invariants can express properties of hidden fields in a client-visible way. For example, the public invariant of class `Point2` (Fig. 3) indirectly depends on the protected field `_x` via the public model field `x`. This is permitted by Rule 1 because `_x` is not directly mentioned in the invariant.

Rule 6 enables modular reasoning about invariants because it is sufficient to prove that a method preserves all visible invariants. Invariants that are not visible to a method are preserved automatically because they cannot be violated.

Lemma 4 *If a program satisfies Rule 6 and does not contain any helper methods, then each invariant that is potentially violated by a method m is visible in m.*

Proof Sketch: Consider an execution of a method m. We may assume that this execution terminates; otherwise there is no post-state in which an invariant could be violated. We continue by induction on the depth of nesting of method executions.

In the base case, the execution of m does not call any other methods. Therefore, the only way for m to violate an invariant is by updating a field f. Type rules guarantee that f is visible in m and so, by Rule 6, all invariants that depend on f are visible in m. By the definition of "depends on," these invariants are the only ones that are potentially affected by the update of f. Consequently, all invariants that are potentially violated by m are visible in m.

For the inductive step, we have to consider (1) field updates and (2) method calls. Case 1 for field updates is analogous to the base case. In a visible state semantics, which we are assuming, all invariants hold in the post-state of a non-helper method. Therefore, case 2 is trivial. ∎

Lemma 4 does not only enable modular verification, but also facilitates maintenance. Since hidden invariants do not lead to proof obligations for clients, these invariants can be modified without re-verifying clients. Being able to adapt private implementation invariants is particularly important during maintenance and refactoring.

5.2 Helper Methods

It is often useful to exclude the pre- and post-states of certain methods from the set of visible states. JML declares such methods with the keyword **`helper`**.

Helper methods may violate invariants. This can make it difficult for a non-helper caller of a helper method to show that it preserves the invariants. To illustrate this problem, consider the helper method `undoX` in Fig. 4, which is an addition to class `Point2`.

Method `undoX` undoes the last change to `_x` and supplies a new value to `_oldX` for the next undo operation. This method potentially violates `Point2`'s private invariant since it does not require the new value for `_oldX` to be non-negative. If `undoX` is called from a client C, then C is in general not able to show that it preserves this invariant because the private invariant is not visible to C. Note that it is not enough to know the exact behavior of `undoX`, which is specified in Fig. 4, because C does not know how to re-establish the hidden invariant. The problem would not occur if `undoX` were private, because then all callers would be able to see the private invariant. This shows that the visibility of helper methods must be restricted.

Rule 7 (Helper Methods) *Each invariant that is potentially violated by a helper method must be visible wherever the helper method is visible.*

```
/*@ protected normal_behavior
  @  assignable x, oldX;
  @  ensures x == \old(oldX) && oldX == v;
  @*/
protected /*@ helper @*/    // illegal
void undoX(int v) {
  _x = _oldX;
  _oldX = v;
}
```

Figure 4. A helper method for class `Point2`, which is illegal because helpers must be private.

JML enforces this by making helper methods be private.

Lemma 5 *If all helper methods are private and the program satisfies Rule 6, then Rule 7 is satisfied.*

Proof Sketch: Consider an execution of a private helper method m. We may assume that this execution terminates; otherwise there is no post-state, and Rule 7 is satisfied trivially. We continue by induction on the depth of nesting of method executions.

In the base case, the execution of m does not call any other methods. As in the base case for Lemma 4, we have that all invariants that are violated by m are visible in m. Since m is private, they are visible wherever m is visible.

For the inductive step, we have to consider (1) field updates, (2) calls to non-helper methods, and (3) calls to helper methods. Cases 1 and 2 are identical to the inductive step for Lemma 4. For Case 3, suppose that m calls a helper method n. By the inductive hypothesis, each invariant that is potentially violated by n is visible wherever n is visible. Since both m and n are private, these invariants are also visible wherever m is visible. ∎

Rule 7 enables modular reasoning about invariants and efficient proof maintenance in the presence of helper methods. That is, Lemma 4 holds also for programs that contain helper methods.

Lemma 6 *If a program satisfies Rules 6 and 7, then each invariant that is potentially violated by a method m is visible in m.*

Proof Sketch: The proof is analogous to the proof for Lemma 4. For the inductive step, we have to consider the additional case that m calls a helper method n. By the inductive hypothesis, each invariant that is potentially violated by n is visible in n. By Rule 7, these invariants are also visible in m. ∎

6 Related Work

Parnas [28] recognized the importance of information hiding for the maintenance of programs. We apply the con-

cept of information hiding to specifications, using an analysis based on formal specifications and proofs.

Meyer [23] discusses visibility rules for assertions and suggests the visible preconditions rule (similar to Rule 2) as well as a requirement similar to the rule for helper methods (Rule 7). We extend his work to model fields, invariants, and Java-like visibility modifiers.

Modular verification requires that the proof obligations for a class deal only with program and specification elements that are visible to that class. Thus information hiding and modular verification techniques are closely related. However, most existing work on modular verification (e.g., [1, 3, 9, 16, 32]) supports only a limited form of information hiding, whereas the rules presented here support the full range of visibility found in languages such as Java.

Leino and Nelson's work on model fields [20] and data groups [17] suggests the rules for model fields we use. Whereas their papers consider only the visibility modifiers **`private`** and **`public`**, we also discuss the implications for the other modifiers found in Java and C#.

Müller's thesis [24] presents sound verification rules in the presence of information hiding. Our rules support stronger information hiding, including hiding of invariants.

