Quelques réflexions sur la programmation post objet.

P. Coindre
Obasco Group
coindre@emn.fr

NOTERE 2007
Marrakech, 8 juin 2007

U. Eco, L’œuvre ouverte

« L’œuvre d’art est un message fondamentalement ambigu, une pluralité de signifiés qui coexistent en un seul signifiant (…). Pour réaliser l’ambiguïté comme valeur, les artistes contemporains ont souvent recours à l’informel, au désordre, à l’indétermination des résultats. On est ainsi tenté d’établir une dialectique entre forme et ouverture, qui déterminerait dans quelle limite une œuvre d’art peut accentuer son ambiguïté et dépendre de l’intervention active du spectateur sans perdre pour autant sa qualité d’œuvre. »

« Objects have failed ? »
Dick Gabriel
Remarks

From Lisp to Smalltalk
« Objects, as envisioned by the designers of languages like Smalltalk and Actors were for modeling and building complex, dynamic worlds. Programming environments for languages like Smalltalk were written in those languages and were extensible by developers. Because the philosophy of dynamic change was part of the post Simula worldview, languages and environments of that era were highly dynamic. »

From Eiffel/C++ to Java
« But with C++ and Java, the dynamic thinking fostered by OOL was nearly fatally assaulted by the theology of static thinking inherited from our mathematical heritage and the assumptions built into our views of computing by C. Babbage whose factory building worldview was dominated by omniscience and omnipotence. »
Form vs Opening (closure vs adaptability)

- The difference between classes and objects has been repeatedly emphasized. In the view presented here, these concepts belong to different worlds: the program text only contains classes at run-time, only objects exist. This is not the only approach. Object-oriented programming, advocated by Yip and exemplified by Smalltalk, views classes as object templates, which still have an existence at run-time. The view taken on the design of Eiffel is that the distinction between descriptions (of processes or classes of objects) and the corresponding execution is a fundamental one, and that any confusion should be avoided, in the same way modern computer techniques do not encourage self-modifying programs (even though a real clever idea). – B. Meyer

Some lessons from OOP

- **metaobject protocols (MOP)**
  - protocols to open the implementation (introspection and intercession) but meta level architectures are expensive to use, difficult to understand and to secure (structural and behavioral reflection).
- **design patterns**
  - no direct representation at the code level (traceability)
- **frameworks**
  - no real support for unanticipated extensions (adaptability)
- **classes**
  - Inheritance is not THE solution for reusing cross-cutting “modules”, see Java colorable, movable, paintable, clonable, serializable, activable interfaces.

Meta Object Protocols

**Structural reflection**
- object creation
  - allocate or initialize
    - constructors
    - metaclasses
    - prototypes
- class modification
  - adding/removing a field
  - changing the class hierarchy

**Behavioral reflection**
- message sending
  - lookup or apply
    - error handling
    - class based inheritance
    - prototype & delegation
    - encapsulators
    - wrappers & proxies
public Object receive (String selector, Object[] args) {
    Method mth = null; Object r = null; Class[] classes = null; int l = 0;
    if (args != null) {
        l = Array.getLength(args);
        classes = new Class[l];
    }
    for (int i = 0; i < l; i++) {
        classes[i] = args[i].getClass();
    }
    try {
        mth = getClass().getMethod(selector, classes); // lookup before pointcut
        r = mth.invoke(this, args); // apply-proceed after pointcut
    } catch (Exception e) {System.out.println(e);}
    return r;
}
Design Patterns

The composite pattern (AWT)

Revisiting the Smalltalk MVC design pattern

A need to introduce a new mechanism (dependencies) to couple the code of the model, its view and its controller.

The dependencies mechanism prefigures event programming. The principle is to connect a source object with a set of listeners. Those listeners will be notified as soon as the source will emit a new event.
The Counter class

Counter (value)

value

value: anInteger
value ← anInteger.

incr: anInteger
self value: value + anInteger
self incr: 1
raz
self value: 0

The CounterViews class

Counter (value)

value

value: anInteger
value ← anInteger.
self changed
incr: anInteger
self value: value + anInteger
self incr: 1
raz
self value: 0

The CounterController class

Controller (model view sensor)
MouseMenuController (redButton..)
CounterController() {
    YMenu YMessage}
initialize
super initialize.
self
yellowButtonMenu: YMenu
yellowButtonMessages: YMessage
isControlActive
super isControlActive &
sensor blueButtonPressed not
/Frameworks
Intuition of a Java Clock (S. Chiba)

