Course: *Advanced Topics in Object Technology*

**Extensibility and Reusability:**
A.O.P. *vs.* O.O.P

*Case studies in Eiffel and AspectJ*

Philippe Lahire

*I3S Laboratory UNSA/CNRS – OCL team*

Zurich – 16 April 2003
Objective: answering some questions

- What are the programmer needs?
- Why looking at some other paradigms?
- What is ASoC?
- What is AOP?
- What is AspectJ?
- What are the pros and cons of ASoC?
- Should we extend O-O languages?
- *etc.*
Summary

I. Context of the presentation

- Programmer needs
- OOP limitations
- Main (current) implementation of AOP (AspectJ)

II. Case studies

- N°1: Implementating contracts
- N°2: Integrating design patterns *a posteriori*
- N°3: Equipping components with testing material

III. Conclusion and perspectives

- Summary
- Possible research topics
Setting the context

- Class-hierarchies:
  - Does perfect design exist?
  - Need of maintenance
  - Need of using new services (“big” functionalities)

- Context of use may vary:
  - Life-cycle: debug, test, etc.
  - Applications may not use it in the same way

- Its needs may evolve:
  - To add set of functionalities (one point /spread away)
  - To adapt existing functionalities

Perfect:
Everything is in the hierarchy (for ever) and
It is at the right place

A posteriori changes within class-hierarchy
Answers of O-O languages

- Inheritance
  - Multiple / repeated
  - Feature redefinition / renaming
  - Frozen / deferred feature
- Client-Supplier relationship
- Genericity
- Anchors
- Once features
- First-class features
Deficiencies of O-O languages

Hierarchy becomes Too complex
References (deficiencies of O-O)

• K. Ostermann, M. Mezini. “Object-Oriented Composition is Tangled”, *Workshop ASoC ECOOP 01*, 2001
• Case study: Eliminating Redundant Codes in the Buffer Library. [http://www.comp.nus.edu.sg/~lisb/javalib/classes.htm](http://www.comp.nus.edu.sg/~lisb/javalib/classes.htm)
• H. Ossher, W. Harrison. “Combination of inheritance hierarchies”, *OOPSLA ’92*
• U. Hölzle. “Integrating independently-Developed Components in Object-Oriented Languages”, *ECOOP ’93*
• …
Approach by Separation of Concerns

Data type encapsulation vs. Preoccupation encapsulation
ASoC (AspectJ): survey

Points addressed in this survey:

- Join Point Model
- Pointcut designators
- Advices
- Feature introduction
- Aspect declaration
- Reflection
- More on aspects: abstract, instantiation, …
- Implementation: static crosscutting
- …
ASoC (AspectJ): context

- **Java extension:**
  - Single inheritance
  - Interfaces, (abstract) classes
  - Modifiers: private, protected, public
  - Implicit redefinition / no renaming
  - Overloading, no variance

- **New features:**
  - Keep the same philosophy
  - Pattern matching language
  - …
**ASoC (AspectJ): Joint Point model**

class Person {
    protected Person partner;
    protected int age;

    public Person (int a) {
        setAge(a);
        partner = new Person(0);
    }

    public setPartner (Person s) {
        if (partner.age == 0)
            partner = s;
        else
            throw new … ;
    }
...
}
A pointcut is a program element, which allows catching a set of joint points within a hierarchy of classes.

A syntax for pointcut declaration:
- Naming it
- Picking out execution context
- Selecting joint points
- More complex pointcut

Type of Joint point:
- basicPointcut (call (public void Person.* (.., Person)) && target (p))

- Method modifier
- Method type
- Class container
- Method name
- Method parameters
- Associate an object to p
**ASoC (AspectJ): Advices**

pointcut viewNames (Person \(pe\), Person \(pa\)) :

\[
\text{target (} pe \text{)} && \text{args (} pa \text{)} && \text{call (public void Person.\(* (.., \text{Person})\))}
\]

**before** ( Person \(pe\), Person \(pa\)) :

viewNames (Person \(pe\), Person \(pa\))

{
   if (\(pe\.age == pa\.age\)) proceed (\(pe, pa\));
   else
      System.out.println ("Error …");
}

