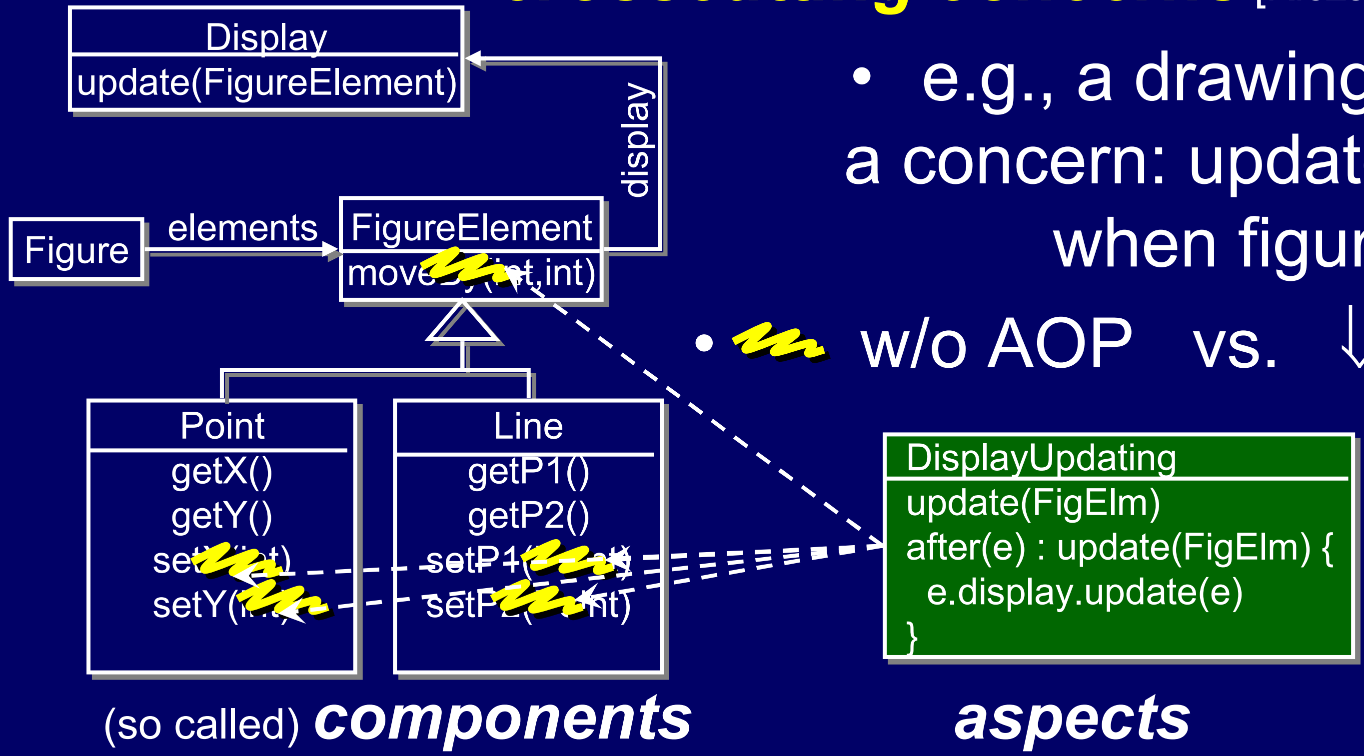
modeling crosscutting in aspect-oriented mechanisms

Hidehiko Masuhara
(University of Tokyo)

joint work with Gregor Kiczales
(University of British Columbia)

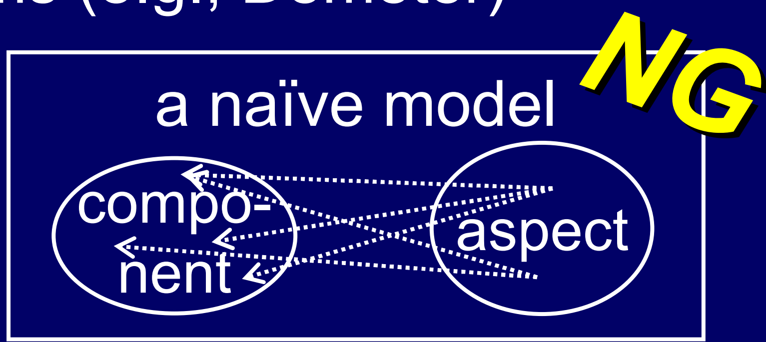
aspect-oriented programming

- AOP supports modularization of ***crosscutting concerns*** [Kiczales et al.1997]
  - e.g., a drawing editor & a concern: update display when figure moves
- w/o AOP vs. ⇓ w/ AOP



what's the essence of AOP?

