De l'évolution des langages de programmation

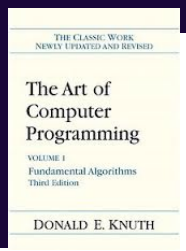
Pierre Cointe
LINA
cointe@emn.fr

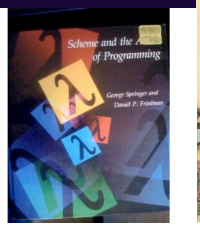
JFSMA'2014

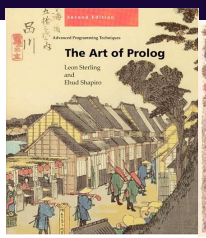
Loriol, le 10 octobre

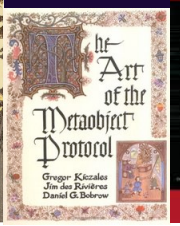
Art de la programmation

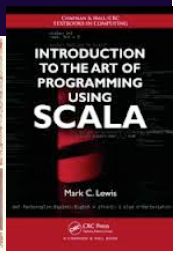
- Dialectique entre forme et ouverture (pouvoir d'expression)
  - Métaprogrammation
  - Langages réflexifs : réification & réflexion, introspection & intercession
  - Architectures logicielles ouvertes
  - Efficacité et sécurité
- Dialectique entre minimalité et uniformité
  - « Less is More » (Mies van der Rohe)
  - « La perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer » (Antoine de Saint-Exupéry)
  - Parcimonie et extensibilité
- Dialectique entre généralisation et spécialisation
  - Langages généralistes
  - Langages dédiés à un domaine
  - Langages pour les machines/objets et langages pour les gens
  - Séparation des préoccupations (tyrannie de la décomposition primaire)
- Dialectique entre modélisation et programmation
  - Quoi/What (spécification/besoins) versus Comment/How (implémentation/stratégies)
  - « Dont Design your programs, Program your designs ! » (William Cook)
  - « Program First, Think Later » (Peter Deutsch)
  - « Programmation au lancer » (Jean-Louis Durieux)











Forme versus ouverture

U. Eco : Opera Aperta

« 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 dialectiques 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. »

La PPO : un pivot dans l'évolution des langages de programmation

- Modèle (Simula) ☞ Objet
- Procédure ☞ Fermeture ☞ Funarg (« λ as first class object »)
- Funarg☞ Objet (Smalltalk) et Acteur (Plasma)
- Type abstrait (Clu) ☞ Objet
- Frame ☞ Objet ☞ Agent
- Objet ☞ Prototype
- Objet ☞ Objet concurrent ☞ Object actif ☞ Acteur ☞ Agent
- Objet ☞ Classe ☞ Trait
- Objet ☞ MétaObjet ☞ Aspect
- Objet ☞ Design Pattern
- Objet ☞ Modèle (UML) ☞ MétaModèle
- Objet ☞ Composant ☞ Service

λ-calculus, Lisp and the Funarg problem
☞ closure, actor and class

```
(define (fact n cont)
  (if (= n 0)
    (cont 1)
    (fact (- n 1)
      (λ (res)
        (cont (* res n))))))
```

(fact 0 (λ (x) x)) ☞ 1

(fact 5 (λ (x) x)) ☞ 120

(fact 3 (λ (x) (* x x)) ☞ 36

```
(define (cons a b)
  (λ (selector . args)
    (case selector
      (first? a)
      (rest? b)
      (class? (quote cons))
      (else (error ...)))))
(define (car l) (l 'first))
(define (cdr l) (l 'rest))
```

(define unCons (cons 'do 're))

(unCons 'first?) ☞ do

(unCons 'rest) ☞ re

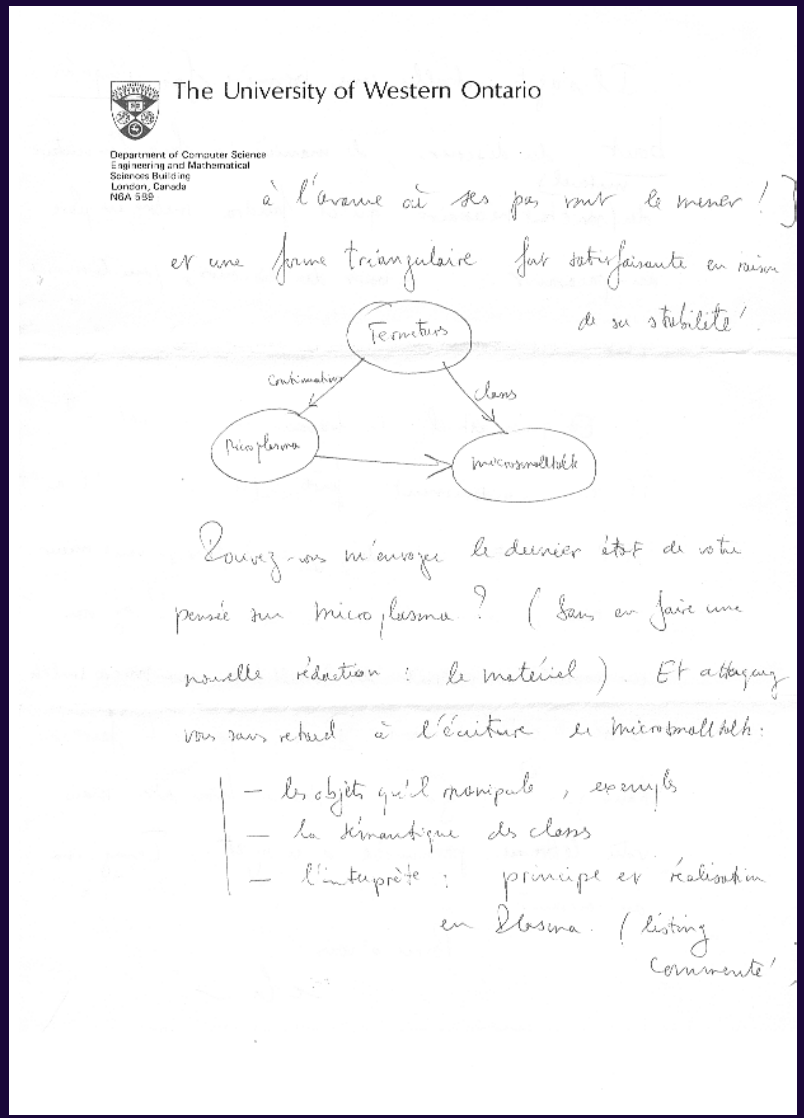
Des fermetures au objets (79/81)

Lambda as a passed argument :

- Continuation
- Hewitt's actors
- Micro-plasma

Lambda as a computed result :

- Instantiation
- Kay's classes
- Micro-smalltalk



« Objects have failed ?» D. Gabriel

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 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 ».

Smalltalk : uniformité et régression infinie

- P1 : chaque entité est un objet
- P2 : chaque objet est instance d'une classe

  ☞ Une classe est un objet instance d'une classe appelée une métaclasse,
  ☞ Une métaclasse est un objet instance d'une classe ….
  ☞ Le bootstrap impose de stopper la régression infinie : la racine du graphe d'instanciation est (co)instance d'elle même

p1 := Point new

p1 class ☞ Point

Point class ☞ Point class

Point class class ☞ Metaclass

Point class class class ☞ Metaclass class

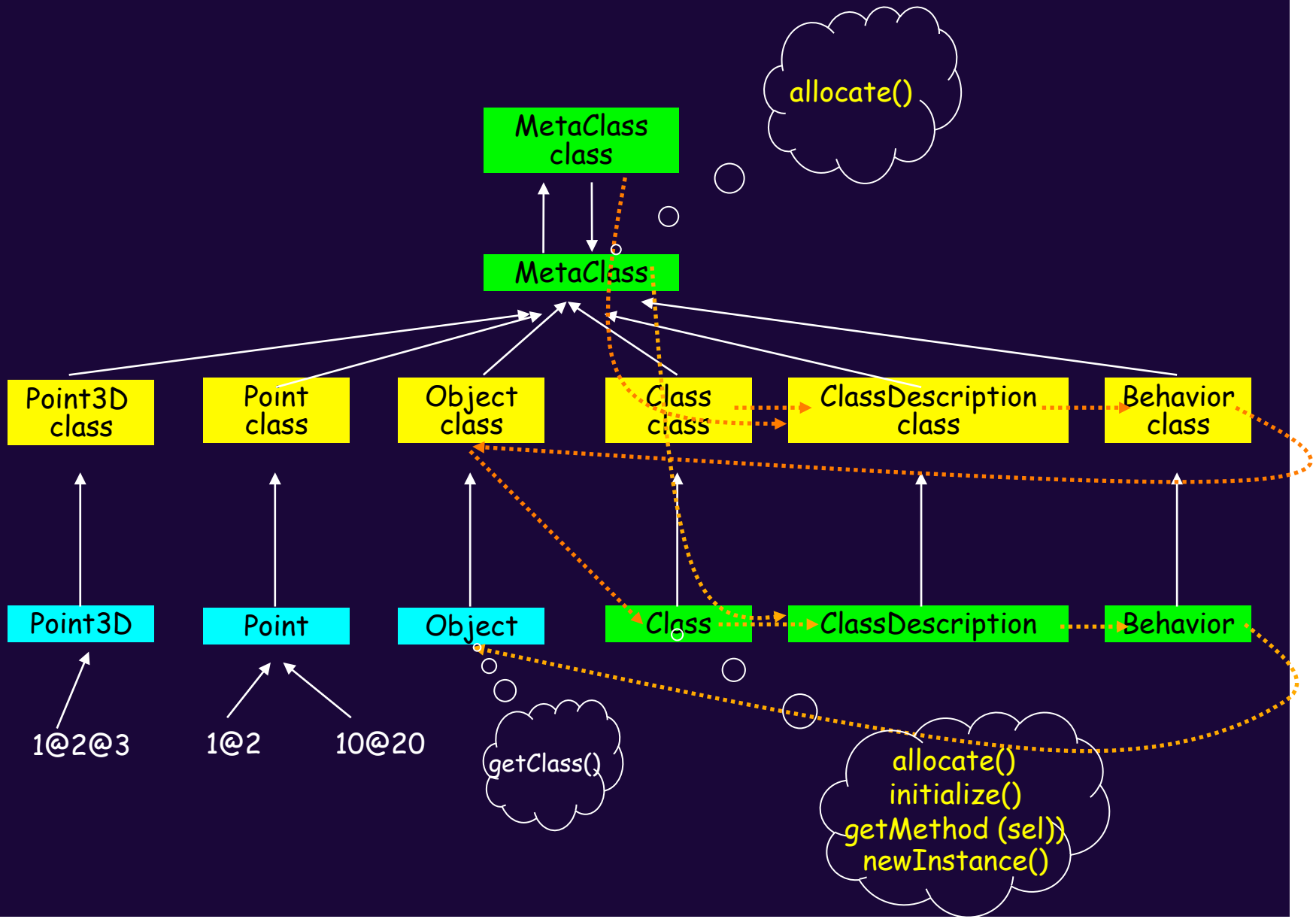
Point class class class class ☞ Metaclass

Point class class class class class ☞ Metaclass class

Contradiction !

- Pour cacher à l'utilisateur-λ l'existence des méta-classes et de la régression infinie associée celles-ci sont anonymes. Elles sont crées automatiquement au moment de la définition d'une classe qui devient l'unique instance de sa méta-classe privée.
- Ce choix de masquer le méta-niveau conduit à une architecture inutilement complexe dont la compréhension est réservée aux seuls programmeurs experts.

Le modèle (caché) des classes Smalltalk



Separation Of Concerns in Smalltalk

« Smalltalk uses classes to describe the common properties of related objects. Unfortunately, the use of classes is the source of a number of complications... One source of the complexity surrounding classes in Smalltalk is the interaction of message lookup with the role of classes as the generators of new objects, which gives rise to the need for metaclasses. »

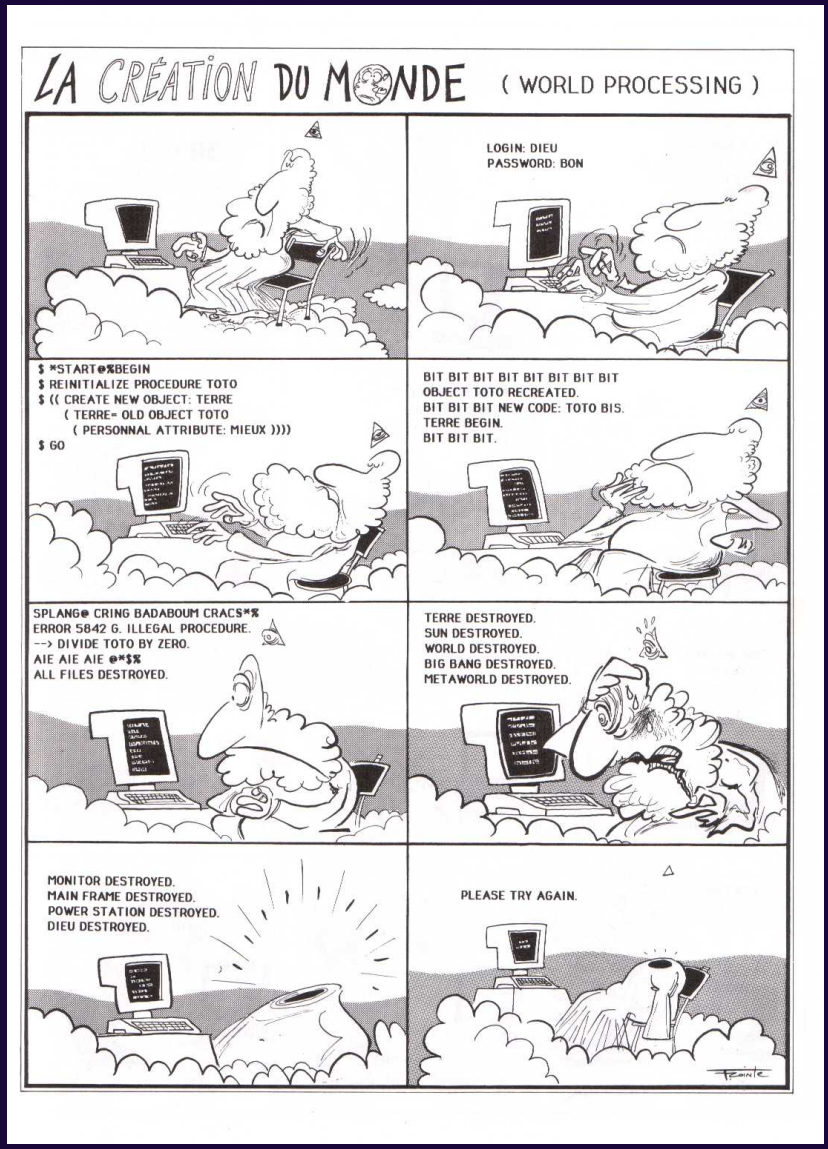
The classes play too many roles and introduce some confusion around those concerns :

1. Object generators
2. Method containers and dispatchers

Then metaclasses are hiden to the end users!