Implementing the Clock intuition by reusing

- the Java GUI (AWT / Swing) frameworks
- the Java thread framework

The standard Java Clock

```java
public class Clock {
    static Thread myThread = new Thread();
    static int QUANTUM = 1000;
    public static void main(String[] args) {
        myThread.start();
        while (true) {
            try {
                myThread.sleep(QUANTUM);
            } catch (InterruptedException e) {
            }
            System.out.println(new Date());
        }
    }
}
```

```java
public class ClockApplet extends AppletWithThread {
    public void init() {
        setBackground(java.awt.Color.blue);
        setForeground(java.awt.Color.yellow);
    }
    public void paint(java.awt.Graphics g) {
        int xc = (getSize().width) / 2;
        int yc = (getSize().height) / 2;
        g.drawString(new Date(), xc, yc);
    }
}
```
Classes ++

More about classes

- Play too many roles and there if some confusion around those concerns:
  - object generators,
  - method dispatchers,
  - parts of the inheritance graph.
- There is no intermediate granularity between a method and a class. No reification of:
  - a sub set of methods (category),
  - a set of classes (application/package),
  - a design pattern as a set of related classes.
- This leads to current evolutions such as:
  - traits and mixin modules,
  - classboxes, open classes and Envy applications.

The Smalltalk (hidden) class model

The Trait model
A class use several traits

![Diagram of Traits Composable Core of Behaviour](image)

Some lessons from Middleware

- Applications typically need multiple services
  - logging, tracing, profiling, locking, displaying, transporting, authentication, security...
- These services don’t naturally fit in usual module boundaries, they are **crosscutting**
  - These services must be called from many places: **code scattering**.
  - An individual operation may need to refer to many services: **code tangling**.

JBoss = EJB + Xcutting services

Associating technical services to EJB components and composing those services on demand:

- **persistence**
- **security**
- **caching**
- **logging**
- **remote**
- **...**

"We give you the hooks to simply specify the interceptor stack you want for a given component."

Scalability and modularity issues.
Code tangling and code scattering

Considering the org.apache.tomcat application and the three next concerns:

- XML parsing
- URL pattern matching
- Logging

Thanks to G. Kiczales
Good modularity

XML parsing in org.apache.tomcat
- Red shows relevant lines of code
- Nicely fits in one box

Good enough modularity

URL pattern matching in org.apache.tomcat
- Red shows relevant lines of code
- Nicely fits in two boxes (using inheritance)

Modularity problems

Logging in org.apache.tomcat
- Red shows lines of code that handle logging
- Not in just one place: a good example of code scattering
- Not even in a small number of places

Tyranny of the primary decomposition

Visiting the Leiden’s NM of Antiquities

“Egyptian depictions consist of strange combinations of side views, frontal views and views from above….
Every part of the representation should be as clear as possible.”
Understanding aspects

Quoting M. Wand:

“Aspect-Oriented Programming is a promising recent technology for allowing adaptation of program units across modules boundaries.”

Aspect intuition (U. Aßmann)

Domain Specific Languages
Two alternatives

- Expressing every aspect with an "aspectual domain specific language" (ADSL).
- Using the same general purpose language (e.g. Java) for the aspects and the base program.

Aspect Oriented Programming (AOP)

- Behavioral AOP : the dynamic Join Point Model
- Structural AOP : the Inter-type declarations (introductions)

Some AspectJ definitions

- Join points : principled points in the dynamic execution of the program (e.g points at which an object receives a method call and points at which a field of an object is referenced).
- Pointcut : a set of join points that optionally exposes some of the values in the execution context of those join points.
- Advice : a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut.
- Inter-type declarations (Introductions) : declarations of members that cut across multiple classes or declarations of change in the inheritance relationship between classes.
Some definitions

Aspects: are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a form similar to that of class declarations. Aspect declarations may include pointcut declarations, advice declarations, inter-type declarations as well as all other kinds of declaration permitted in class declarations.

Notice that:
- AspectJ aspects are implemented as « singleton classes ».
- AspectJ Aspects can be specialized.

Picking join points:
- method call
- method execution
- constructor call
- initializer execution
- static initializer execution
- constructor execution
- object preinitialization
- object initialization
- field reference get/set
- exception handling execution

Set-based semantics for pointcuts

- && operator = logical and = set intersection
- || operator = logical or = set union
- ! operator = logical negation = set difference

Advices

Are executed when a given pointcut is reached. The associated code can be told to run:
- before executing the "instruction" at the join point
- after executing the "instruction" at the join point
  - after returning
  - after throwing
  - plain after (after returning or throwing)
- around instead of executing the "instruction" at the join point
  - proceed
Tiny Examples