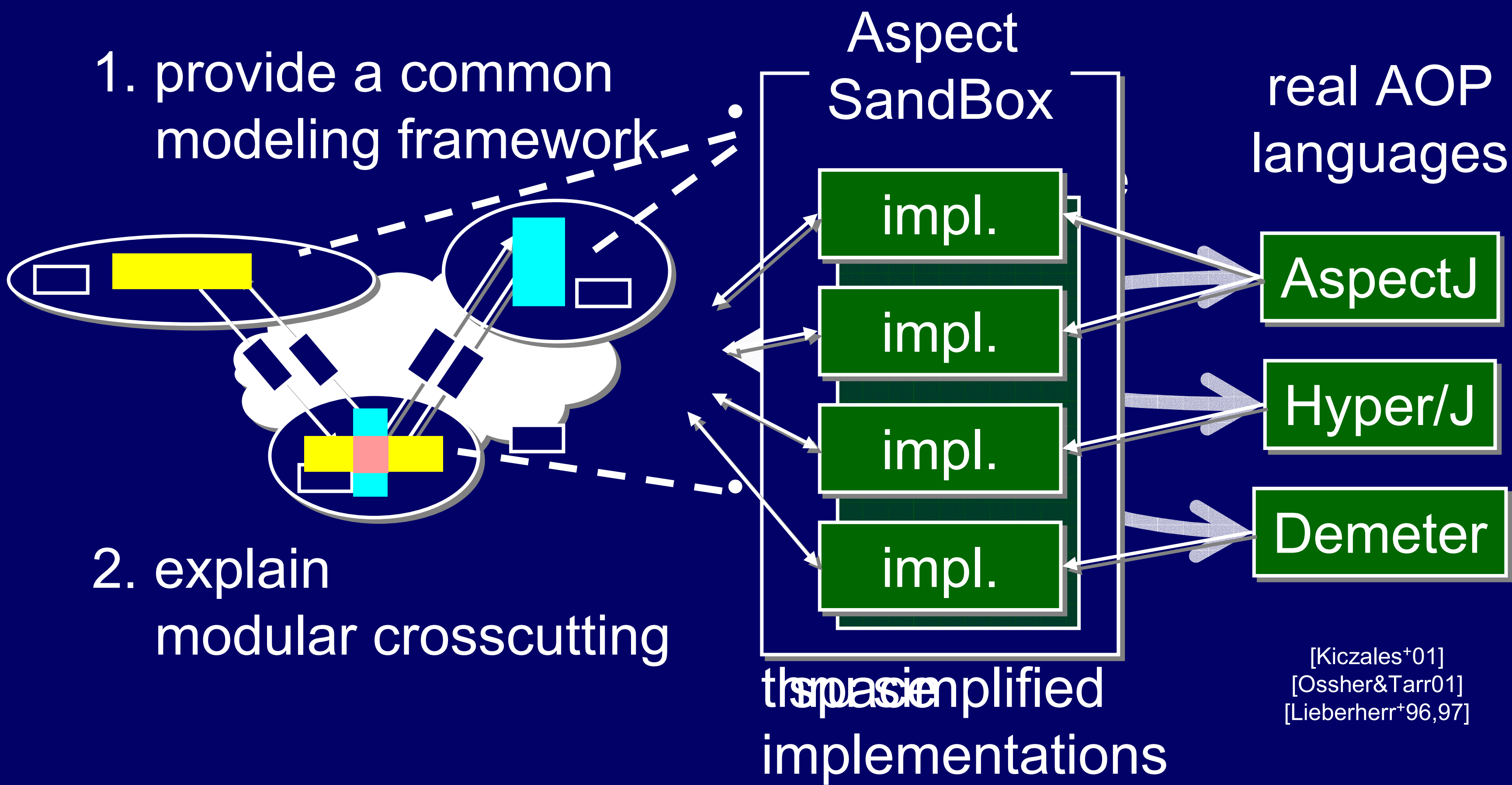
- a naïve model does not capture
  - symmetric mechanism in Hyper/J
  - dynamic mechanism in AspectJ
  - more specialized mechanisms (e.g., Demeter)
  - …
- we'd like to find a model
  - general enough to capture many mechanisms
  - not too general so that we can see the nature of AOP



crosscutting!