**Match joint point**
\* set context

**Action to perform**

**Possible advices:**
- before
- after
- around
ASoC (*AspectJ*): Aspect Declaration

```java
aspect PersonCrossCutting {
  pointcut viewNames (Person pe, Person pa) :
    target (pe) && args (pa) && call (public void Person.* (.., Person))
  before (Person pe, Person pa) :
    viewNames (Person pe, Person pa) {
      System.out.println (pa.name + “will become the partner of” + pe.name);
    }
  after ( Person pe, Person pa) :
    viewNames (Person pe, Person pa) {
      System.out.println (pa.name + “became the partner of” + pe.name);
    }
  around (Person pe, Person pa) :
    viewNames (Person pe, Person pa) {
      if (pe.age == pa.age) proceed (pe, pa);
      else System.out.println (“Error : the age of persons are not compatible”);
    }
  ...
}
```
ASoC (*AspectJ*): More about advices

Pointcut `viewNames (Person pe)`:

- `target (pe) && call (public Person Person.getPartner(..))`
- `after (Person pe)`
- `viewNames (Person pe)`

```java
{
    System.out.println (e.getMessage() +
            “is the explanation of exception”);
}
```

Exception raised by `getPartner (applied on pe)`

Possible clauses:
- `returning`
- `throwing`

Object on which the advice applies

Result of `getPartner` is attached to `pa`
ASoC (*AspectJ*): More about pointcuts

- **Some other pointcuts:**
  - *execution* (public * Person+.*(..))
  - *Initialization* (Person (int))
  - *get/set* (public int Person.*)
  - *handler* (Person)

- **Combining pointcuts:**
  - *get* (int Person+.*)) && ! *get* (* Employee.*)
  - *cflow* (execution (public * Person.set*(..)))
ASoC (AspectJ): More about pointcuts

Class Person {
  setPartner (Person p) {
    if (p.getAge() == age) {
      …
    }
  }
}

Person p1

OneClient

take into account the code of called routines

Target (Person)

args (Employee)

Class Person {
  setPartner (Person p) {
    if (p.getAge() == age) {
      …
    }
  }
}

cflow (Person.setPartner)
within code (Person.setPartner)

Within (Person)

within code (Person.setPartner)

Target (Person)

args (Employee)

Class Person {
  setPartner (Person p) {
    if (p.getAge() == age) {
      …
    }
  }
}

p1.setPartner(p)

call (* set* (..))

Any method

Starting by “set”

this (OneClient)

call (* set* (..))

Any method

Starting by “set”

this (OneClient)

p1.setPartner(p)

call (* set* (..))

Any method

Starting by “set”

this (OneClient)
ASoC (*AspectJ*): Static Crosscutting

**privileged aspect** PersonCrossCutting {

    public int count = 0

    private boolean Person.isEfficient = False;

    public void Person.setIsEfficient () { isEfficient = true;}

    public boolean Person.IsEfficient () { return isEfficient ;}

    private boolean application.*.trace = True;

    public void application.*.setTrace() {this.r;}

    declare parents : Person extends Observer ;

    declare parents : Person implements Subject ;

    pointcut …

    …

    }

In aspect only!

Attribute

Method

Several classes!

Class or interface

Access to private feature!
ASoC (*AspectJ*): More on aspects

```java
abstract aspect viewing {
    abstract pointcut viewNames (Person pe, Person pa) ;

    before (Person pe, Person pa) :
        viewNames (Person pe, Person pa) {
            System.out.println (pa.name + “will become the partner of” + pe.name); }
    ...
}

aspect viewing1 extends viewing {
    pointcut viewNames (Person pe, Person pa) ;
    target (pe) && args (pa) && call (public void Person.* (.., Person))
    ...
}
```

### issingleton

### Perthis (Person)
**ASoC (AspectJ): Reflection**

```java
aspect PersonCrossCutting {
  pointcut viewNames (Person pe) :
    target (pe) && call (public void Person+.\* (..))
  before (Person pe) : viewNames (Person pe) {
    Object targetObject = thisJoinPoint.getTarget(); // Dynamic
    Object currentObject = thisJoinPoint.getThis(); // Dynamic
    Object[] currentargs = thisJoinPoint.getArgs(); // Dynamic
    Signature currentSignature = thisJoinPoint.getSignature(); // Static
    Class currentClass = currentSignature.getDeclaringType(); // Static
    .... Doing something according to dynamic type of pe or J.P. static information
  }
}
```
ASoC (*AspectJ*): Summary