Eiffel and self-modifying programs

« 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 contain classes; at run-time, only objects exist. This is not the only approach. One of the subcultures of object-oriented programming, influenced by Lisp and exemplified by Smalltalk, views classes as object themselves, 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 executions 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

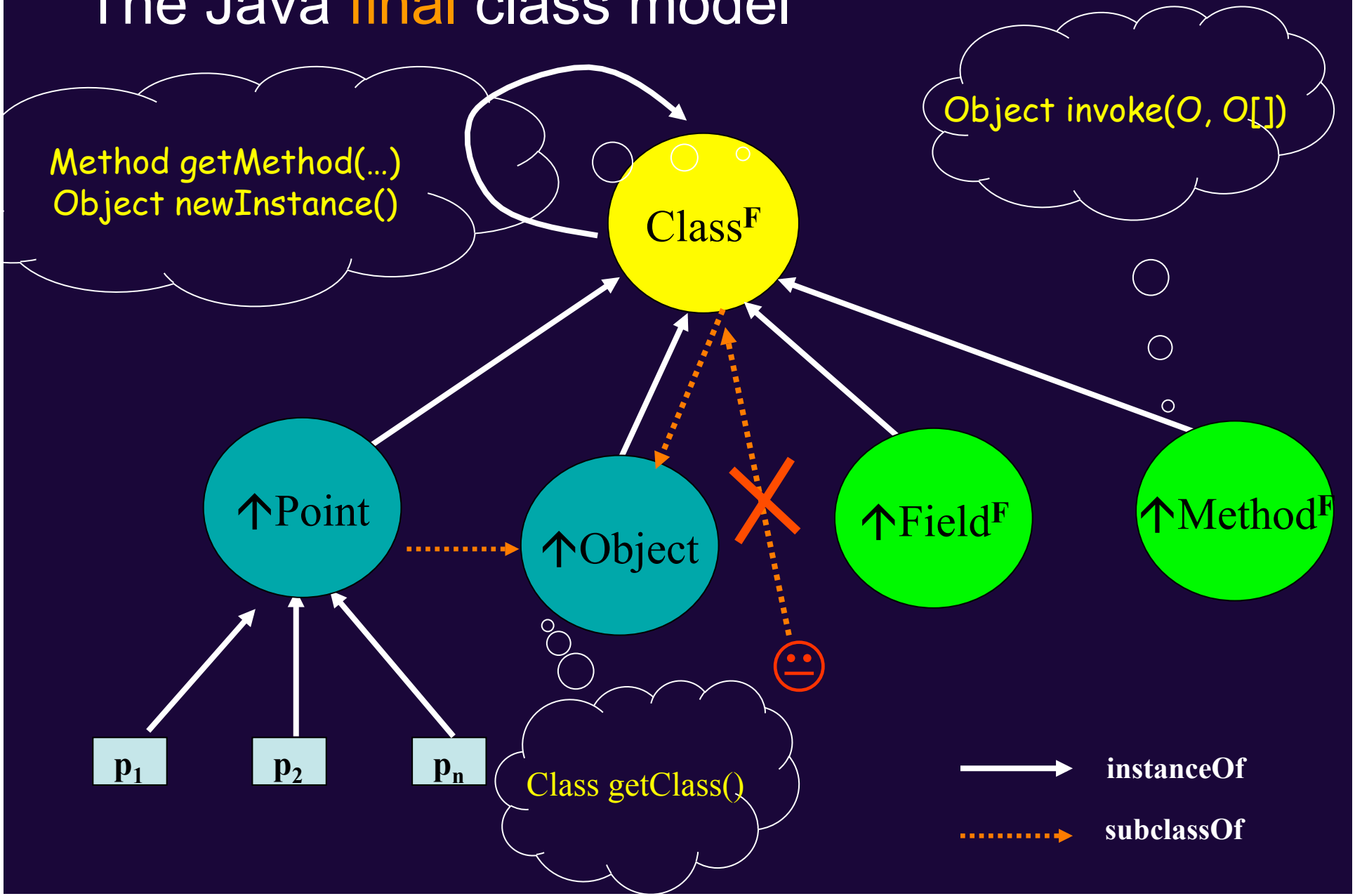


Java and introspection

« The Java Core Reflection API provides a small, type-safe, and secure API that supports introspection about the classes and objects in the current Java Virtual Machine. If permitted by security policy, the API can be used to :

- o construct new class instances and new arrays,
- o access and modify fields of objects and classes,
- o invoke methods on objects and classes,
- o access and modify elements of arrays. »

The Java final class model

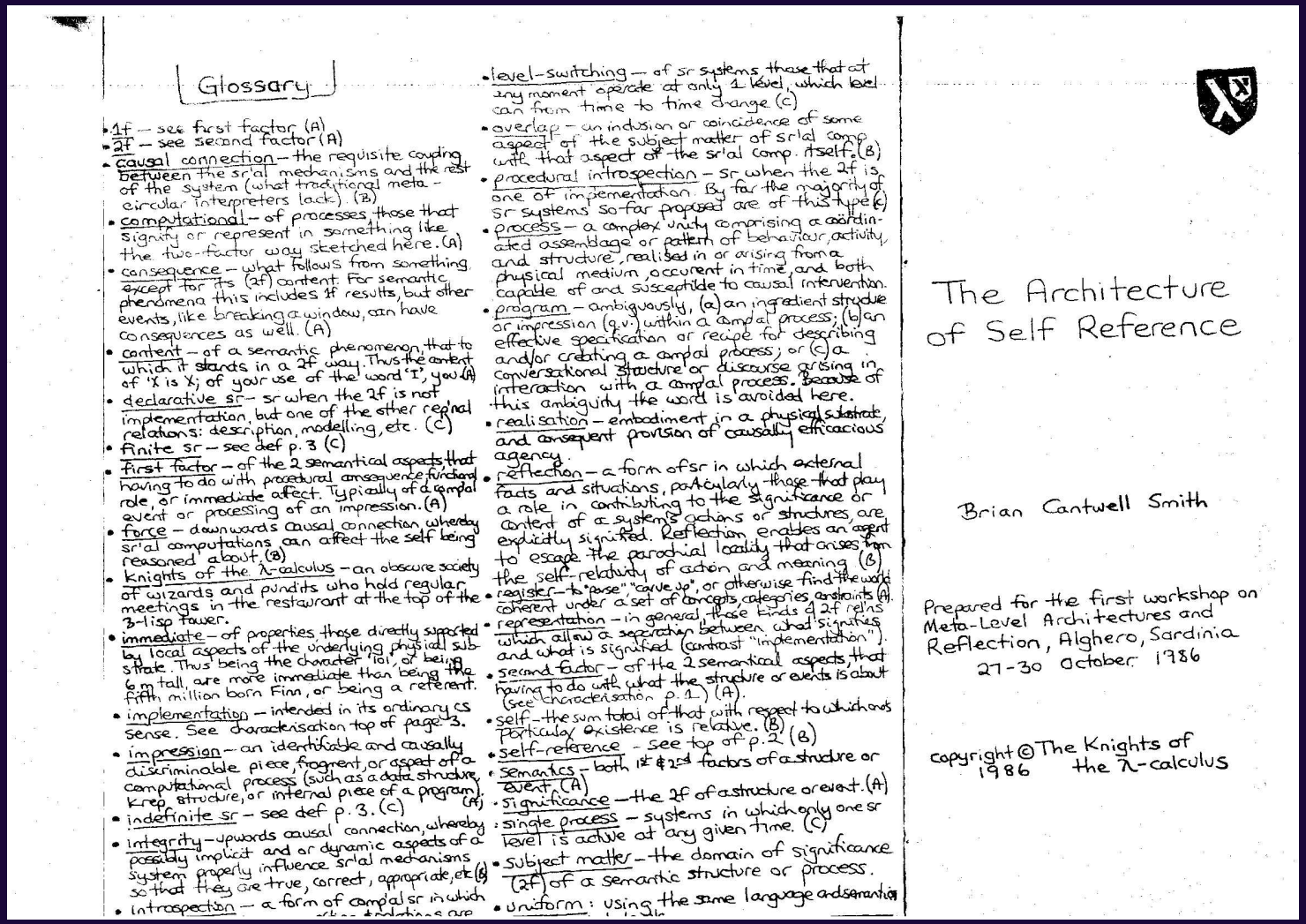


```
public Object receive (String selector, Object[] args) {
   Method mth = null; Object r = null; Class[] classes = null;  int l = 0;
   if (args != null) {
      l = Array.getLength(args);
      classes = new Class[l];
   }
   for (int i = 0; i < l; i++) { classes[i] = args[i].getClass();}
   try {
      mth = getClass().getMethod(selector, classes); // lookup
      // before poincut
      r = mth.invoke(this, args);   // apply-proceed
      // after pointcut
      } catch (Exception e) {System.out.println(e);}
   return r;
}
```

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 / patrons de conception
  - no direct representation at the code level (traceability)
- frameworks / cadriciels
  - 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.

The Architecture of Self Reference. B.-C. Smith 86



Glossary

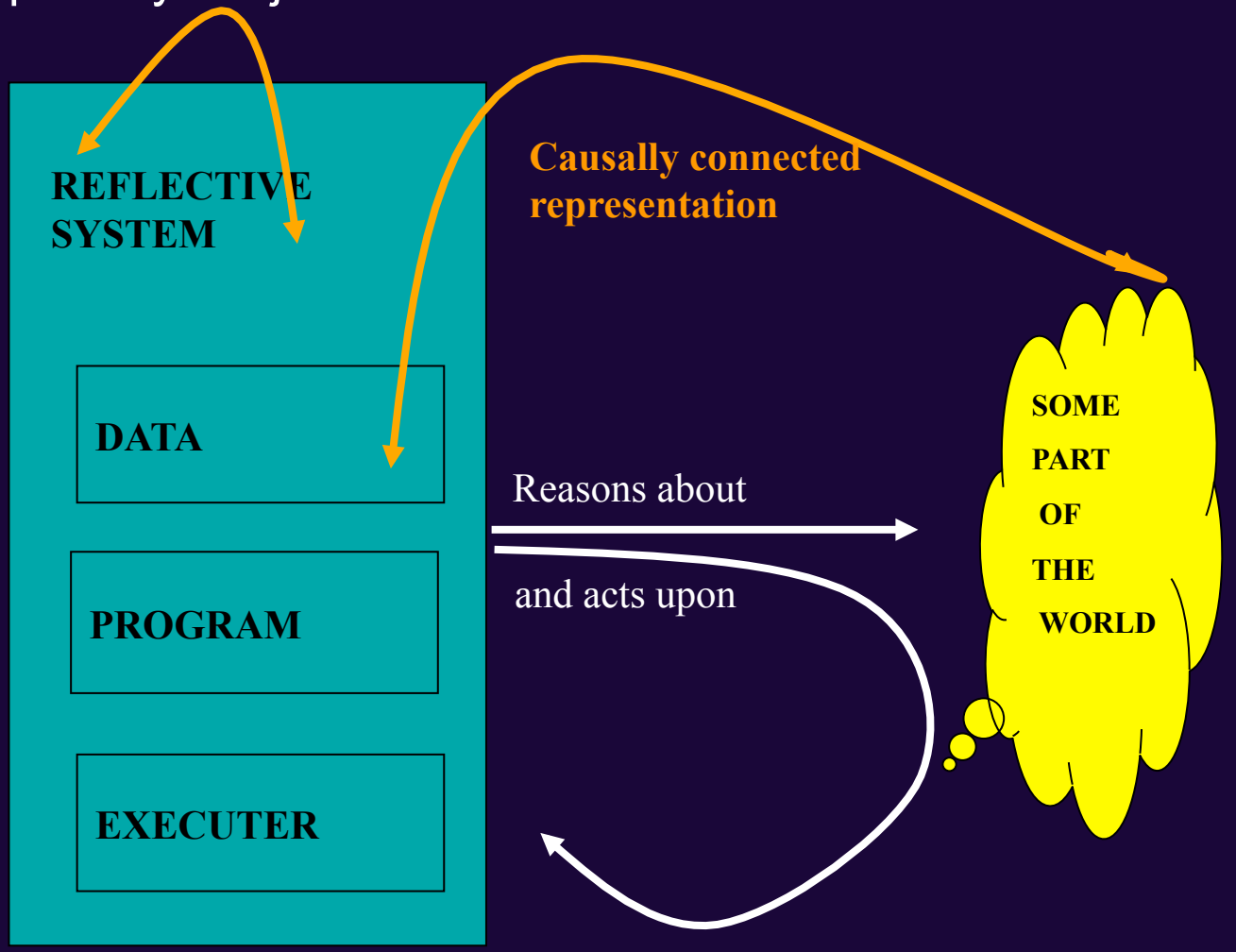
- 1f – see first factor (A)
- 2f – see second factor (A)
- causal connection – the requisite coupling between the sr'al mechanisms and the rest of the system (what traditional meta-circular interpreters lack). (B)
- computational – of processes, those that signify or represent in something like the two-factor way sketched here. (A)
- consequence – what follows from something, except for its (2f) content. For semantic phenomena this includes 1f results, but other events, like breaking a window, can have consequences as well. (A)
- content – of a semantic phenomenon, that to which it stands in a 2f way. Thus the content of 'X is Y;' of your use of the word 'I', you (A)
- declarative sr – sr when the 2f is not implementation, but one of the other rep'nal relations: description, modelling, etc. (C)
- finite sr – see def p. 3 (C)
- first factor – of the 2 semantical aspects that having to do with procedural consequence, functional role, or immediate affect. Typically of a comp'al event or processing of an impression. (A)
- force – downwards causal connection, whereby sr'al computations can affect the self being reasoned about. (B)
- knights of the λ-calculus – an obscure society of wizards and pundits who hold regular meetings in the restaurant at the top of the 3-lisp tower.
- immediate – of properties, those directly supported by local aspects of the underlying physical substrate. Thus being the character 'a', or being 6 m tall, are more immediate than being the fifth million born Finn, or being a referent.
- implementation – intended in its ordinary CS sense. See characterisation top of page 3.
- impression – an identifiable and causally discriminable piece, fragment, or aspect of a computational process (such as a data structure, k-rep structure, or internal piece of a program). (A)
- indefinite sr – see def p. 3. (C)
- integrity – upwards causal connection, whereby possibly implicit and or dynamic aspects of a system properly influence sr'al mechanisms so that they are true, correct, appropriate, etc. (B)
- introspection – a form of comp'al sr in which
- level-switching – of sr systems, those that at any moment operate at only 1 level, which level can from time to time change. (C)
- overlap – an inclusion or coincidence of some aspect of the subject matter of sr'al comp. with that aspect of the sr'al comp. itself. (B)
- procedural introspection – sr when the 2f is one of implementation. By far the majority of sr systems so far proposed are of this type. (C)
- process – a complex unity comprising a coordinated assemblage or pattern of behaviour, activity, and structure, realised in or arising from a physical medium, occurent in time, and both capable of and susceptible to causal intervention.
- program – ambiguously, (a) an ingredient structure or impression (q.v.) within a comp'al process; (b) an effective specification or recipe for describing and/or creating a comp'al process; or (c) a conversational structure or discourse arising in interaction with a comp'al process. Because of this ambiguity the word is avoided here.
- realisation – embodiment in a physical substrate, and consequent provision of causally efficacious agency.
- reflection – a form of sr in which external facts and situations, particularly those that play a role in contributing to the significance or content of a system's actions or structures, are explicitly signified. Reflection enables an agent to escape the parochial locality that arises from the self-relativity of action and meaning. (B)
- register – to "parse", "carve up", or otherwise find the words coherent under a set of concepts, categories, constraints (A)
- representation – in general, those kinds of 2f relns which allow a separation between what signifies and what is signified (contrast "implementation"). (see characterisation p. 1.) (A)
- second factor – of the 2 semantical aspects, that having to do with what the structure or event is about
- self – the sum total of that with respect to which ons particular existence is relative. (B)
- self-reference – see top of p. 2. (B)
- semantics – both 1st & 2nd factors of a structure or event. (A)
- significance – the 2f of a structure or event. (A)
- single process – systems in which only one sr level is active at any given time. (C)
- subject matter – the domain of significance (2f) of a semantic structure or process.
- uniform: using the same language and semantics