contributions & approach

1. provide a common modeling framework
2. explain modular crosscutting



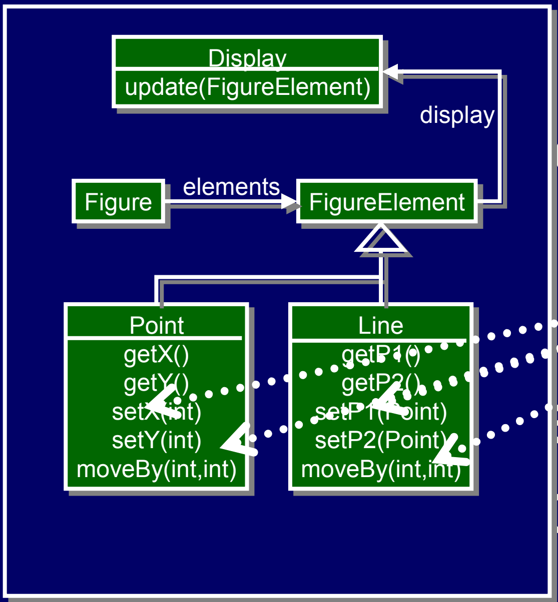
talk outline

- implementations of core AOP mechanisms
  - PA: an AspectJ-like (dynamic) mechanism
  - COMPOSITOR: a Hyper/J-like mechanism
  - (TRAV: a Demeter-like mechanism)
  - (OC: an AspectJ-like (static) mechanism)
- the modeling framework
- modular crosscutting
  in terms of the modeling framework

PA – pointcuts and advice

- simplified from (dynamic part of) AspectJ [Kiczales01]
- key elements:
  - join point: ***point in execution***
    i.e., a method call

an example advice:
update display
after moving
any element



fies *when*

es

t to do

erior

dularity

```
after( ):
      call(void Point.setX(int))
  || call(void Point.setY(int))
  || call(void Line.setP1(Point))
  || call(void Line.setP2(Point)) {
  display.update();
}
```

PA: implementation

```
(define eval-exp
  (lambda (exp env)
    (cond ((call-exp? exp)
           (call-method (call-exp-mname exp)
               (eval-exp (call-exp-obj exp) env)
               (eval-rands (call-exp-rands exp) env)))
          ...)))

(define-struct call-jp (mname target args))

(define call-method
  (lambda (mname obj args)
    (let*((jp (make-call-jp mname obj args))
          (method (lookup-method jp))
          (advice   (lookup-advice jp)))
     (execute-advice advice jp
       (lambda ( )
         (execute-method method jp))))))
```

an interpreter (à la EOPL)

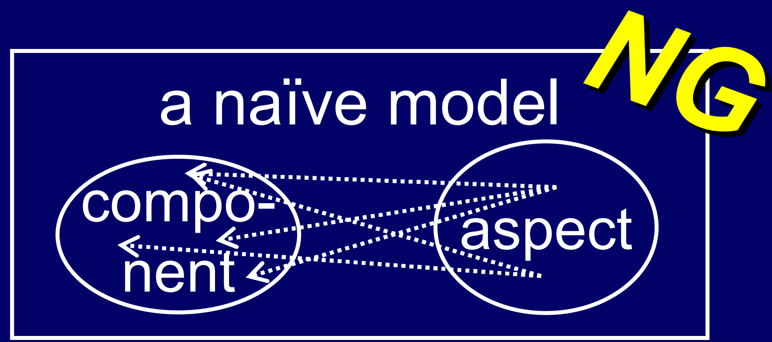
a join point represents a method call

a method call is to:

- create a join point
- identify a method
- identify advice decls.
- execute advice decls.
- execute method

observations from PA implementation

- method and advice are treated similarly:
  lookup & execute
  ➔ symmetric model