- To adapt existing hierarchies:
  - Piece of code in existing methods
  - New methods or attributes
  - New ancestors

- To adapt easily:
  - Reusable adaptation
  - Expressive language for the specification of adaptation

- Join points to catch where to adapt
- Non invasive adaptation
- Adaptations written in hierarchy of aspects
ASoC and Component adaptations

Possible needs

• To add testing material
  ✓ Check/debug clauses from Eiffel
  ✓ To handle capture/replay of events in GUI (regression testing)
    – Test automation (execution, logs, generation)

✓ To insert design patterns
✓ To handle assertions

• To try some structure modifications
• To try some new implementation
• …

• More generally : to add/adapt/“remove” functionalities
Case-study n°1: Handling of contracts

**Description:**
- ✓ To provide a way to define assertions
- ✓ To provide a mechanism to execute assertions

**Context:**
- ✓ Adding a new service to the programmer
- ✓ It is orthogonal to application specification

\[ \text{Is ASoC a good alternative to implement assertions?} \]
\[ \text{Is ASoC a good alternative to implement check / debug?} \]
class PERSON
feature
  partner: PERSON
  age: INTEGER
  is_married: BOOLEAN

set_age (a: INTEGER) is
  -- Set age to a
  require
    a >= 0
  do
    age := a
  ensure
    age = a
end

set_partner (s: PERSON) is
  -- Set partner to s.
  require
    s /= Void
    s /= Current
  do
    partner := s
    is_married := True
  ensure
    partner = s and is_married
end

invariant
  is_married implies
    partner /= Void and
    partner /= Current
end
Contracts: same example with AspectJ

```java
class Person {
    protected Person partner ;
    protected int age ;
    protected boolean isMarried ;

    setPartner (Person s) {
        // set partner with ‘s’
        partner = s; …
    }
    setAge (int a) {
        // set age with 'a'
        age = a
    }
    …
}

aspect PersonAssertion {
    // continuation - invariant
    pointcut invPerson (Person p) :
        target (p) && target (Person+) && call (* * (.))
        && !within (PersonAssertion);

    // continuation
    after (Person p, int a): // postcondition
        assertSetAge(p, a) {
            if (p.age != a)
                throw new IllegalArgumentException("…..");
        }
    …
}

An aspect per class hierarchy & a pointcut per assertion!
```
Contracts: reification and AspectJ

Two alternatives: Abstract or not

```java
aspect GenericContract {
    private JoinPoint currentJoinPoint;
    private void applyAssertion (String s) { … }

    pointcut checkToBePerformed () :
    target (Object+) && call(* * (..)) && !within (GenericContract) &&
    ! call(* inv_*(..)) && ! call(* pre_*(..)) && ! call(* post_*(..)) ;

    before() : checkToBePerformed () { currentJoinPoint = thisJoinPoint;
    applyAssertion (“inv_”); applyAssertion (“pre_”); }

    after() returning : checkToBePerformed () { currentJoinPoint = thisJoinPoint;
    applyAssertion (“post_”); applyAssertion (“inv_”); }
}
```

Much more complicated but quite generic
Contracts: reification and AspectJ

```java
class Person { // normal contents + assertions
    private boolean pre_setAge_1 (int arg) { return arg >= 0 && arg <= 100; }
    private boolean pre_setName_1 (String arg) { return arg != null; }
    private boolean pre_setPartner_1 (Person arg) { return arg != null; }
    private boolean post_setPartner_1 (Person arg) { return arg != this; }
    private boolean inv_Person_1 () { return this.age >= 0; }
    private boolean inv_Person_2 () {
        if (this.is_married && (this.partner == null || this.partner.partner != this))
            return false;
        else return true;
    }
    ...
}
```

*1st solution: Assertions are embedded in a class*
Contracts: reification and AspectJ

privileged Aspect PersonContract extends genericContract {

private boolean Person. Pre_setAge_1 (int arg) {identical }
private boolean Person.pre_setName_1 (String arg) {identical }
private boolean Person.pre_setPartner_1 (Person arg) {
     Caution: Features are added to class Person
private boolean Person.post_setPartner_1 (Person arg) {
private boolean Person.inv_Person_1 () {identical }
private boolean Person.inv_Person_2 () {identical }
...