The Architecture of Self Reference

Brian Cantwell Smith

Prepared for the first workshop on Meta-Level Architectures and Reflection, Alghero, Sardinia 27-30 October 1986

« Reflection : a process's integral ability to represent, operate on, otherwise deal with itself in the same way that it represents, operates and deals with its primary subject matter »

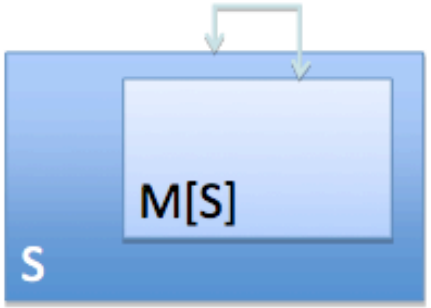


Leonard Cohen « I am your man »

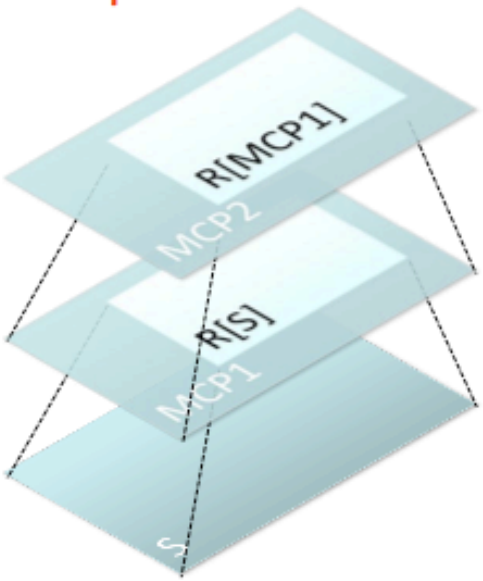
« They sentenced me twenty years of boredom for trying to change the system for within. I'am coming now, I'am coming to reward them. First we take Manhattan, then we take Berlin. »

Computational Reflecfion
Thanks to Yonezawa (Dalh-Nygaard prize 08)

Computation about Oneself: Introspection & Self Modification



A reflective system S can reason about or act upon itself via the causally-connected self-representation M[S]=Model of S.



MCPs: meta-interpreters, meta-objects

base-level

Pioneers:
3-Lisp (B. C. Smith, 1982)
3-KRS (P. Maes, 1986)

Representation in a Reflective Tower (Smith, 1982)
S is reified as R[S] within the meta-circular processor MCP1. MCP1 is also reified in MCP2, and so forth.
Reflective behaviors are realized as normal operations in the meta-levels (MCPs).

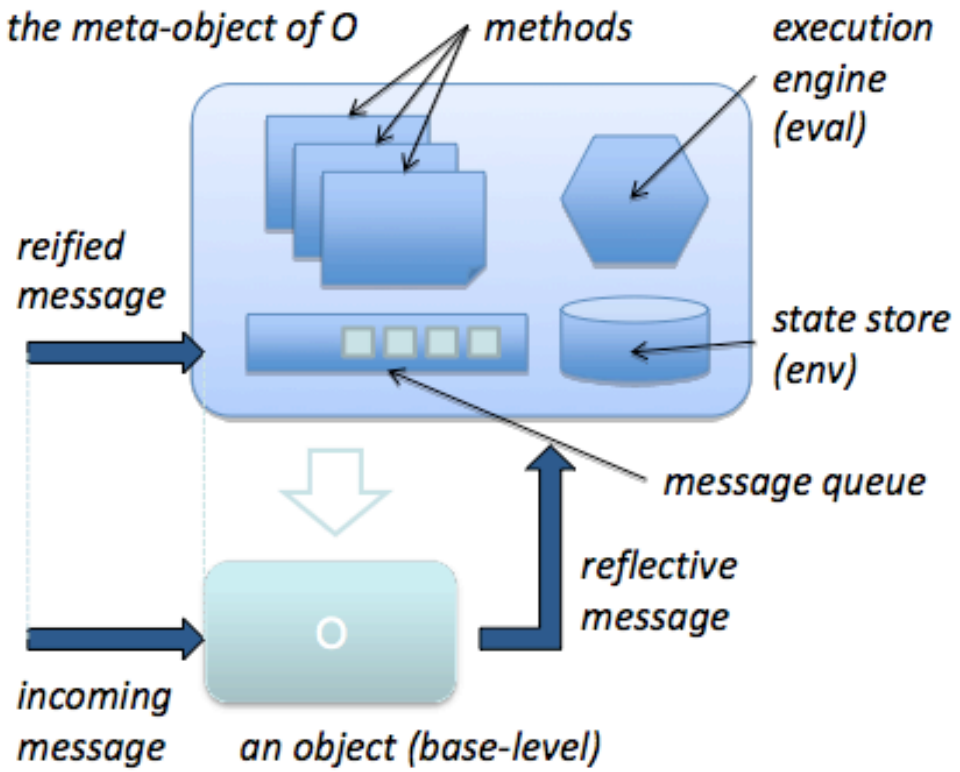
Computational Reflecfion
Thanks to Yonezawa (Dahl-Nygaard prize 08)

One Concurrent Object A –> One Meta-Object (Model of A)

Each concurrent object has its own meta-object that reifies its entire structure and solely governs its computation.

The meta-object is a 1st class object and thus has its meta-object. This implies that the reflective tower exists for every object.

Any object can send messages to its meta-object. Reflective behaviors are realized with such inter-level messages.



Meta Object Protocols

Structural reflection

- object creation

  allocate o initialize

  - constructors
  - metaclasses
  - prototypes

- class modification
  - adding/removing a field
  - changing the class hierarchy

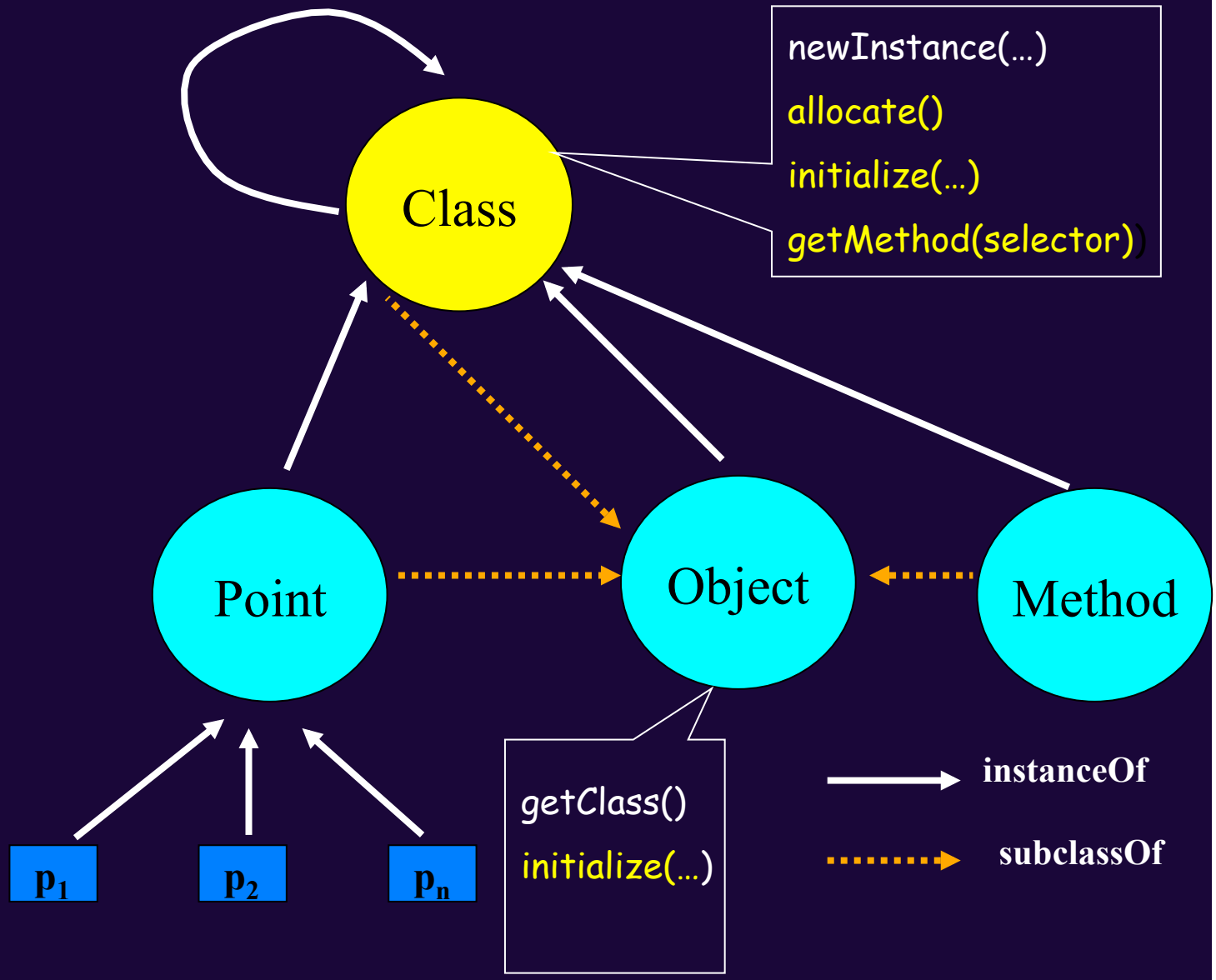
Behavioral reflection

- message sending

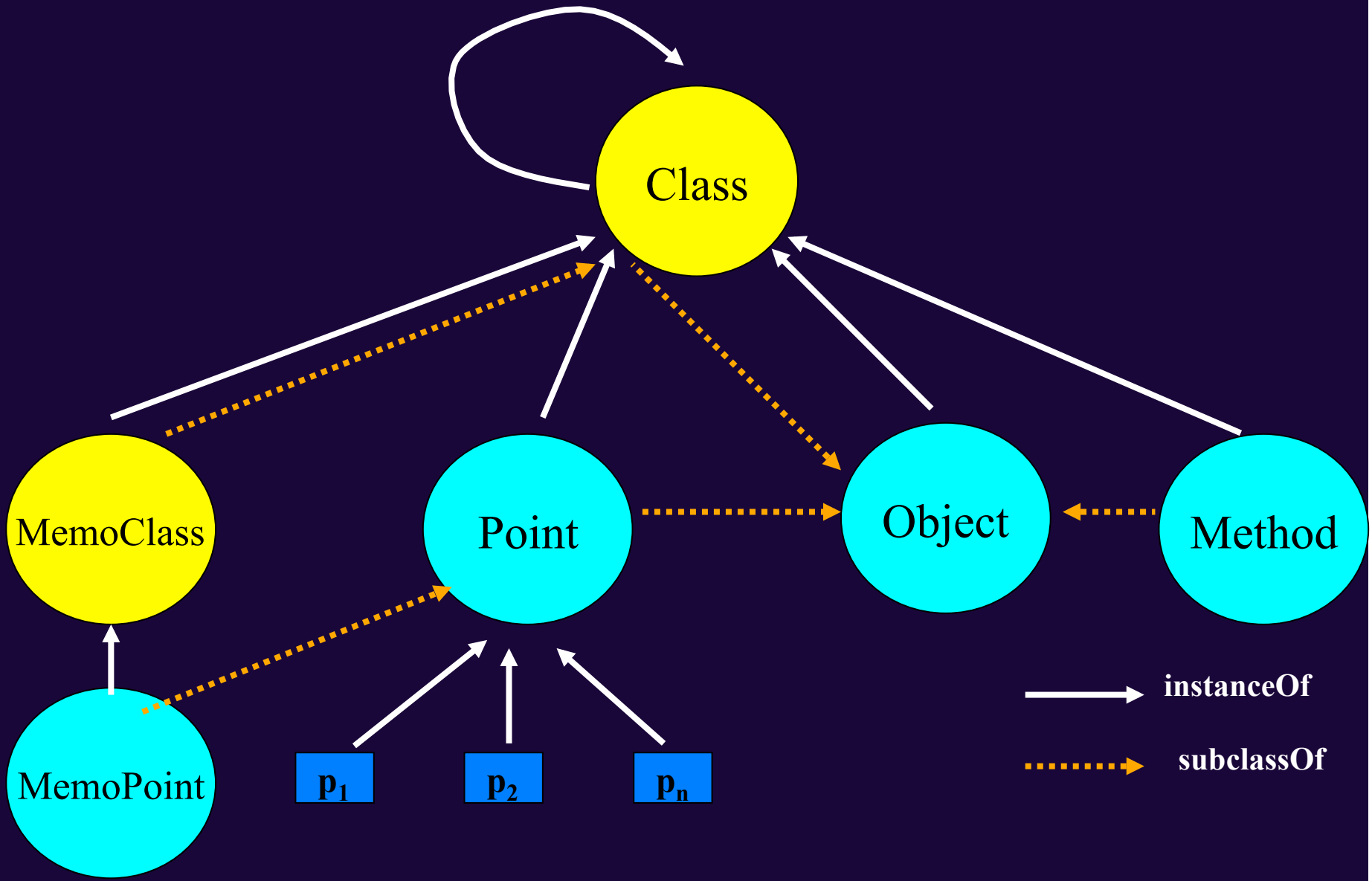
  lookup o apply

  - error handling
  - class based inheritance
  - prototype & delegation
  - encapsulators
  - wrappers & proxies