<table>
<thead>
<tr>
<th>FP</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>demoing</td>
<td></td>
</tr>
<tr>
<td>memoizing</td>
<td></td>
</tr>
<tr>
<td>incr(int)</td>
<td>tracing</td>
</tr>
<tr>
<td>setValue(int)</td>
<td></td>
</tr>
<tr>
<td>incr()</td>
<td></td>
</tr>
<tr>
<td>raz()</td>
<td></td>
</tr>
<tr>
<td>decr()</td>
<td></td>
</tr>
</tbody>
</table>

```java
public class FP {
    static boolean pair(int n) {
        if (n == 0)
            return true;
        else
            return impair(n - 1);
    }
    static boolean impair(int n) {
        if (n == 0)
            return false;
        else
            return pair(n - 1);
    }
    public static void main(String[] args) {
        System.out.println("pair(4)-- >" + pair(4));
        System.out.println("pair(3)-- >" + pair(3));
    }
}
```

```java
public aspect Demo {
    pointcut callpair(): call(static boolean FP.pair(int));
    pointcut callimpair(): call/static boolean FP.impair(int));
    before(int n): callpair() && args(n) {
        System.out.println("Before callpair(" + n + ")");
    }
    after() returning (boolean b) : callpair() {
        System.out.println("After callpair(" + b + ")");
    }
    before(int n): callimpair() && args(n) {
        System.out.println("Before callimpair(" + n + ")");
    }
    after() returning (boolean b) : callimpair() {
        System.out.println("After callimpair(" + b + ")");
    }
}
```

```java
public class FP {
    static boolean pair(int n) {
        if (n == 0)
            return true;
        else
            return impair(n - 1);
    }
    static boolean impair(int n) {
        if (n == 0)
            return false;
        else
            return pair(n - 1);
    }
    public static void main(String[] args) {
        System.out.println("pair(4)-- >" + pair(4));
        System.out.println("pair(3)-- >" + pair(3));
    }
}
```

```java
public aspect Demo {
    pointcut callpair(): call(static boolean FP.pair(int));
    pointcut callimpair(): call/static boolean FP.impair(int));
    before(int n): callpair() && args(n) {
        System.out.println("Before callpair(" + n + ")");
    }
    after() returning (boolean b) : callpair() {
        System.out.println("After callpair(" + b + ")");
    }
    before(int n): callimpair() && args(n) {
        System.out.println("Before callimpair(" + n + ")");
    }
    after() returning (boolean b) : callimpair() {
        System.out.println("After callimpair(" + b + ")");
    }
}
```
public class FP {
    public static int fact(int n) {
        if (n == 0)
            return 1;
        else return n * fact(n - 1);
    }
    public static int fib(int n) {
        if (n <= 1)
            return 1;
        else
            return fib(n - 1) + fib(n - 2);
    }
    public static void main(String[] args) {
        fact(5); fact(4); fact(6);
        fib(3);
    }
}

abstract aspect TraceProtocol {
    int counter = 0;
    abstract pointcut trace();

    before(int n): trace() && args(n){
        counter++;
        for (int j = 0; j < counter; j++) {System.out.print(" ");}
        System.out.println("-->f(" + n + ")");
    }
    after(int n) returning (int r): trace() && args(n){
        for (int j = 0; j < counter; j++) {System.out.print(" ");}
        System.out.println(r + "<--f(" + n + ")");
        counter--;
    }
}

public aspect Trace extends TraceProtocol {
    pointcut trace() :
        call(static int FB.fact(int)) ||
        call(static int FP.fib(int)) ||
        call(void Counter.incr(int));
}

Execution traces (fact & fib)

-->f(5)
-->f(4)
-->f(3)
-->f(2)
-->f(1)
-->f(0)
1<--f(0)
1<--f(1)
2<--f(2)
6<--f(3)
24<--f(4)
120<--f(5)
public aspect Memoization {
    public static HashMap memo = new HashMap();
    pointcut callfact(): call(int FP.fact(int)) ;

    int around(int n): callfact() && args(n){
        Integer N = new Integer(n);
        if (memo.containsKey(N)) {
            return ((Integer)memo.get(N)).intValue();}
        else {
            int r = proceed(n) ;
            memo.put(N, new Integer(r));
            return r;
        }
    }
}

public class FP {
    public static int fact(int n){
        if (n == 0)
            return 1;
        else return n * fact(n - 1);
    }

    public static int fib(int n){
        if (n <= 1)
            return 1;
        else
            return fib(n-1) + fib(n-2);
    }

    public static void main(String[] args) {
        fact(5);fact(4);fact(6);
        fib(3);
    }
}

Execution traces (factorial)