2nd solution: Assertions are embedded in concrete aspect

} }

genericContract becomes an abstract aspect
Contracts: reification and AspectJ

```java
p1.setPartner(p2); p1.setAge(30);
```

Class Person

assertions

```java
p1.setPartner(p2); p1.setAge(30);
```

Static
- To insert assertion
- To adapt the pointcut

Dynamic
- For each method call:
  - To get target object & class
  - To get method signature
  - To find matching assertion
  - To perform the assertion

Aspect

- `GenericContract`
- `PersonContract`
Contracts: reification and AspectJ

```java
aspect genericContract { // Declaration

private void applyAssertion (String typeOfAssertion) {
    CodeSignature currentSignature; // Signature of the called method
    Method currentMethod; // Method corresponding to currentSignature
    Class currentClass; // Class containing currentMethod
    Class[] currentFormalArgs; // Formal arguments of currentMethod
    Object[] currentEffectiveArgs; // Effective arguments of the called method
    Object currentObject; // Object on which is called currentMethod
    Boolean currentCallResult; // Contains the result of the assertion execution

    …
}
```

✓ Source code: around 200 lines
✓ Here: some of the declarations
✓ Next: a piece of code applied on `setPartner`
currentEffectiveArgs = currentJoinPoint.getArgs();
currentObject = currentJoinPoint.getTarget();
currentSignature = (CodeSignature) currentJoinPoint.getSignature();
currentClass = currentSignature.getDeclaringType();
currentFormalArgs = currentSignature.getParameterTypes();
currentMethod = currentClass.getDeclaredMethod(currentSignature.getName(), currentFormalArgs);

While (!stop) {
    currentM = currentClass.getDeclaredMethod (currentName, new Class[]{})
    ReflectPermission x = new ReflectPermission("suppressAccessChecks");
    AccessibleObject.setAccessible (new AccessibleObject[] {currentM}, true);
    currentCallResult = (Boolean) currentM.invoke (currentObject, null);
    if (! currentCallResult.booleanValue())
        throw new IllegalArgumentException(currentMethod.getName() + "…");
    …} …} …} …}
Contracts and ASoC: summary

Benefits:
✓ Several models of assertion can be addressed
✓ A very expressive implementation platform

Drawbacks:
✓ Naming constraint / robustness of implementation
✓ Not straightforward to use / Specification purpose?
✓ Inheritance and assertions hard to implement

Assertion handling depending on user needs but requiring a set of tools for visualization assertions from the language
Contracts and ASoC: conclusion

To implement contracts as aspects?

- Contracts may be seen as a concern of the application
- But all concerns are not at the same level:
  - **Robustness** of application depends on it
  - Contracts requires some syntactical sugar
    (documentation, easy to use, …)

- Aspects could be used for implementation purpose only
Implementing check/debug

**First impression**: AspectJ does not bring any significant facility for inserting debug/check clauses into a routine code.

**Possible solutions**:

- **AspectJ**: To use the reification of the source code (line number within the file)

- **Java**: To insert a routine in the class + a call to it at the right place

  - *First solution is too complicated*
  - *Both are poorly readable*
Case-study n°2: Use a Design Pattern

Description:
• To adapt a library in order to use a design pattern
• Design pattern: Observer/Subject
• An interface handles the interactions between buttons and labels. When people click on a button the label should be modified (color change).