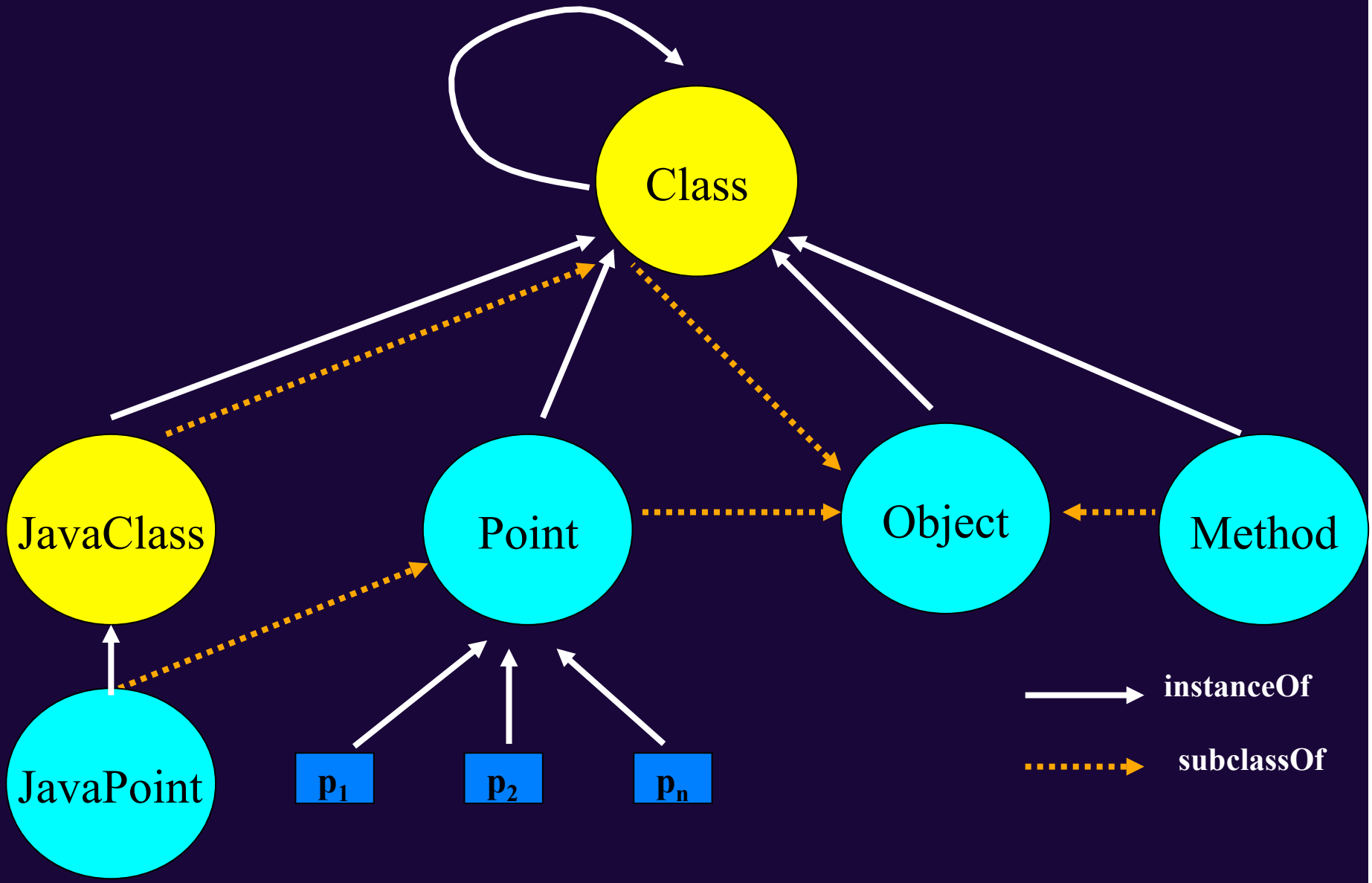
ObjVlisp a minimal model (IRCAM 1984)



ObjVlisp Self-Extension

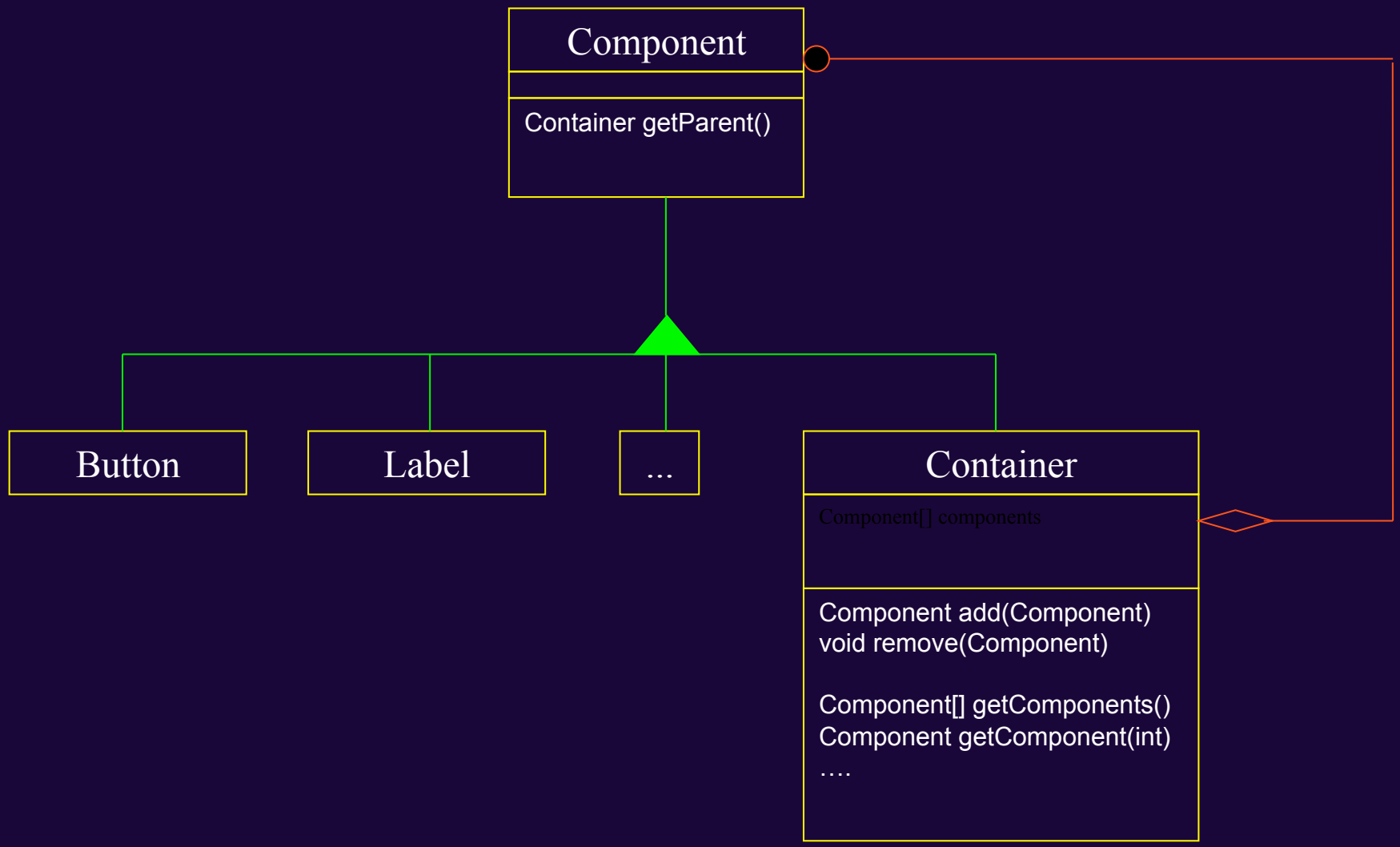


ObjVlisp Self-Extension



Design Patterns

Le patron Composite (AWT)



Les objets ont gagné ! (W. Cook)

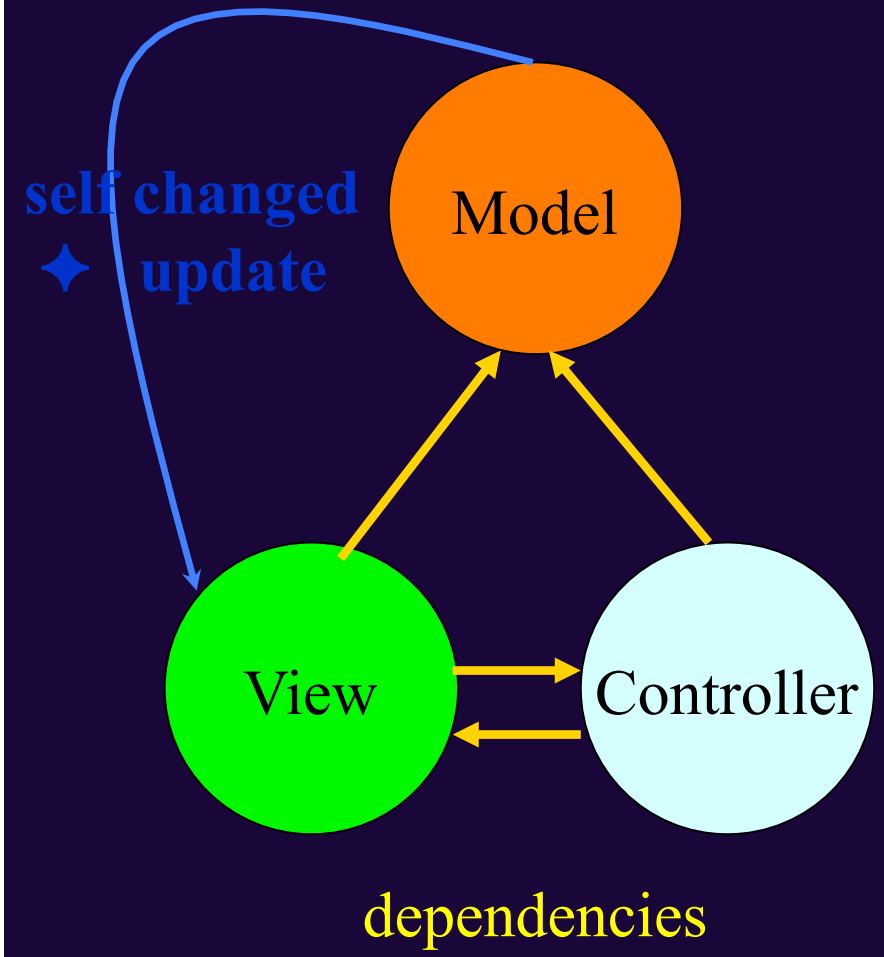
Killer application = les cadriciels pour GUI

écrire des classes (et les mettre en relations)

les instancier pour créer un modèle de GUI

exécuter le modèle

Le patron MVC design revisité



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 objet with a set of listeners. Those listeners will be notified as soon as the source will emit a new event.

La classe Counter (Smalltalk)

```
Counter (value)
-----------------------------------------------
value
   ↑value
value: anInteger
   value ← anInteger.

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

```
Counter (value)
-----------------------------------------------
value
   ↑value
value: anInteger
   value ← anInteger.
   self changed
incr: anInteger
   self value: value + anInteger
incr
   self incr: 1
raz
   self value: 0
```

La classe CounterView

```
Counter (value)
-----------------------------------------
value
  ↑value
value: anInteger
  value ← anInteger.
  self changed
incr: anInteger
  self value: value + anInteger
incr
  self incr: 1
raz
  self value: 0
```

Join point introduction to notify the observers

```
View (model controller)
  CounterView()
-----------------------------------------
defaultControllerClass
  ↑CounterController

update: dummy
  super displayView

displayView
  model value printString displayAt:
       insetDisplayBox center
```

advice

Frameworks

Intuition of a Java Clock (S. Chiba)

```
class Clock {
  static void main(String[] args) {
    while (true) {
      System.out.println(currentTime());
      sleep(ONE_MINUTE);
    }
  }
```

Implementing the Clock intuition

Reusing :

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

Une horloge Java

```
class Clock {
  static void main(String[] args) {
    while (true) {
      System.out.println(currentTime());
      sleep(ONE_MINUTE);
    }
  }
```

```
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());
      }
   }
}
```

Wed Nov 24 10:13:50 CET 2004

```
public class ClockApplet extends AppletWithThread {
    public void init() {
      setForeground(java.awt.Color.blue);
      setBackground(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 Trait model
Composable Units of Behavior

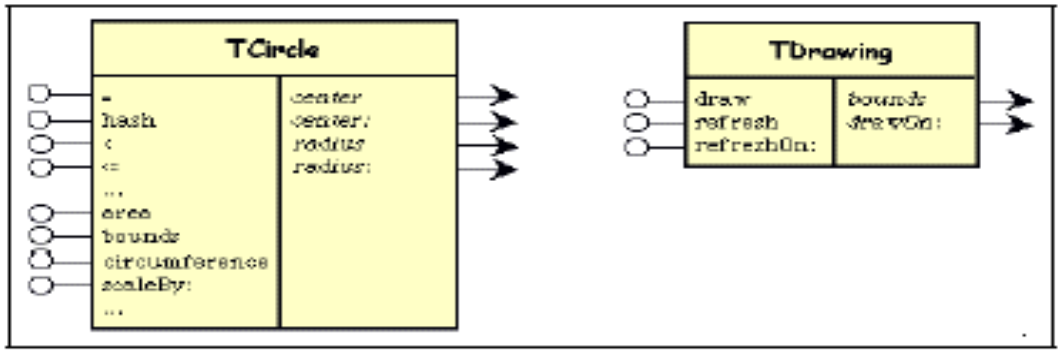


Fig. 3. The traits TDrawing and TCircle with provided methods in the left column and required methods in the right column.

