Enforcing Information Hiding in Interface Specifications:
with The AspectJML specification language

A Client-Aware checking Approach

Henrique Rebêlo
Universidade Federal de Pernambuco
Brazil

Gary T. Leavens
University of Central Florida
USA
What is information hiding?
How to abstract away the details?

The task of the software development team is to engineer the illusion of simplicity.
Black-box abstraction

The task of the software development team is to engineer the illusion of simplicity.

© Copyright 1994 - Extracted from Booch’s OOAD book
Parnas

Whatever *is likely* to change!

Hiding the secret of a module behind an interface
Abstraction is an important key

Abstraction allows you to take a simpler view of a complex concept.
Encapsulation helps in the process

Figure 5-8  Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are *not* allowed to look at any of the details of the complex concept. What you see is what you get—it’s all you get!
Is Encapsulation equivalent to Information Hiding?
Think about these examples

```java
class EncapsulationWithoutInformationHiding {
    private ArrayList list = new ArrayList();

    public ArrayList getList() {
        return this.list;
    }
}

class InformationHidingWithoutEncapsulation {
    public List list = new ArrayList();
}
```
Avoid exposure implementation details

Encapsulation hides the details of the implementation of an object.

© Copyright 1994 – Extracted from Booch’s OOAD book
Information hiding for other artifacts
(Leavens and Muller. ICSE, 2007)

• Visibility modifiers on specifications
• Some specifications hidden from some clients
• Some specifications say more to privileged clients

```java
class Package {
   // @ public model JMLDouble pWeight;
   private double weight;
   // @ private represents weight = pWeight;

   // public normal_behavior
   @ requires weight <= 5;
   @ ensures this.pWeight == weight;
   @ also
   @ private normal_behavior
   @ requires weight <= 5;
   @ ensures this.weight == weight;

   public void setWeight(double weight) {
      this.weight = weight;
   }

   /* other methods omitted */
}```
Design by Contract

• Specifications (contracts) in OO programming Language
  • preconditions
  • postconditions

```plaintext
decrement is
  -- Decrease counter by one.
  require
  item > 0
  ensure
  item = old item - 1
```
Running example
(Package delivery system)

https://www.google.com.br/search?q=Package+delivery
Delivery package classes...

- Package,
- Gift-Package
- Coupon-Package
- Courier
- ...

```
Package
  setWeight(double)
  setSize(double, double)
  reSize(double, double)
  containsSize(double, double)
...
```

```
GiftPackage
  setWeight(double)
  setSize(double, double)
...
```

```
CouponPackage
  setWeight(double)
  setSize(double, double)
...
```
Package contracts with a DbC language

class Package {
   /* intentionally public */
   public double weight;

   public void setWeight(double weight) {
      @pre weight <= 5;
      @post this.weight == weight;
      this.weight = weight;
   }

   /* other methods omitted */
}

/* other methods omitted */
Consider the following Package’s client

```java
class ClientClass {
    public void clientMeth(Package p) {
        p.setWeight(5);
    }
}
```

```java
class Package {
    /** intentionally public */
    public double weight;

    public void setWeight(double weight)
    @pre weight <= 5;
    @post this.weight == weight;
    {
        this.weight = weight + 1;
    }

    /* other methods omitted */
}
```

RAC

PostconditionError:
this.weight is 6.0
weight is 5.0

Written by Cathy

Written by Alice
Consider now the following change by Alice

```java
class Package {
    private double weight;

    public void setWeight(double weight) {
        @pre weight <= 5;
        @post this.weight == weight;
        this.weight = weight;
    }

    /* other methods omitted */
}
```
But now RAC breaks information hiding!

```java
class Package {
    private double weight;
    public void setWeight(double weight) {
        this.weight = weight + 1;
    }
}
```

```java
class ClientClass {
    public void clientMeth(Package p) {
        p.setWeight(5);
    }
}
```

**PostconditionError:**

- this.weight is 6.0
- weight is 5.0
Kiczales: Beyond the black-box

Clients confront an issue that the interface claimed to hide.

An open implementation presents two interfaces
Do DbC languages present this information hiding problem?
code contracts

Is this program correct?

```csharp
using System;
using System.Diagnostics.Contracts;

class Package {
    private double weight;

    public void setWeight(double weight) {
        Contract.Requires(weight <= 5);
        Contract.Ensures(this.weight == weight);
        this.weight = weight + 1;
    }
}