Context:
• Adding new functionalities to some library classes
• It depends on one type of application

Is ASoC a good alternative to use design patterns?
Design pattern Observer: UML view

<<interface>>
Subject

+attach(observer: Observer)
+detach(observer: Observer)
+notifyObservers()

<<interface>>
Observer

+update(o: Subject)

I_Subject

observers [*]: Observer

+attach(observer: Observer)
+detach(observer: Observer)
+notifyObservers()

<<description>>
forall o in observers
{
o.update (this);
}

22/04/2003
Philippe Lahire - team OCL - I3S
Laboratory UNSA / CNRS
Use of Design pattern: Example

**Example: EV_WIDGET**

**Example: EV_PRIMITIVE**

**Example: EV_LABEL**

**Example: EV_BUTTON**

**Example: EV_TOGGLE_BUTTON**

**Example: EV_RADIO_BUTTON**

**Subject**

**Observer**

**I_Subject**

**Update**

Creation of buttons (subject)

Creation of labels (observers)

User click on Buttons

Label color changes

Attach to label

---

22/04/2003

Philippe Lahire - team OCL - I3S
Laboratory UNSA / CNRS
Use of Design pattern: O-O answer

update (o: SUBJECT) is
local
c1: EV_COLOR
do
create c1.make (…)
set_foreground_color (c1)
end

b1: EV_BUTTON; l1: EV_LABEL ...
create {EV_RADIO_BUTTON2} b1
create {EV_LABEL2} l1
b1.attach (l1)
b1.click ...
The same drawbacks as explained (part I)

- Number of classes in the hierarchy may be increased
- Clients must use new name
- Composition of several hierarchies
- …

Or: changes in the source class itself …

Ability to reuse the library decreased?
Use of Design pattern: ASoC

```java
EV_BUTTON b1 = new EV_RADIO_BUTTON () ;
EV_LABEL l1 = new EV_LABEL () ;
b1.attach (l1) ; b1.click(); ...
```

Diagram:
- EV_WIDGET
- EV_PRIMITIVE
- EV_LABEL
- EV_BUTTON
- EV_RADIO_BUTTON
- EV_TOGGLE_BUTTON
- DesignPatternObserver
- abstract ProtocolOfObserver
- ROOT_CLASS
- ProtocolOfObserver1
- Observer*
- Subject*
Design pattern Observer: ASoC

import DesignPatternObserver;

public abstract aspect ProtocolOfObserver
{
    abstract pointcut specificStateModification();
    pointcut stateModification(Subject s) : target(s) && specificStateModification();
    after(Subject s) : stateModification(s) { s.notifyObservers(); }

    private Vector Subject.observers = new Vector();
    public void Subject.attach(Observer o) { observers.addElement(o); }
    public void Subject.detach(Observer o) { observers.removeElement(o); }
    public void Subject.notifyObservers()
    {
        for (int i = 0; i < observers.size(); i++) ((Observer) observers[i]).update(this);
    }
}
public aspect ProtocolOfObserver1 extends ProtocolOfObserver
{
    declare parents : EV_BUTTON implements Subject ;

    pointcut specificStateModification () : call (void EV_BUTTON.click () ) ;

    declare parents : EV_LABEL implements Observer ;

    public void EV_LABEL.update (Subject s) {
        EV_COLOR c = new EV_COLOR (...);
        set_foreground_color (c);
    }
}
ASoC Answer: summary

• Does not increase the number of classes of the hierarchy
• Modification are specified outside of the class (*aspects*)
• A large part of adaptation is shared (*abstract aspect*)
• Only application specific part is repeated (*concrete aspects*)
• Clients can use the same name
• Composition of several hierarchies is possible
• Code for adaptation is mixed with application code
• …

*Ability to reuse the library is increased*
Case-study n°3: Handling of tests

Description:
• Handling capture/replay of events in graphical libraries
• To be able to activate/deactivate capture/replay
• Test automation (execution, logs, generation)

Context:
• Adding new functionalities to some classes of library
• It depends on one type of application

Is ASoC a good alternative to implement tests?
Testing techniques

• **A wide subject**
  • Tracking bugs / Deviations from specification
  • Granularity, reification and storage
  • Several categories of test (scope/intent categorization)
  • …

• **Objectives of next slides**
  • Scope categorization: unit testing & regression testing
  • Intent categorization: white-box & fault-based testing
  • Small example to show how to implement with ASoC
Handling Capture/replay of events