A class use several traits : Pharo & Scala

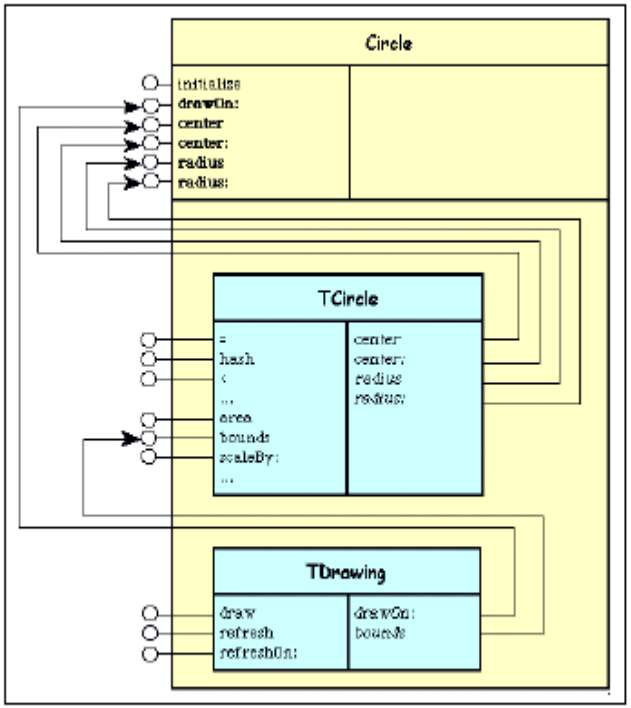
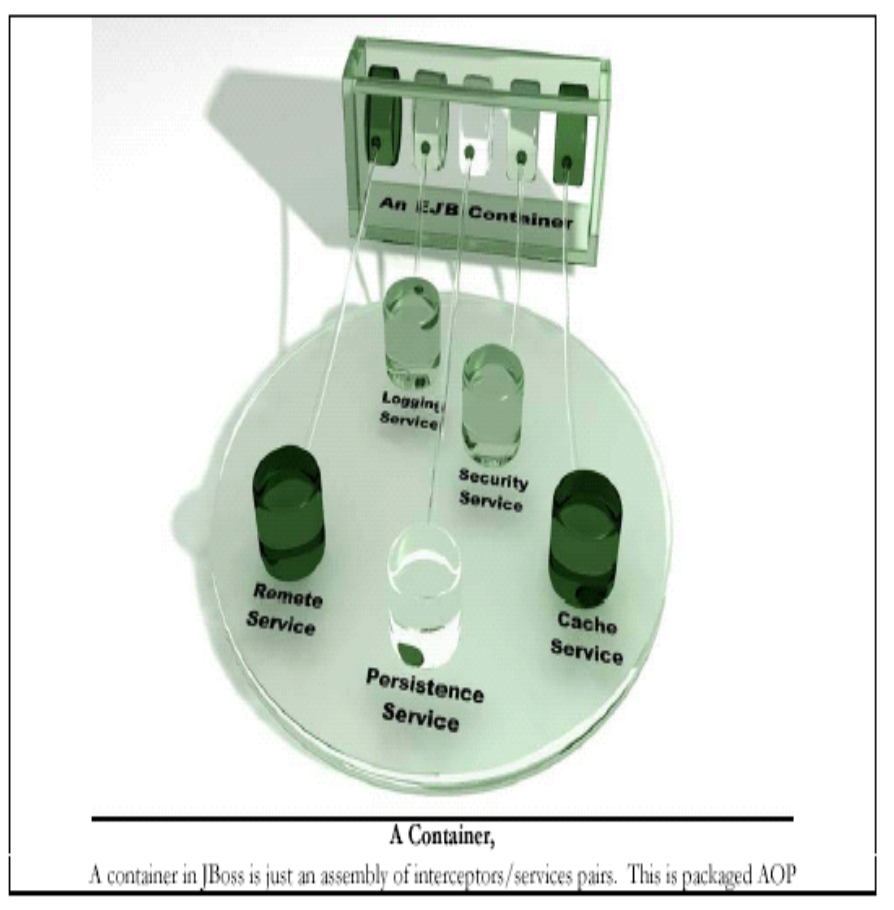


Fig. 4. The class Circle is composed from the traits TCircle and TDrawing. The requirement for TDrawing>>bounds is fulfilled by the trait TCircle. All the other requirements are fulfilled by accessor methods specified by the class.

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

Code entremêlé et code diffus

Considering the org.apache.tomcat application

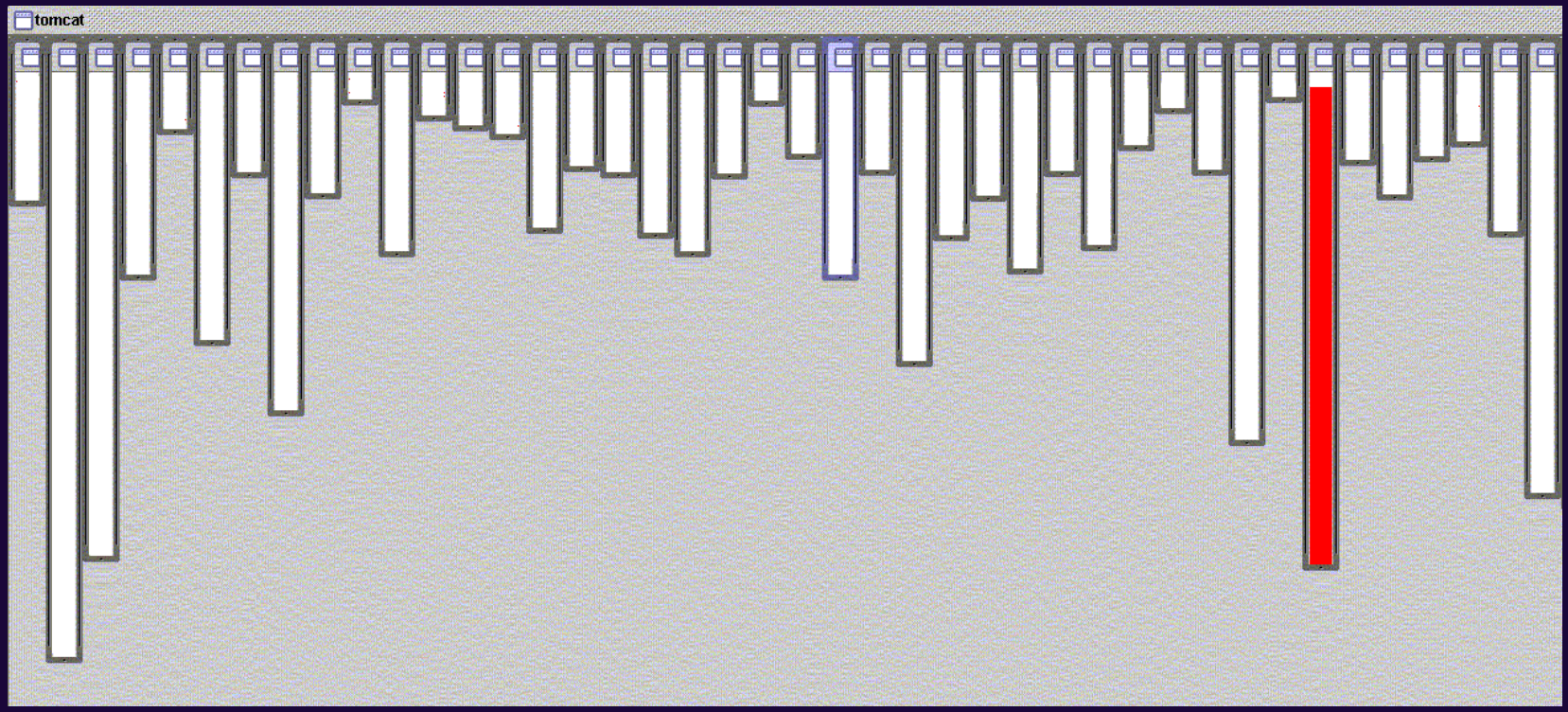
and the three next concerns:

- XML parsing
- URL pattern matching
- Logging

Thanks to G. Kiczales

Good modularity

XML parsing

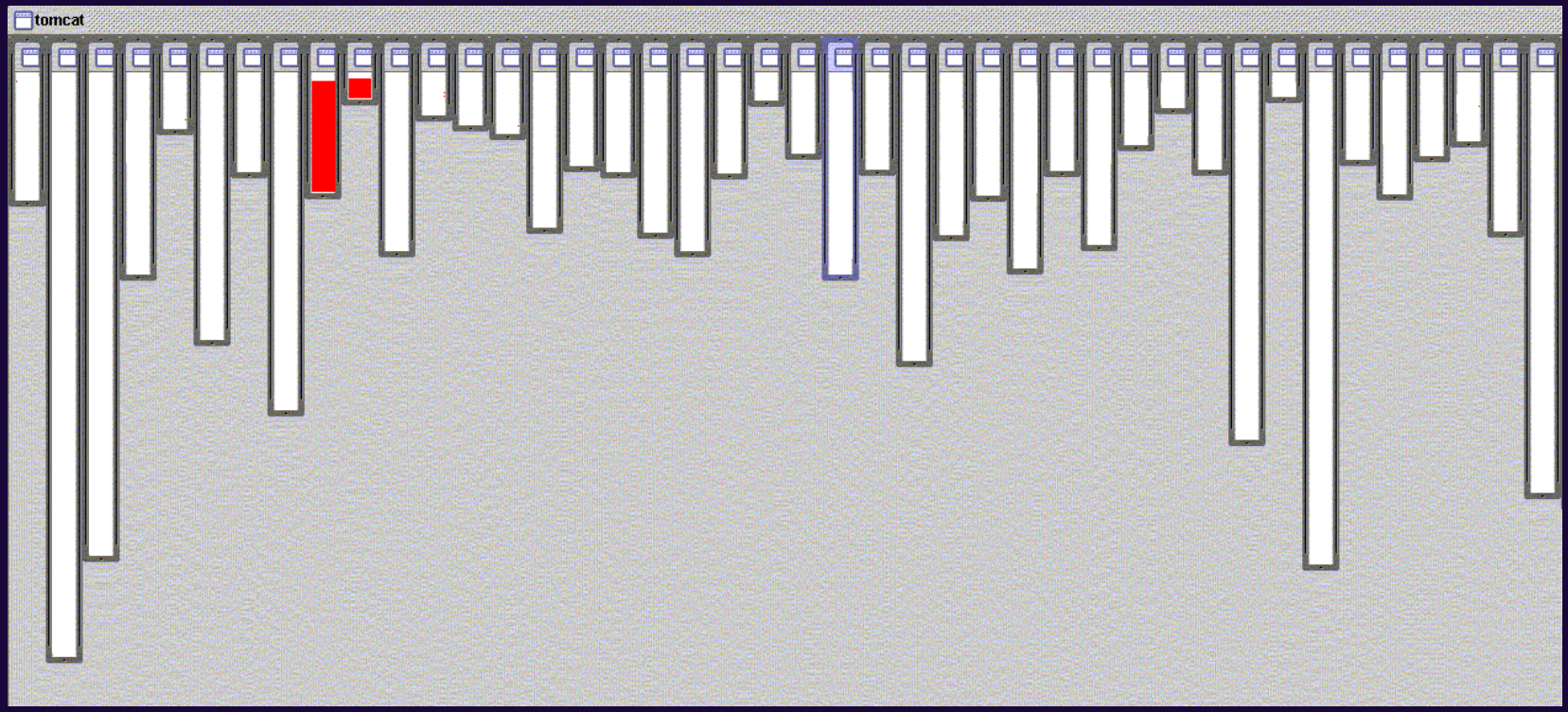


XML parsing in org.apache.tomcat

- red shows relevant lines of code
- nicely fits in one box

Good enough modularity

URL pattern matching

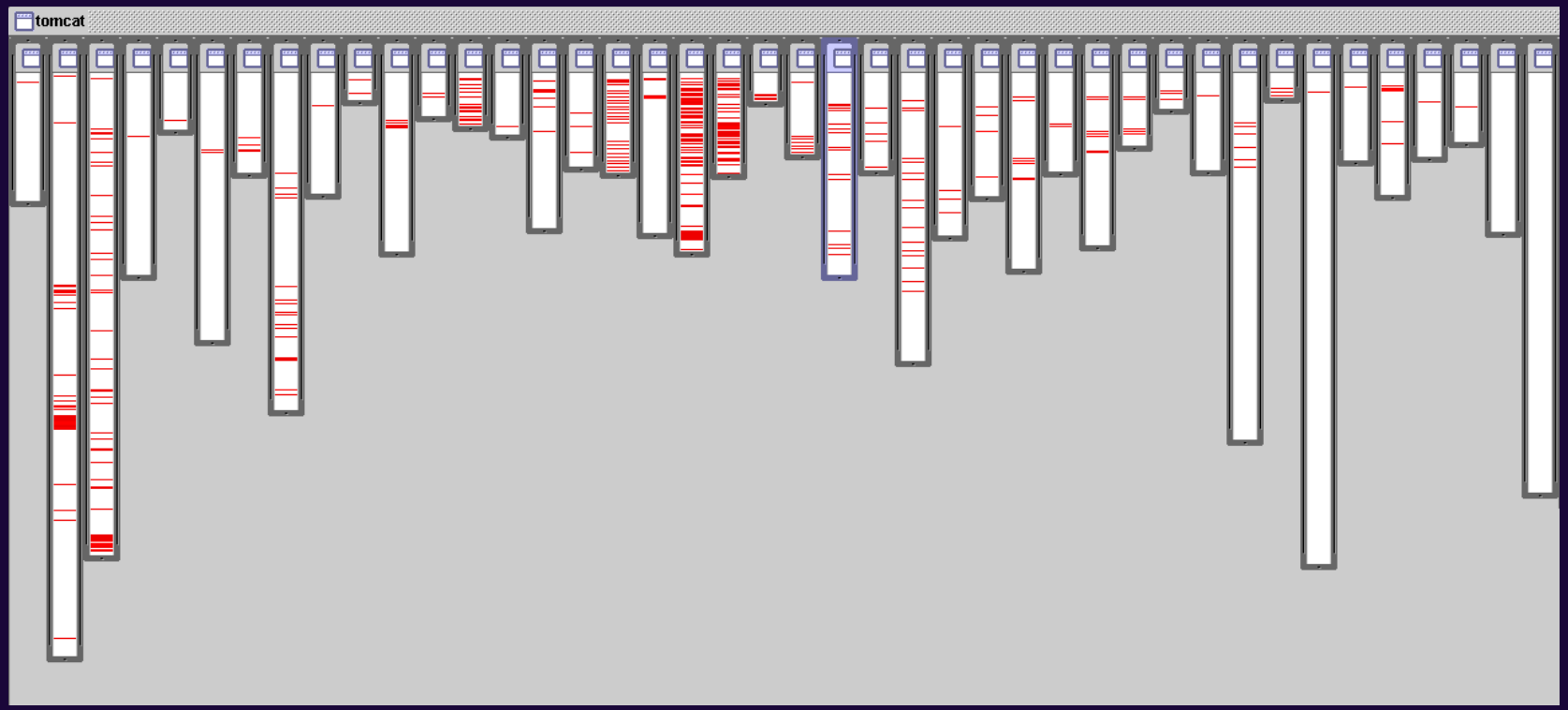


URL pattern matching in org.apache.tomcat

- red shows relevant lines of code
- nicely fits in two boxes (using inheritance)

