

Last update: 3 April 2007

Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 3: Language constructs for modularity and information hiding

Ilinca Ciupa

Information hiding

Underlying question: how does one "advertise" the capabilities of a module?

Every module should be known to the outside world through an official, "public" interface.

The rest of the module's properties comprises its "secrets".

It should be impossible to access the secrets from the outside.

Overview

Review of modularity and information hiding (discussed in previous lecture)

- Information hiding mechanisms in Eiffel
- Information hiding mechanisms in Java
- Comparison: Eiffel vs. Java
- Information hiding mechanisms in C#
- Modularity and information hiding in Ada 83
- Wrap-up

Information Hiding Principle

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

Modularity

Reusability + Extendibility

Some principles of modularity:

- > Decomposability
- > Composability
- > Continuity
- > Information hiding
- > The open-closed principle
- > The single choice principle

How to ensure information hiding

NOT through management or marketing policies
BUT through **language rules**

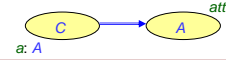
Encapsulation

Wrapping data and routines within classes in combination with information hiding is often called "encapsulation". However, "encapsulation" is often used as a synonym for information hiding.

Abstraction and client privileges

If class A has an attribute att : $SOME_TYPE$, what may a client class C with

$a: A$ do with $a.att$?



Read access if attribute is exported

> $a.att$ is an expression.

> An assignment ~~$a.att = v$~~ would be syntactically illegal!

(It would assign to an expression, like ~~$x + y := v$~~ .)

Overview

Review of modularity and information hiding (discussed in previous lecture)

Information hiding mechanisms in Eiffel

Information hiding mechanisms in Java

Comparison: Eiffel vs. Java

Information hiding mechanisms in C#

Modularity and information hiding in Ada 83

Wrap-up

Applying abstraction principles

Beyond read access: full or restricted write, through exported procedures.

Full write privileges: `set_attribute` procedure, e.g.

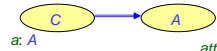
```
set_temperature (u: REAL) is
  -- Set temperature value to u.
do
  temperature := u
ensure
  temperature_set: temperature = u
end
```

Client will use e.g. `x.set_temperature (21.5)`.

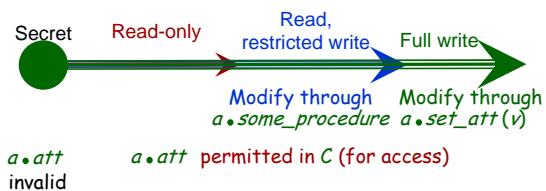
Possible client privileges in Eiffel

If class A has an attribute att : $SOME_TYPE$, what may a client class C with

$a: A$ do with $a.att$?



The attribute may be:



Other uses of a setter procedure

```
set_temperature (u: REAL) is
  -- Set temperature value to u.
```

```
require
  not_under_minimum: u >= -273
  not_above_maximum: u <= 2000
do
  temperature := u
  update_database
ensure
  temperature_set: temperature = u
end
```

Having it both ways

Make it possible to call a setter procedure

```
temperature: REAL assign set_temperature
```

Then the syntax

```
x.temperature := 21.5
```

is accepted as a shorthand for `x.set_temperature(21.5)`

Retains contracts etc.

Exporting selectively

```
class
  LINKABLE[G]
  feature {LINKED_LIST}
```

```
  put_right(...) is do ... end
```

```
  right: G is do ... end
```

```
  ...
end
```

These features are selectively exported to `LINKED_LIST` and its descendants (and no other classes)

Information hiding

```
class A
```

```
feature
```

```
  f ...
```

```
  g ...
```

```
feature {NONE}
```

```
  h, i ...
```

```
feature {B, C}
```

```
  j, k, l ...
```

```
feature {A, B, C}
```

```
  m, n ...
```

```
end
```

Status of calls in a client with `a1: A`:

- > `a1.f, a1.g`: valid in any client
- > `a1.h`: invalid everywhere (including in `A`'s own text!)
- > `a1.j`: valid only in `B, C` and their descendants (not valid in `A`)
- > `a1.m`: valid in `B, C` and their descendants, as well as in `A` and its descendants

Information hiding

Information hiding only applies to use by clients, using dot notation or infix notation, as with `a1.f` (*Qualified calls*).

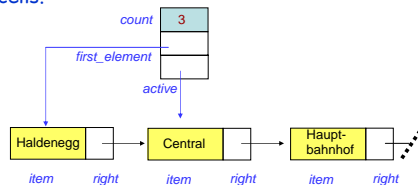
Unqualified calls (within class) not subject to information hiding:

```
class A feature {NONE}
  h is ... do ... end
feature
  f is
    do
      end
  ...; h ...
end
```

An example of selective export

`LINKABLE` exports its features to `LINKED_LIST`

- > Does not export them to the rest of the world
- > Clients of `LINKED_LIST` don't need to know about `LINKABLE` cells.