class ClientClass {
    public void clientMeth(Package p) {
        p.setWeight(5);
    }
}
```

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Possible precision mismatch for the arguments of ==</td>
</tr>
<tr>
<td>2 ensures is false: this.weight == weight</td>
</tr>
</tbody>
</table>
In this scenario, we can say that...
... standard DbC/RAC tools are NOT...

Effective

+ 

Useful
But the DbC language **JML** starting fixing the problem...
Java modeling language—JML

• **Formal specification language** for Java
  – behavioral specification of Java modules
• Adopts design by contract based on Hoare-style with **assertions**
  – pre-, postconditions and invariants
  – \{P\} C \{Q\}
• Main goal → **Improve functional software correctness** of Java programs
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 (overridable) methods must be visible to overriding subclass methods, otherwise the overriding method cannot respect behavioral subtyping [1, 5, 15, 21].
Kinds of clients in Java and JML
class Package {
//@ public model JMLDouble pWeight;
private double weight;
//@ private represents weight = pWeight;

/*@ public normal_behavior*/
//@ requires weight <= 5;
//@ ensures this.pWeight == weight;@
//@ also
//@ private normal_behavior
//@ requires weight <= 5;
//@ ensures this.weight == weight;
public void setWeight(double weight) {
    this.weight = weight;
}

/*@ public normal_behavior*/
//@ requires weight <= 5;
//@ ensures this.pWeight == weight;@
//@ also
//@ private normal_behavior
//@ requires weight <= 5;
//@ ensures this.weight == weight;
public void setWeight(double weight) {
    this.weight = weight;
}

/* other methods omitted */
}

Package contracts with JML
JML RAC still breaks information hiding!

```java
class ClientClass {
    public void clientMeth(Package p) {
        p.setWeight(5);
    }
}
```

```
class Package {
    //@ public model JMLDouble pWeight;
    private double weight;
    //@ private represents weight = pWeight;
    
   /*@ public normal_behavior
    @   requires weight <= 5;
    @   ensures this.pWeight == weight;
    @ also
    @ private normal_behavior
    @   requires weight <= 5;
    @   ensures this.weight == weight;
    public void setWeight(double weight) {
        this.weight = weight + 1;
    }
    /* other methods omitted */
}
```

JMLPostconditionError: when this.weight is 6.0
weight is 5.0
The problem can become even worse...
class Package {
   //@ public model JMLDouble pWeight;
   protected double weight;
   //@ @ protected represents weight = pWeight;

   //@ @ public normal_behavior
   @ requires weight <= 5;
   @ ensures this.pWeight == weight;
   @ also
   @ @ protected normal_behavior
   @ requires weight <= 8;
   @ ensures this.weight == weight;
   public void setWeight(double weight) {
      this.weight = weight;
   }

   /* other methods omitted */
}

Package contracts for subtypes

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>setWeight(...)</td>
</tr>
<tr>
<td>GiftPackage</td>
<td>setSize(...)</td>
</tr>
<tr>
<td></td>
<td>reSize(...)</td>
</tr>
<tr>
<td></td>
<td>containsSize(...)</td>
</tr>
<tr>
<td></td>
<td>setWeight(...)</td>
</tr>
<tr>
<td>CouponPackage</td>
<td>setSize(...)</td>
</tr>
<tr>
<td></td>
<td>reSize(...)</td>
</tr>
<tr>
<td></td>
<td>containsSize(...)</td>
</tr>
<tr>
<td></td>
<td>setWeight(...)</td>
</tr>
</tbody>
</table>

Diagram: Package inheritance and contracts.
JML RAC misses a precondition violation!

```java
class ClientClass {
    public void clientMeth(Package p) {
        p.setWeight(8);
    }
}
```

```java
class GiftPackage extends Package {
}
```

Returns successfully!
JML/RAC is NOT...

Effective

+ Useful
Unanswered questions can arise

What happened with RAC? Did Alice specified correctly?

Did I provide the right specifications?

Did Cathy associated the right specs during RAC?
This is caused by the...
...supplier-side instrumentation of contracts in JML and any other RAC

class Package {
    //@ public model JMLDouble pWeight;
    protected double weight;
    //@ protected represents weight = pWeight;

    //@ public normal_behavior
    // also
    //@ protected normal_behavior
    public void setWeight(double weight) {
        this.weight = weight;
    }

    /* other methods omitted */
}

class Package {
    ...
    public void setWeight(double weight) {
        //@ assume w <= 5 || w <= 8;
        ...
        //@ assert this.pWeight == weight
        // this.weight == weight;
    }

    /* other methods omitted */
}
we say that a RAC compiler that checks specifications based at supplier-side as overly-dynamic
The AspectJML Language

is one

solution

to the illustrated problem
Client-aware checking approach

- CAC cuts through clients
  - with proper runtime checks
- Runtime checking itself is modular
  - based on privacy-kind of clients
Harrison & Harold Ossher on Subjectivity

Diagram showing relationships between concepts such as 'maple', 'cherry', 'locust', 'pine', 'dandelion', 'bird', 'woodsman', 'hardwood', 'softwood', 'tree', 'nontree', 'plant', 'nestable', 'predator', 'object', 'nectar plant', and 'insect plant'.
Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.
CAC implementation with AspectJML

JML annotated Java source files

OOP

Classes

Weaver

AOP

Advice

Aspects with JML features

Class.class

Advice

Advice
To hide or not to hide?

- CAC cuts through clients
  - with proper runtime checks
- Runtime checking itself is modular
  - based on privacy-kind of clients
Future work

• Find case studies

• More study on the problems caused by overly-dynamic checking
  • dynamic-dispatch
AspectJML/CAC in action…
Dedicated to the Memory of

Robert France