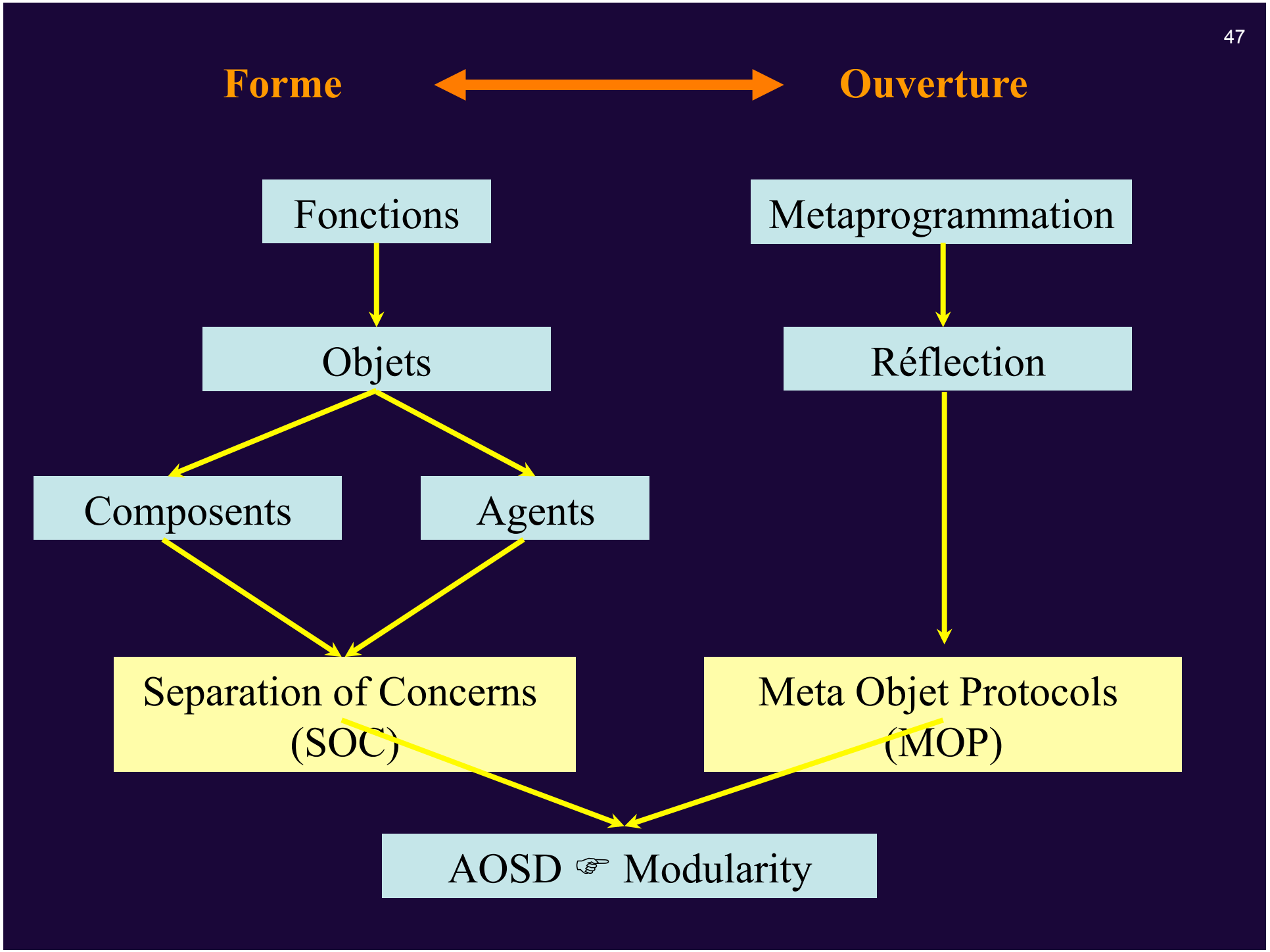
Modularity problems

logging is not modularized



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

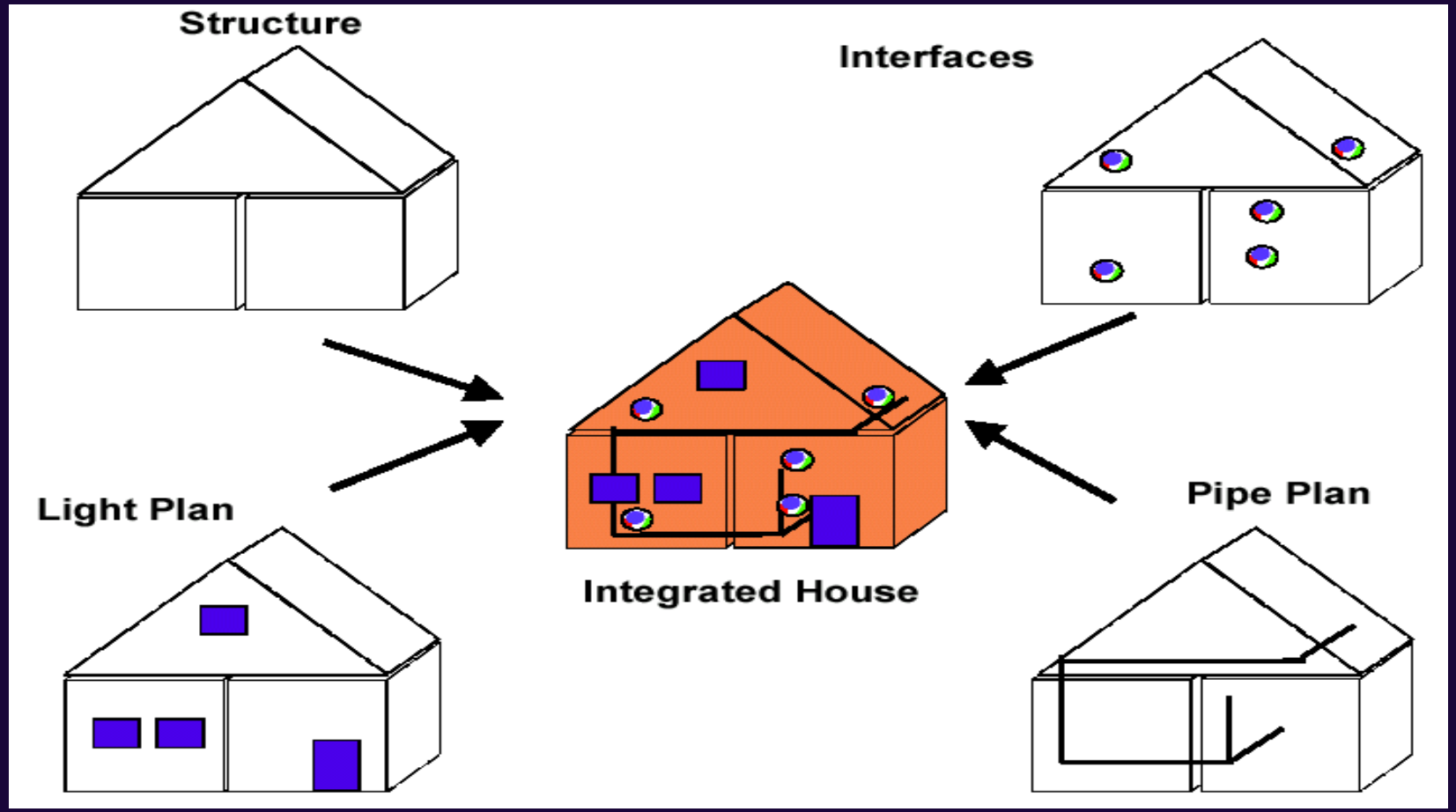

Forme
Ouverture
Fonctions
Metaprogrammation
Objets
Réflection
Composents
Agents
Separation of Concerns
(SOC)
Meta Objet Protocols
(MOP)
AOSD ☞ Modularity

Understanding aspects

Quoting M. Wand :

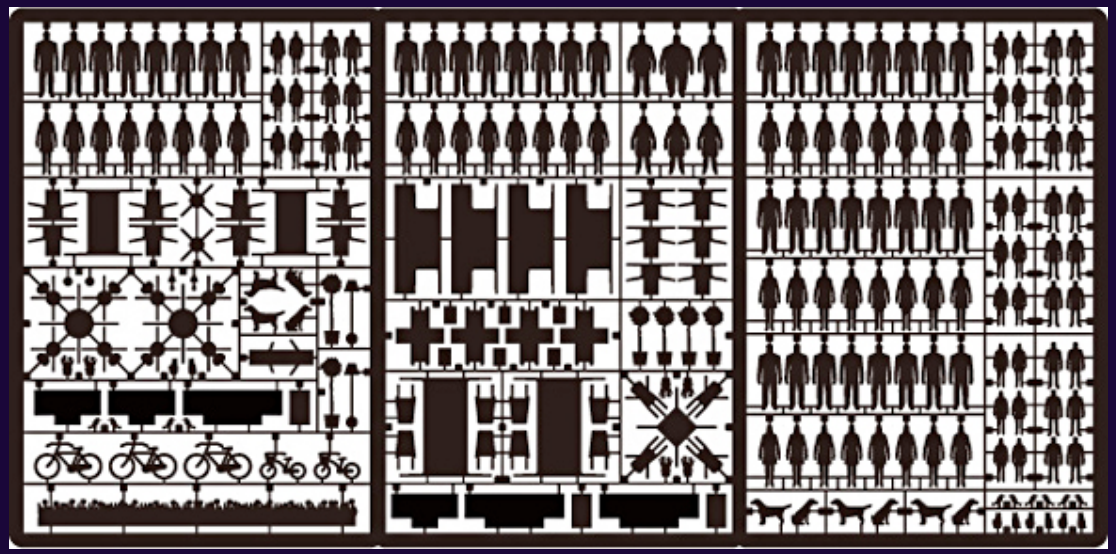
"Aspect-Oriented Programming is a promising recent technology for allowing adaptation of program units across modules boundaries"

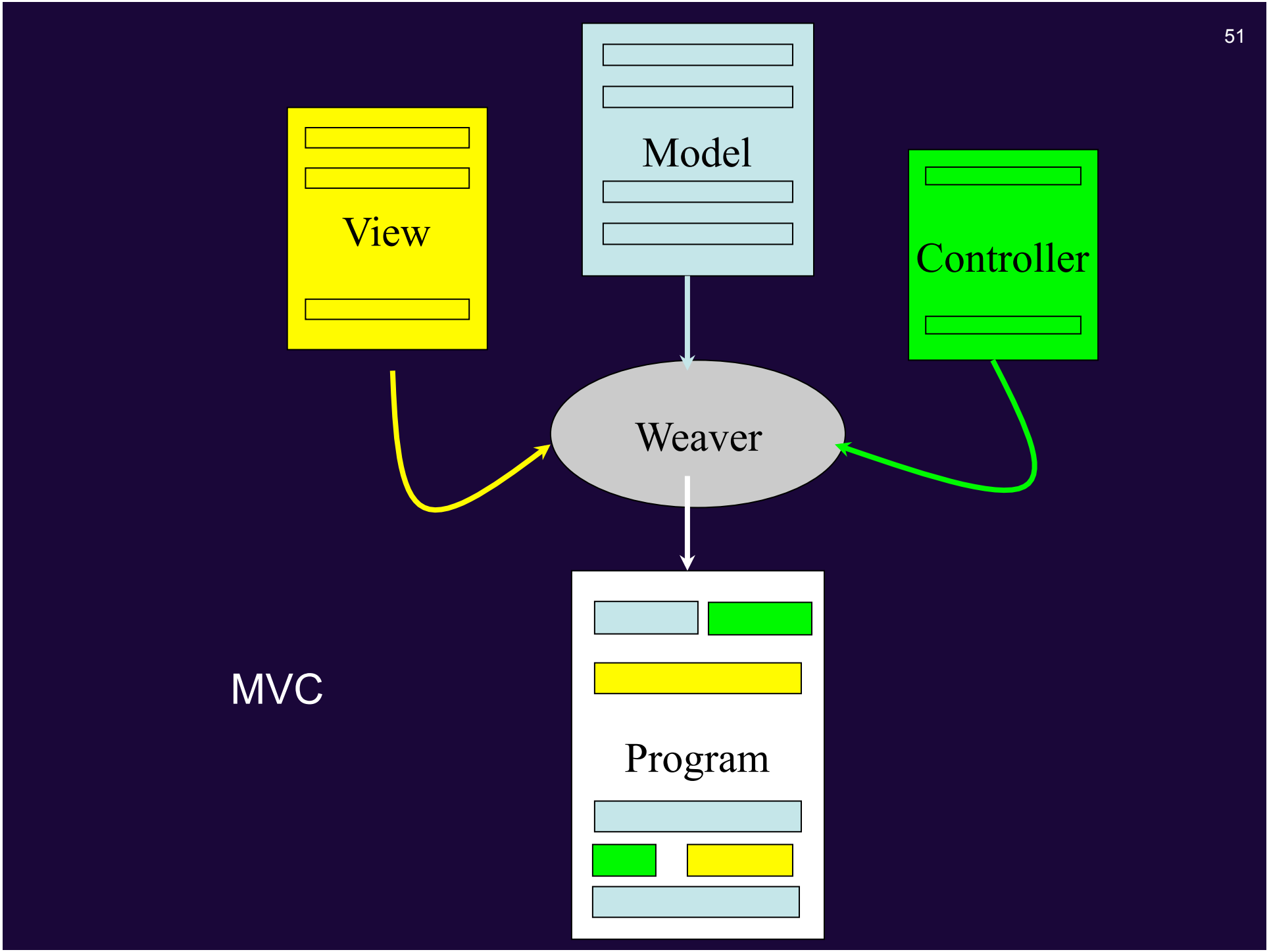
Intuition des aspects (U. Aßmann)