-->fact(5)
-->fact(4)
-->fact(3)
-->fact(2)
-->fact(1)
1 <-- fact(0)
1 <-- fact(1)
2 <-- fact(2)
6 <-- fact(3)
24 <-- fact(4)
120 <-- fact(5)

**Intuition of a Java Clock**

```java
public aspect Concurrency {
    pointcut executeMain() : execution(static void Clock.main(String[])) ;

    void around() : executeMain() {
        new Thread()
        {
            public void run() {
                System.out.println("Started in another thread");
                proceed();
            }
        }.start();
    }
}
```

**Behavioral vs structural crosscutting**

Intercessing with the program behavior
- user-defined and primitive pointcuts
- cflow

Changing the program structure
- introducing field
- introducing method
- introducing constructor
- modifying class hierarchy

**Aspectisation of the Observer DP**
Extending/Modifying a class

- Field introduction
  - (modifiers) type ClassName.addedFieldName [= expression];
  - private int Point.z=0;
  - private Subject Observer.subject = null;

- Method introduction
  - public int Circle.getRed(){...}

- Constructor introduction

- Modifying the class hierarchy
  - declare parents: ClassName extends BaseClassName
  - declare parents: ClassName implements InterfaceNamesList

Java Counters

class Counter extends Object {
    private int value;
    public int getValue() {
        return value;
    }
    public void setValue(int nv) {
        value = nv;
    }
    public String toString() {
        return "@" + value;
    }
    public void incr() {
        this.incr(1);
    }
    public void incr(int delta) {
        this.setValue(value + delta);
    }
    public void raz() { // reset
        this.setValue(0);
    }
}

Defining an abstract aspect monitoring state change

abstract aspect PatternObserverProtocol {

    // update logic
    abstract pointcut stateChanges(Subject s):

    after(Subject s): stateChanges(s) {
        for (int i = 0; i < s.getObservers().size(); i++) {
            ((Observer)s.getObservers().elementAt(i)).update();
        }
    }

    // registration logic (2 next slides)
Adapting classes implementing the Subject Interface

```java
private Vector Subject.observers = new Vector();
public void Subject.addObserver(Observer obs) {
    observers.addElement(obs);
    obs.setSubject(this);
}
public void Subject.removeObserver(Observer obs) {
    observers.removeElement(obs);
    obs.setSubject(null);
}
public Vector Subject.getObservers() { return observers; }
```

Adapting classes implementing the Observer interface

```java
private Subject Observer.subject = null;
public void Observer.setSubject(Subject s) {subject = s;}
public Subject Observer.getSubject() {return subject; }
```

aspect CounterObserver extends PatternObserverProtocol {

    declare parents: Counter implements Subject;
    declare parents: Printer implements Observer;

    public void Printer.update() {
        this.print("update occured in" + this.getSubject());
    }

    pointcut stateChanges(Subject s) :
        target(s) &&
        call(public void Counter.setValue(int));
}

main in Counter

```java
public static void main(String[] args) {
    Counter c1 = new Counter();
    Printer scribe = new Printer();
    c1.raz();
    c1.addObserver(scribe);
    c1.incr();
    c1.incr(6);
    c1.raz();
}
```

-->f(1)
[Printer]update occured in Counter@1
-->f(6)
[Printer]update occured in Counter@7
[Printer]update occured in Counter@0
Thèses récentes sur le sujet

- Pierre-Charles David (Nantes 05) : Développement de composants Fractal adaptatifs : un langage dédié à l’aspect adaptation.
- Stéphane Hong Tuan Ha (Rennes 07) : Programmation par aspects et tissage de propriétés. Application à l’ordonnancement et à la disponibilité.
- Nicolas Pessemier (Lille 07) : unification des approches par aspects et par composants (Fractal)
- Simon Denier (Nantes 07) : aspectisation de schémas de conception. Application à l’étude de la densité des motifs dans JHotDraw.

Interesting discussions at :

- G. Kiczales
  - http://www.cs.ubc.ca/~gregor/
- M. Wand
  - http://www.ccs.neu.edu/home/wand/
- R. Gabriel
- J.-F. Perrot’s group
- AOSD conference and European Network On Excellence
  - http://www.aosd.net/
  - http://www.aosd-europe.net/
Papers : AspectJ’s friends

- **Composition filter**

- **HyperJ**

- **Demeter et DemeterJ**
  - K. Lieberherr and D. Lorenz. Coupling Aspect-Oriented and Adaptive Programming, AOSD 2004 Book Chapter, Addison Wesley

Papers : formal works

- **Semantics**

- **Aspect calculus**

- **Aspect calculus + type safe advices**

- **Grammar manipulation**
  - Lämmel. PEPM’99: Declarative AOP.