Overview

Review of modularity and information hiding (discussed in previous lecture)

Information hiding mechanisms in Eiffel

Information hiding mechanisms in Java

Comparison: Eiffel vs. Java

Information hiding mechanisms in C#

Modularity and information hiding in Ada 83

Wrap-up

Possible client privileges in Java

Access specifiers (placed in front of each definition for each member of the class):

- > **public**
- > **protected**
- > **Package access (no keyword)**
- > **private**

Overview

Review of modularity and information hiding (discussed in previous lecture)

Information hiding mechanisms in Eiffel

Information hiding mechanisms in Java

Comparison: Eiffel vs. Java

Information hiding mechanisms in C#

Modularity and information hiding in Ada 83

Wrap-up

Access specifiers

public

- > The member declared to be public is available to everyone

private

- > No one can access that member except the class that contains that member, inside methods of that class

protected

- > Member can be accessed by
 - Descendants of the class
 - Classes in the same package

Package access

- > **Default**
- > Also called "friendly"
- > All other classes in current package have access to that member
- > To all classes outside of current package, the member appears to be **private**

Comparison: Eiffel vs. Java

Access level	Eiffel	Java
only current class	-	private
only current class and its descendants	feature {NONE}	-
current class + "friends"	feature {B, C} ("friends" = B, C and their descendants)	default ("friends" = classes in the same package)
current class + its descendants + "friends"	feature {A, B, C} ("friends" = B, C and their descendants, A = current class)	protected ("friends" = classes in the same package)
everyone	feature {ANY}	public

Access modifiers at the class level

Either **public** or default (no access modifier)

- > **public**
 - Appears before the `class` keyword
 - Makes the class available to a client programmer
- > **No access modifier**
 - Makes the class available only within the package

No **private** and **protected**!

More comparison: Eiffel vs. Java

Eiffel - no **package** mechanism

Eiffel - no way of hiding a feature from your descendants

- > **Module viewpoint: If B inherits from A, all the services of A are available in B (possibly with a different implementation).**

Java - no way of exporting a member only to self and descendants

Java - no language rule to distinguish between access to attributes for reading and for writing

Java - additional way of making a class available outside its package or not

Access control more fine grained in Eiffel

Overview

Review of modularity and information hiding (discussed in previous lecture)
Information hiding mechanisms in Eiffel
Information hiding mechanisms in Java
Comparison: Eiffel vs. Java
Information hiding mechanisms in C#
Modularity and information hiding in Ada 83
Wrap-up

The C# solution: properties

```
public class Heater {  
    private int TemperatureInternal; ← attribute  
    public int Temperature { ← property  
        get {return TemperatureInternal;}  
        set {  
            if (!InRange(value)) {  
                throw new ArgumentException  
                    ("Temperature out of range");  
            }  
            TemperatureInternal = value;  
            NotifyObservers();  
        }  
    }  
}
```

C# access modifiers

C# adds the **internal** access modifier, which restricts access within the assembly
Classes can be:
 > **public**
 > **internal**
Class members can be:
 > **public** - accessible to everyone
 > **internal** - accessible only from current assembly
 > **protected** - accessible only from containing class or types derived from containing class (a.k.a. "family" export status)
 > **protected internal** - accessible only from current assembly or types derived from the containing class
 > **private** - accessible only from containing type

Overview

Review of modularity and information hiding (discussed in previous lecture)
Information hiding mechanisms in Eiffel
Information hiding mechanisms in Java
Comparison: Eiffel vs. Java
Information hiding mechanisms in C#
Modularity and information hiding in Ada 83
Wrap-up

The problem with attribute export status

If an attribute is exported, clients can both read it and **assign any value that they want** to it.

Ex: heater.temperature := 19

Packages in Ada 83

Construct for grouping logically related program elements
Purely *syntactic* notion
Only needed for readability and manageability of the software
The decomposition of a system into packages does not affect its semantics
Unlike the classes of object-oriented languages, a package does not by itself define a type

Ada packages - Separation of interface and body

Every package has 2 parts:

- > "specification"
 - Introduced by the keyword `package`
 - Contains the interface
 - Lists the public properties of the package: exported variables, constants, types, and routines (only headers)
- > "body"
 - Introduced by the keywords `package body`
 - Provides the routines' implementations
 - Adds any needed secret elements
 - Needs to repeat most of the interface information (routine headers) that already appeared in the interface

Overview

Review of modularity and information hiding (discussed in previous lecture)

Information hiding mechanisms in Eiffel

Information hiding mechanisms in Java

Comparison : Eiffel vs. Java

Information hiding mechanisms in C#

Modularity and information hiding in Ada 83

Wrap-up

Ada - the private story

The problem:

- > Declaring a type in the interface part of a package gives clients unrestricted access to the type, and not declaring it hides it completely from the clients

The solution:

- > The `private` part of the interface (introduced by the `private` keyword)
- > Any declaration appearing in the private part is unavailable to clients

Wrap-up

Why control access to features?

- > to allow the class designer to change the internal workings of the class without worrying about how it will affect the client programmer (service to class designer)
- > to show a client programmer what is important to him/her and what not (service to client programmer)

Levels of access for clients may vary from no access to full read and write

Not all languages allow all levels of access

Ada - an example of a package interface

```
generic
  type G is private;
package STACKS is
  type STACK(capacity: POSITIVE) is private;
  procedure put(x: in G; s: in out STACK);
  procedure remove(s: in out STACK);
  function item(s: STACK) return G;
  function empty(s: STACK) return BOOLEAN;
  Overflow, Underflow: EXCEPTION;
private
  type STACK_VALUES is array (POSITIVE range <>) of G;
  type STACK(capacity: POSITIVE) is
    record
      implementation: STACK_VALUES(1..capacity);
      count: NATURAL := 0;
    end record
end STACKS;
```