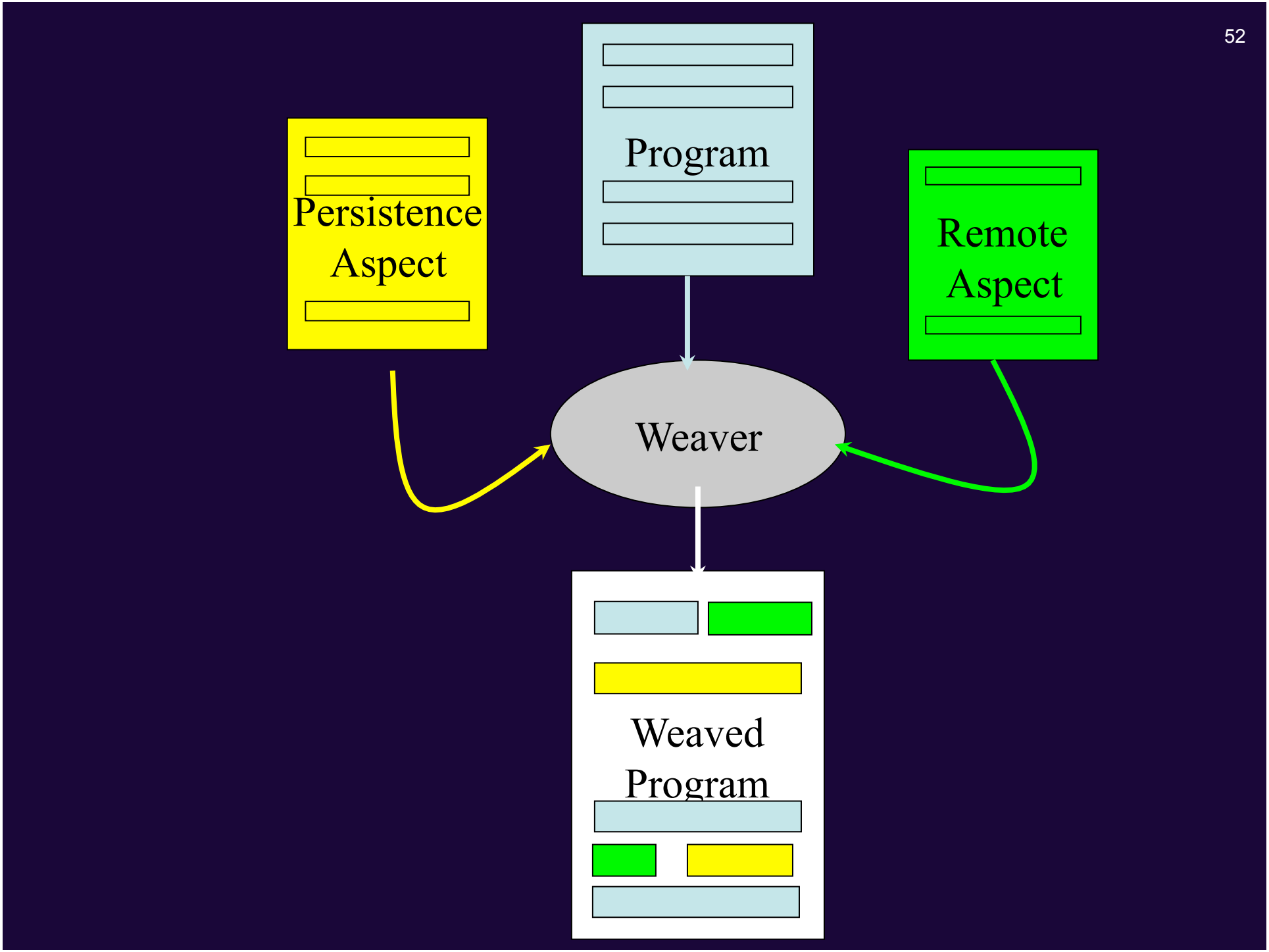


Travail des architectes des années 60

- Calque sur calque = super position
- Trame à coller = Zip = copier/coller



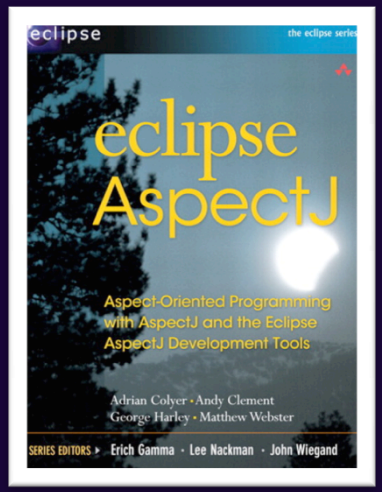

Model
View
Controller
Weaver
Program
MVC


Program
Persistence Aspect
Remote Aspect
Weaver
Weaved Program

Two alternatives

① Expressing every aspect with an “aspectual domain specific language” (ADSL).

② Using the same general purpose language (e.g. Java) for the aspects and the base program ☞ AspectJ



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 (eg 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 AspectJ 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

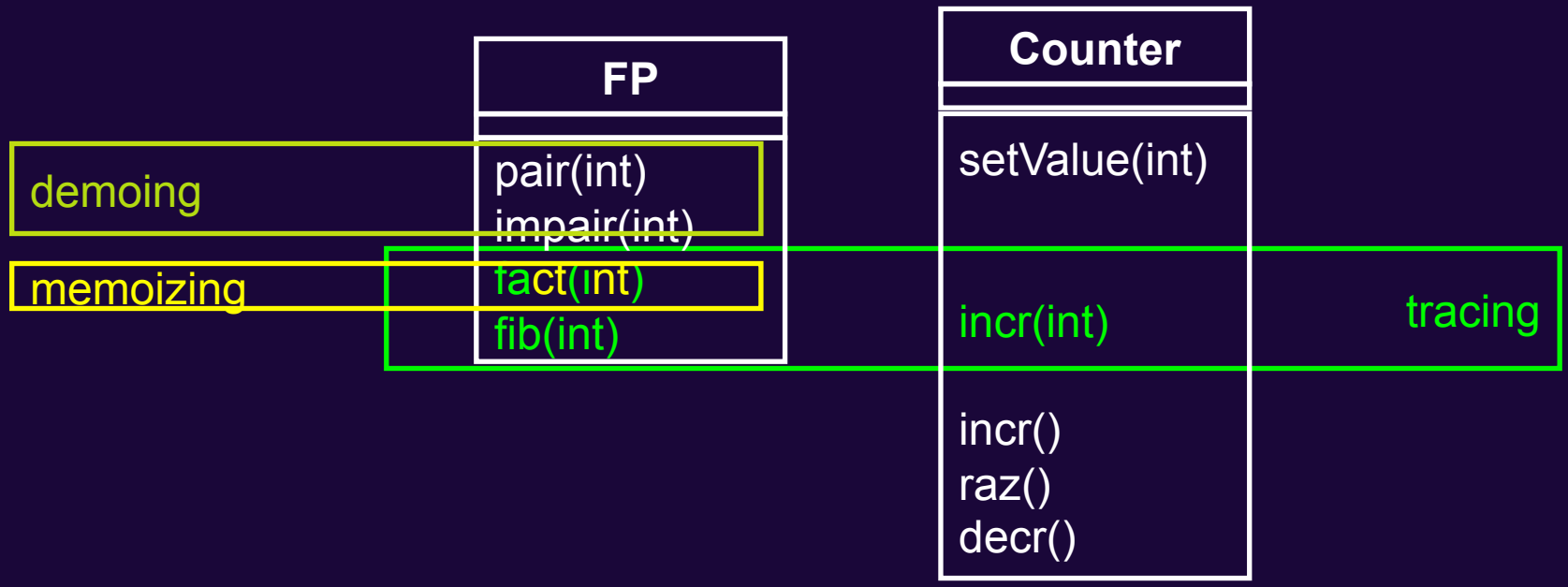
- && operator = logical and = set intersection
- || operator = logical or = set union
- ! operator = logical negation = set difference

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



```
public class FP {
   static boolean pair(int n) { // even
      if (n == 0)
         return true;
      else
         return impair(n - 1);
   }
   static boolean impair(int n) { // odd
      if (n == 0)
         return false
      else
         return pair(n - 1);
   }
```

```
   public static void main(String[] args) {
      S.o.pln("pair(4)-->" + pair(4));
      S.o.pln("pair(3)-->" + pair(3));
   }
```

pair(4)-->true

pair(3)-->false

```
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) {/*dito*/}
   after() returning (boolean b) : callimpair() {/*dito*/}
}
```

```
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) {
      S.o.pln("pair(4)-->" + pair(4));
      S.o.pln("pair(3)-->" + pair(3));
}
```

Before callpair(4)
Before callimpair(3)
Before callpair(2)
Before callimpair(1)
Before callpair(0)
After callpair(true)
After callimpair(true)
After callpair(true)
After callimpair(true)
After callpair(true)
pair(4)-->true

Before callpair(3)
Before callimpair(2)
Before callpair(1)
Before callimpair(0)
After callimpair(false)
After callpair(false)
After callimpair(false)
After callpair(false)
pair(3)-->false

FP++

```
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);
      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--;
   }
}
```

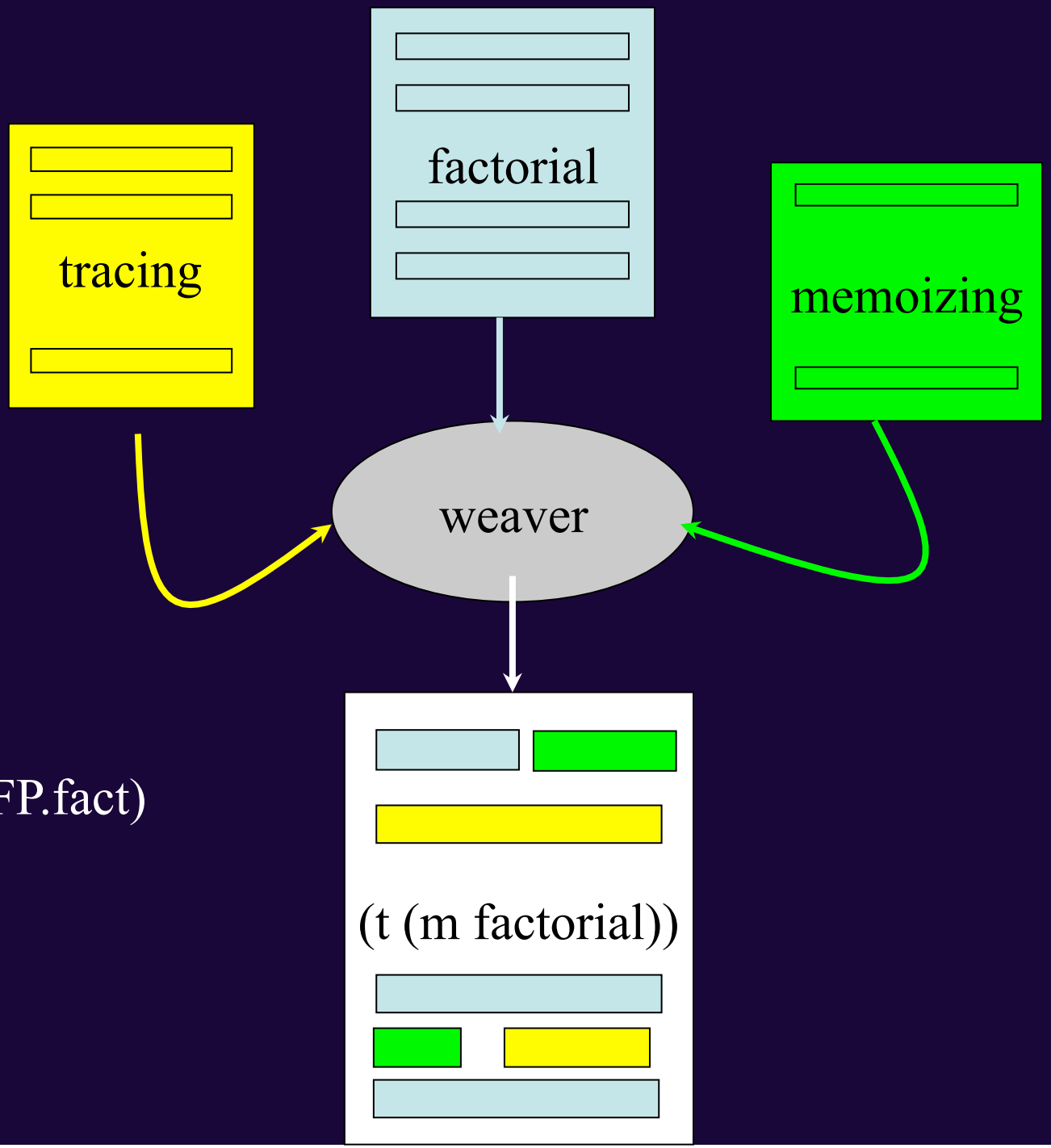
```
public aspect Trace extends TraceProtocol {

   pointcut trace() :
      call(static int FB.fact(int))  ||
      call(static int FP.fib(int))  ||
      call(void Counter.incr(int));

}
```

Execution traces : fact(5) ; fib(4)

```
-->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)
```

```
-->f(4)
-->f(3)
 -->f(2)
  -->f(1)
  1<--f(1)
  -->f(0)
  1<--f(0)
 2<--f(2)
 -->f(1)
 1<--f(1)
3<--f(3)
-->f(2)
 -->f(1)
 1<--f(1)
 -->f(0)
 1<--f(0)
2<--f(2)
5<--f(4)
```

```
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;
      }
   }
}
```




```
(trace
   (memoize FP.fact)
 )
```

FP++

```
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);
   }
```

Execution traces : fact(5);fact(4);fact(6)

-->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)

-->f(4)

24<--f(4)

-->f(6)

-->f(5)

120<--f(5)

720<--f(6)