ACTION_SEQ • call • extend

EV_APP • launch

EV_APP_I

EV_APP_IMP • loop_message • process_message

EV_WINDOW • process_message

WEL_WINDOW • process_message

WEL_APP

WEL_MSG • peek_all • dispatch

WEL_DISPATCH • window_procedure

Eiffelvision on Windows

EV_WINDOW I

EV_WINDOW_IMP • process_message

EV_APP • launch

EV_WINDOW • process_message

EV_APP_I

EV_APP_IMP • loop_message • process_message

WEL_WINDOW • process_message

WEL_APP

WEL_MSG • peek_all • dispatch

WEL_DISPATCH • window_procedure

EV_WINDOW_I

EV_WINDOW_IMP • process_message
Handling Capture/replay of events

Trying to understand Eiffelvision event handling (Windows):

- Initialisation of graphic objects and user actions generate events
- Application object receipts all events and then dispatch them
  - message_loop: “to get message” and then “to handle message”
  - use MSG_WEL routines: peek_all, dispatch
- window_procedure (call-back): to dispatch events related to one WEL_WINDOW (in WEL_DISPATCHER)
- Call process_message: send one message to one graphic object with its two coordinates (in WEL_WINDOW)

It looks quite centralized
Handling Capture/replay of events

Possible approach:
- To implement a flag: *capture / replay / normal use*
- All work is at one location *EV_APP_IMP and suppliers*
- To add a pieces of code *around and after* *peek_all*

Capture:
- To take info from current object (mainly *MSG_WEL*)
- To store information in DB *after peek_all*

Replay:
- To retrieve information from DB, to put it in *MSG_WEL* object
- To *bypass peek_all*
import eiffelVision;
public aspect EventCaptureReplayProtocol
{
    private int EV_APP_IMP.action = normal_use;
    public void EV_APP_IMP.set_normal() { action = normal_use; }
    public void EV_APP_IMP.set_capture() { action = capture; }
    public void EV_APP_IMP.set_replay() { action = replay; }

    pointcut eventCaptureReplay(WEL_MSG m, EV_APP_IMP a):
        target(m) && this(a) call (void WEL_MSG.peek_all()) && ( 
            withincode (void WEL_APP+.message_loop()) || 
            withincode (void EV_APP_IMP.process_events()) 
        );
Handling Capture/replay of events

// Continuation

private Database db = new Database();

after (WEL_MSG m, EV_APP_IMP a) : eventCaptureReplay(m, a) {
    if (action == capture) // store information in database
        db.set_hwnd(m.get_hwnd()); ... // same for all information to store
}

around (WEL_MSG m, EV_APP_IMP a) : eventCaptureReplay(m, a) {
    if (action == replay) {
        m.set_hwnd(db.get_hwnd()); ... // same for all stored information
    } else proceed(m, a); // normal call to peek_all
}

22/04/2003

Philippe Lahire - team OCL - I3S
Laboratory UNSA / CNRS
Automation of test execution: summary

**OO Programming**:
- Insertion using inheritance (new descendant): In this case it looks fine
  - `EV_APP(LICATION)_IMP` is a leaf
  - `EV_APP_IMP` never used by clients other than `eiffelvision` itself

**AO Programming**:
- Capture / replay is also encapsulated at one place
  - Open to more sophisticated test implementation
  - Access to “process_message” in descendant of `WEL_WINDOW`

OOP is sufficient but AOP is more open
Conclusion & Perspectives

• Maintenance:
  – Pointcut, advices, facility to add declaration
  – Powerful language for code adaptation

• Adding of new services:
  – Most should be carried out by syntactical add-on: contract, debugging, concurrency, etc.
  – Fit for specific services but not for others

• AOP for Application Maintenance

• Syntactical add-on for better readability
Perspectives

• To keep the O-O language semantics and syntax for treatment handling
  – Is it necessary to distinguish maintenance code from original using special keyword (*aspect*)?
  – Is it necessary to get some other capability?

• To delegate to “Maintenance/configuration” language adaptation of class hierarchy
  – Do we need all the power of AspectJ?
  – As soon as modifications are stable: application should be rebuilt … *In theory!*