Ruby and Leavens [29] allow subclass invariants to depend on superclass fields. They achieve soundness by imposing additional obligations on subclasses, such as mandatory overrides of some superclass methods, and by prohibiting some super-calls.

In Spec# [3], method specifications have the same visibility as the method itself, and invariants are always private. Spec# enforces Rule 1 and, like JML, allows hidden fields to be declared public for specification purposes. Spec# does not use a visible state semantics for invariants. Instead, invariants are checked at designated pack statements [18]. Since Spec# does not enforce Rule 6, client code has to be re-verified when an invariant is changed. This re-verification could be avoided by restricting where pack statements may be placed. However, it is unclear how severe such a restriction would be. In combination with the rules we propose in this paper, such a placement restriction would also allow representations of Spec#'s model fields [19] to be changed without re-verifying client code.

In the Object Constraint Language OCL, assertions have the same visibility as the element of the UML model they are associated with. Rule 1 is mentioned in the standard [27, Sec. 2], but not mandatory. Since none of our rules are enforced, OCL supports neither modular verification nor efficient proof maintenance.

7 Conclusions

There are maintenance advantages to having some specifications at different visibility levels. This is especially true for invariants and other specification language constructs that describe detailed design decisions about implementations, such as relationships among hidden fields. Hidden method specifications are less important, but are still useful, for instance, for verifying calls to overridden methods (super-calls). This is especially the case when the public model of an object does not uniquely determine the values of hidden fields, and in which the hidden details matter (for instance, for efficiency). Hiding such specifications allows design decisions to be changed without invalidating client proofs. It also enables a more efficient dependency analysis when design decisions are changed. For all these reasons, such hidden specifications are extensively used in JML's specification of the Java libraries.

When specifications can be written at different visibility levels, there are several soundness problems that arise. These are avoided by following the rules we have described above. While the rules are intuitive, they have non-obvious consequences. For example, Rule 6 prohibits mentioning public fields in non-public invariants. Another non-obvious consequence is that helper methods must be private, not merely hidden (Rule 7). Finally, the rules are careful to use transitive visibility ("wherever"), since visibility modifiers such as **`protected`** are not transitive in general.

As mentioned in the introduction, these rules can also be used for unit tests (partial specifications) and documentation (informal specifications) when augmented with a notion of visibility. For example, by Rule 1 a "public" unit test for a class should not access the class's private members, for instance, in the code that decides if a test fails. Similarly, Rule 3 says that no clients can use "private" documentation describing dynamically-bound method calls.

While JML already enforces most of our rules, this paper explores some parts of JML's semantics for the first time. In particular the semantics for preconditions described in Sec. 3.1 is not implemented yet, and restrictions on data-groups and Rule 6 are not enforced. Some JML tools, including ESC/Java2 [11, 7], ignore visibility modifiers in specifications. Thus documentation and enforcement of these rules is important future work, as is investigation of their completeness. JML also relaxes some of our assumptions, especially about method calls in invariants and represents clauses; this gives more problems for future work.

Future work could also extend our rules to cover new language constructs. Language designers could do this by first determining the semantics of these constructs in terms of proof obligations, then sorting out what obligations fall on the clients and the various entities being specified (such as methods), and finally constructing rules to make sure that each client and each such entity can see the specifications necessary to meet its proof obligations.

Acknowledgments. Thanks to the formal methods club at ETH for discussions about the ideas contained in this paper, and to Patrice Chalin, David Cok, Faraz Hussain, Clyde Ruby, and Joseph N. Ruskiewicz for comments on drafts.

References

[1] P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 60–90. Springer, New York, NY, 1991.

[2] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6), 2004.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[4] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, 2005.

[5] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference on Software Engineering (ICSE)*, pages 258–267. IEEE Computer Society Press, 1996.

[6] *Standard ECMA-367: Eiffel Analysis, Design and Programming Language*. ECMA International, 2005.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM, 2002.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[9] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[10] G. Kiczales and J. Lamping. Issues in the design and documentation of class libraries. In A. Paepcke, editor, *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 27(10) of *ACM SIGPLAN Notices*, pages 435–451, 1992.

[11] J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.

[12] G. T. Leavens. JML's rich, inherited specifications for behavioral subtypes. Technical Report 06-22, Department of Computer Science, Iowa State University, Aug. 2006. To appear in *ICFEM'06*.

[13] G. T. Leavens and A. L. Baker. Enhancing the pre- and post-condition technique for more expressive specifications. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods (FM)*, volume 1709 of *LNCS*, pages 1087–1106. Springer, 1999.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[15] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Aug. 2006.

[16] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[17] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153, October 1998.

[18] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.

[19] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 115–130. Springer, 2006.

[20] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[21] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[22] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[23] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition edition, 1997.

[24] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *LNCS*. Springer, 2002.

[25] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency & Computation: Practice & Experience*, 15:117–154, 2003.

[26] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[27] OMG. Object constraint language specification, version 2.0. `http://tinyurl.com/k7rfm`, May 2006.

[28] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, Dec. 1972.

[29] C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Object-Oriented Programming, Systems, Languages, and Applications (OOSPLA)*, volume 35(10) of *ACM SIGPLAN Notices*, pages 208–228, Oct. 2000.

[30] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 30(10) of *ACM SIGPLAN Notices*, pages 200–214, 1995.

[31] A. Wills. Specification in Fresco. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer, 1992.

[32] J. M. Wing. A two-tiered approach to specifying programs. Technical Report TR-299, MIT LCS, 1983.