Intuition of a Java Clock

```
class Clock {
  static void main(String[] args) {
    while (true) {
      System.out.println(currentTime());
      sleep(ONE_MINUTE);
    }
  }
```

```
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

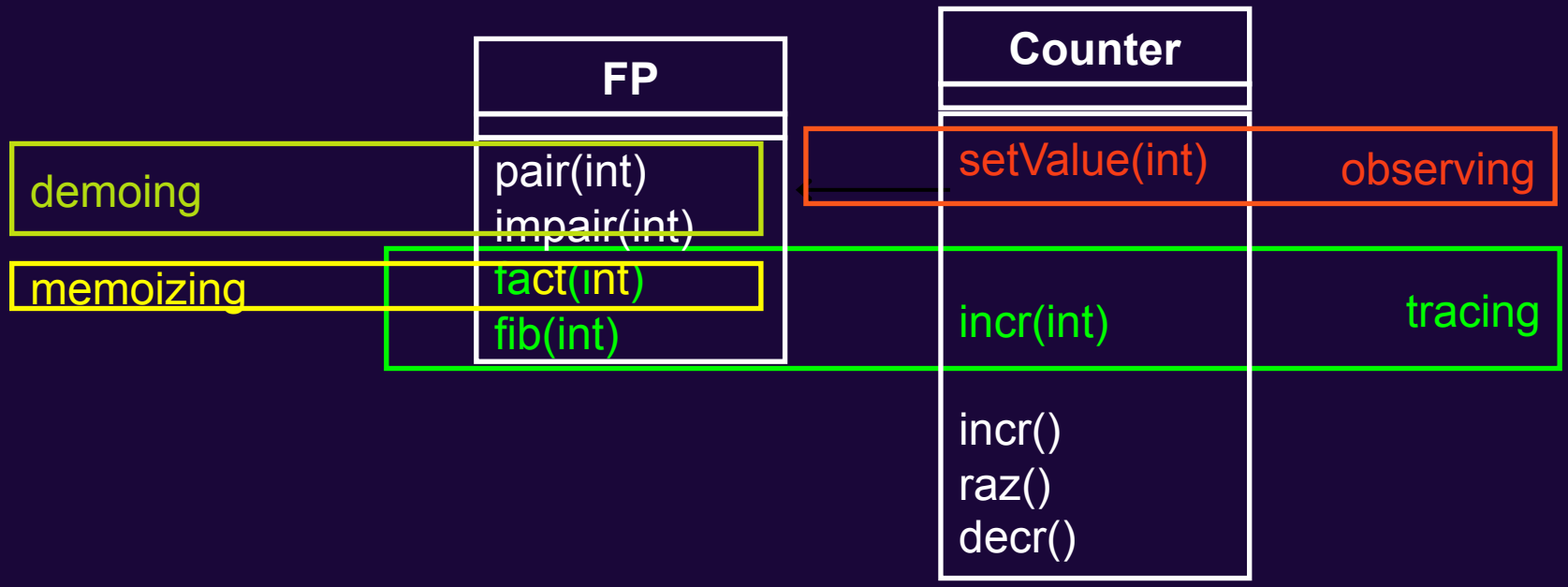
Intercessing with the program behavior

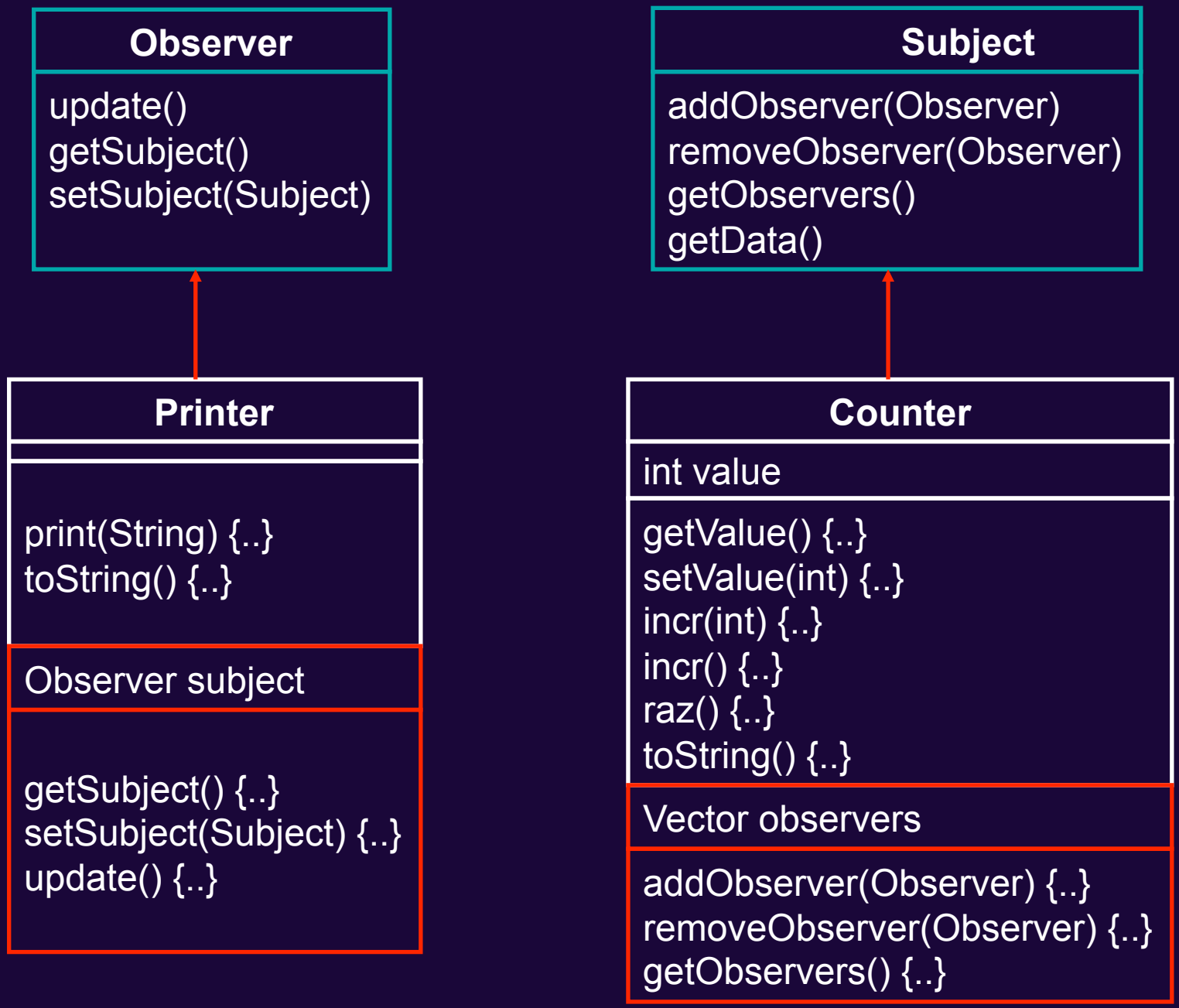
- user-defined and primitive pointcuts
- cflow

Changing the program structure

- introducing field
- introducing method
- introducing constructor
- modifying class hierarchy

Aspectisation of the Observer DP




Observer
update()
getSubject()
setSubject(Subject)
Subject
addObserver(Observer)
removeObserver(Observer)
getObservers()
getData()
Printer
print(String) {..}
toString() {..}
Observer subject
getSubject() {..}
setSubject(Subject) {..}
update() {..}
Counter
int value
getValue() {..}
setValue(int) {..}
incr(int) {..}
incr() {..}
raz() {..}
toString() {..}
Vector observers
addObserver(Observer) {..}
removeObserver(Observer) {..}
getObservers() {..}

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

```
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

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

}
```

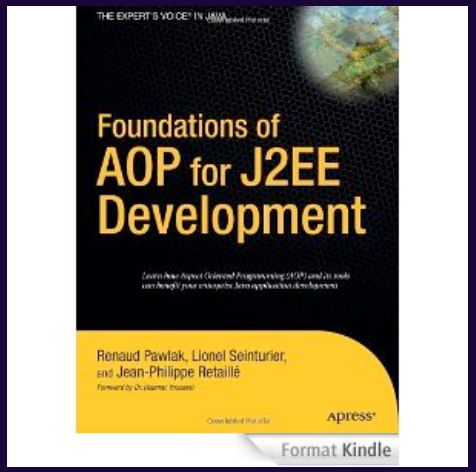
```
aspect CounterObserver extends PatternObserverProtocol {

   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));
}
```

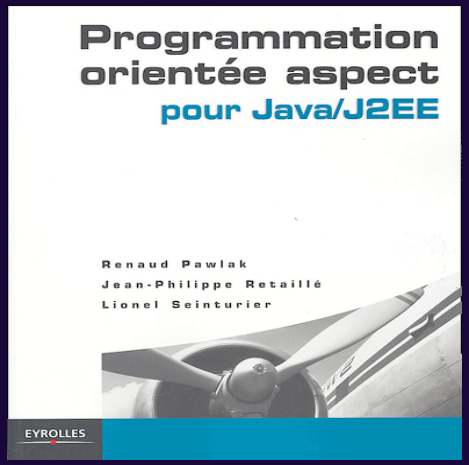
main in Counter

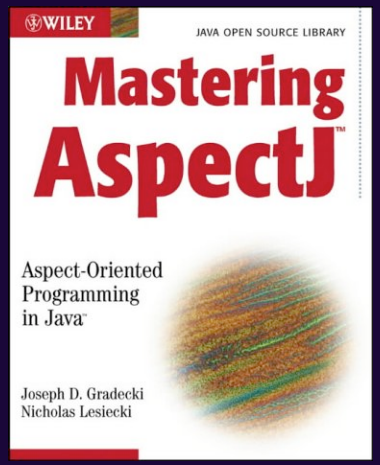
```
public static void main(String[] args) {
   Counter c1 = new Counter();
   Printer scribe = new Printer();
   c1.raz();
   c1.addObserver(scribe);
   c1.incr();
   c1.incr(6);
   c1.raz();
}
```

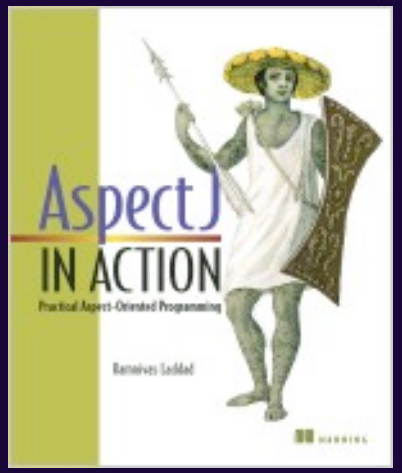
```
-->f(1)
[Printer]update occured in
Counter@1
   -->f(6)
[Printer]update occured in
Counter@7
[Printer]update occured in
Counter@0
```

AspectJ dans la littérature









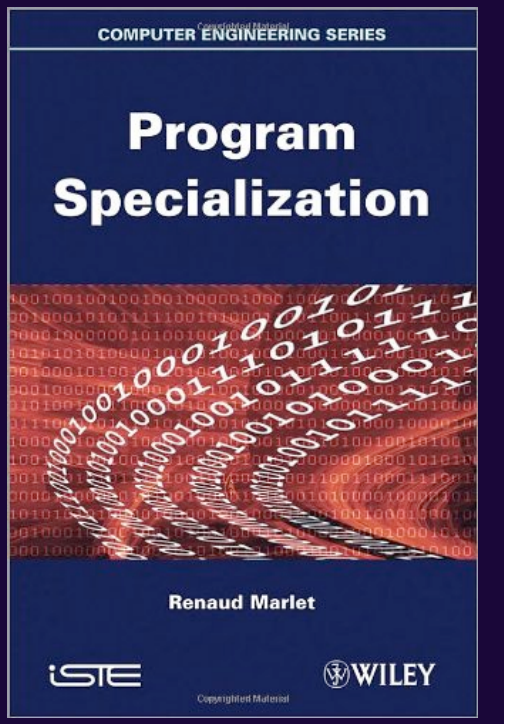
Two alternatives

① Expressing every aspect with a domain specific language ☞ (A)DSL

② Using the same general purpose language (e.g. Java) for the aspects and the base program

Langages dédiés

- Renaud Marlet – HDR 207
  - Spécialiser les programmes
    - Les grands principes
    - Un spécialiseur pour C : Tempo
    - Précision des analyses
    - Spécialisation des données
  - Spécialiser les langages
    - Notion de langage dédié
    - Une méthodologie de construction
    - Langages dédiés pour la génération de pilotes
    - De la théorie à la programmation
  - http://imagine.enpc.fr/~marletr/



- http://fr.wikipedia.org/wiki/Langage_d%C3%A9di%C3%A9

Langages multi-paradigmes

- SCALA (SCAlable Language) Martin Odersky
  - Pattern Matching
  - Traits
  - Concurrency
  - Type inference
  - Higher-order functions
  - http://www.scala-lang.org/

- OCAML
  - Projet Cristal@Inria
  - http://caml.inria.fr/ocaml/release.fr.html

Programmer le Web

- HOP & HIP-HOP
  - Héritage BigLoo & Esterel
  - Emmanuel Serrano et Gérard Berry
  - http://hop.inria.fr/
- Rackett
  - Héritage Scheme
  - http://racket-lang.org/
- Lively (Wiki)
  - Héritage Smalltalk et AspectJ
  - Dan Ingalls et Robert Hirschfield
  - http://lively-kernel.org/
- Pharo
  - Version libre de Squeak sous licence MIT
  - Stéphane Ducasse
  - http://pharo.org/

Présentations récentes

- W. Cook
  - A View of the Past and Future of Objects. ECOOP 2014
  - Enso
  - http://www.cs.utexas.edu/~wcook/Drafts/2012/2012-07-13EnsoMSR.pdf
- J.-P. Briot
  - Composants et agents : évolution de la programmation et analyse comparative. TSI 2014
  - http://www-desir.lip6.fr/~briot/cv/composants-agents-briot-tsi.pdf
- R. Ducourneau
  - Les talons d'Achille de la programmation par objets. GDR GPL 2014
  - http://gdr-gpl.cnrs.fr/sites/default/files/documentsGPL/JourneesNationales/GPL2014/achille.pdf

Discussion sur les objets et les aspects

- G. Kiczales
  - http://www.cs.ubc.ca/~gregor/
- M. Wand
  - http://www.ccs.neu.edu/home/wand/
- R. Gabriel
  - http://www.dreamsongs.com/Essays.html
- J.-F. Perrot's group
  - http://www.lip6.fr/colloque-JFP/
- AOSD conference and European Network On Excellence
  - http://www.aosd.net/
  - http://www.aosd-europe.net/