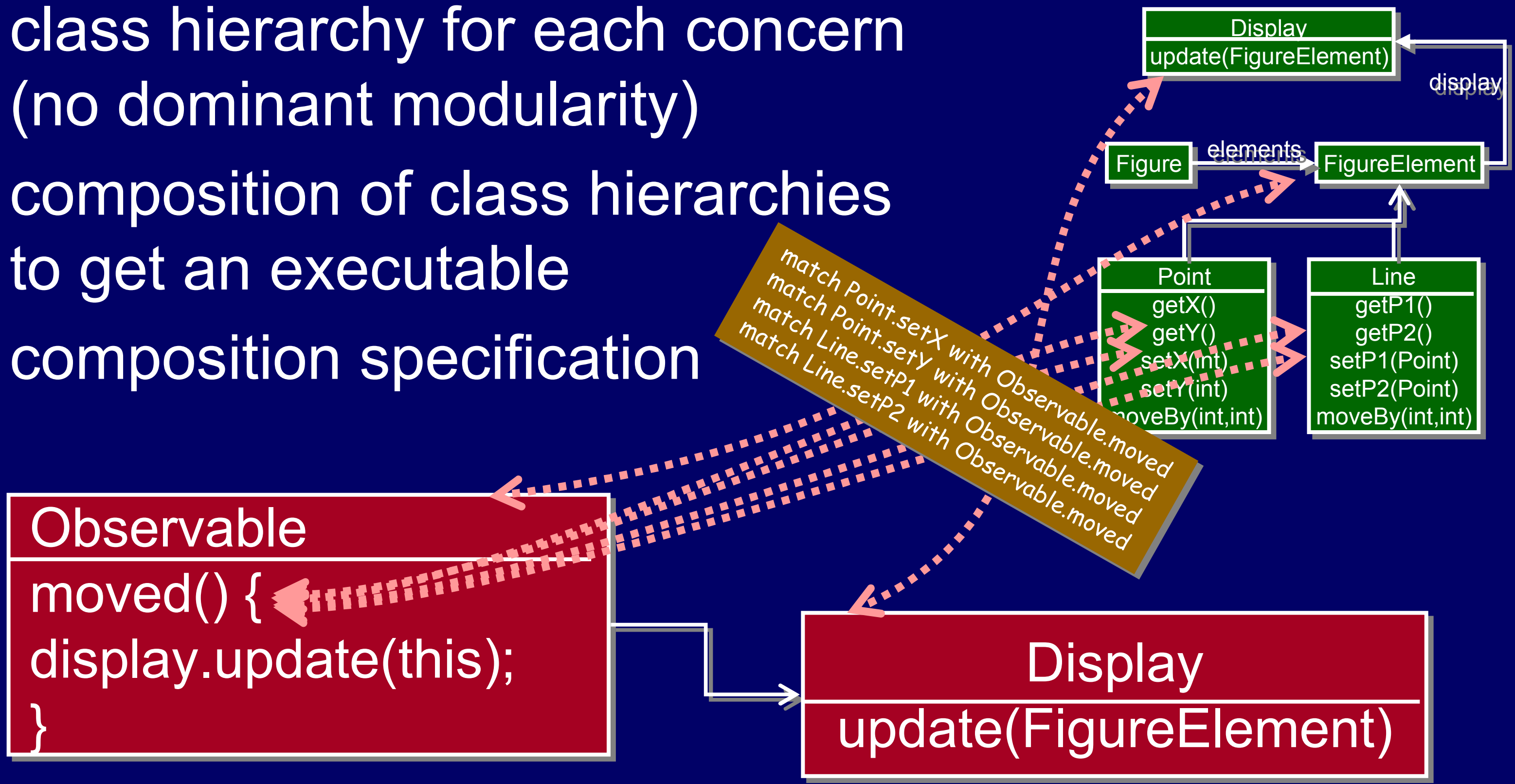


- join points come from execution
  ➔ "weaving into components" is not good
  ➔ weaving in the third space
  (i.e., execution)

COMPOSITOR – class composition

simplified from Hyper/J [Ossher01]

- class hierarchy for each concern (no dominant modularity)
- composition of class hierarchies to get an executable
- composition specification

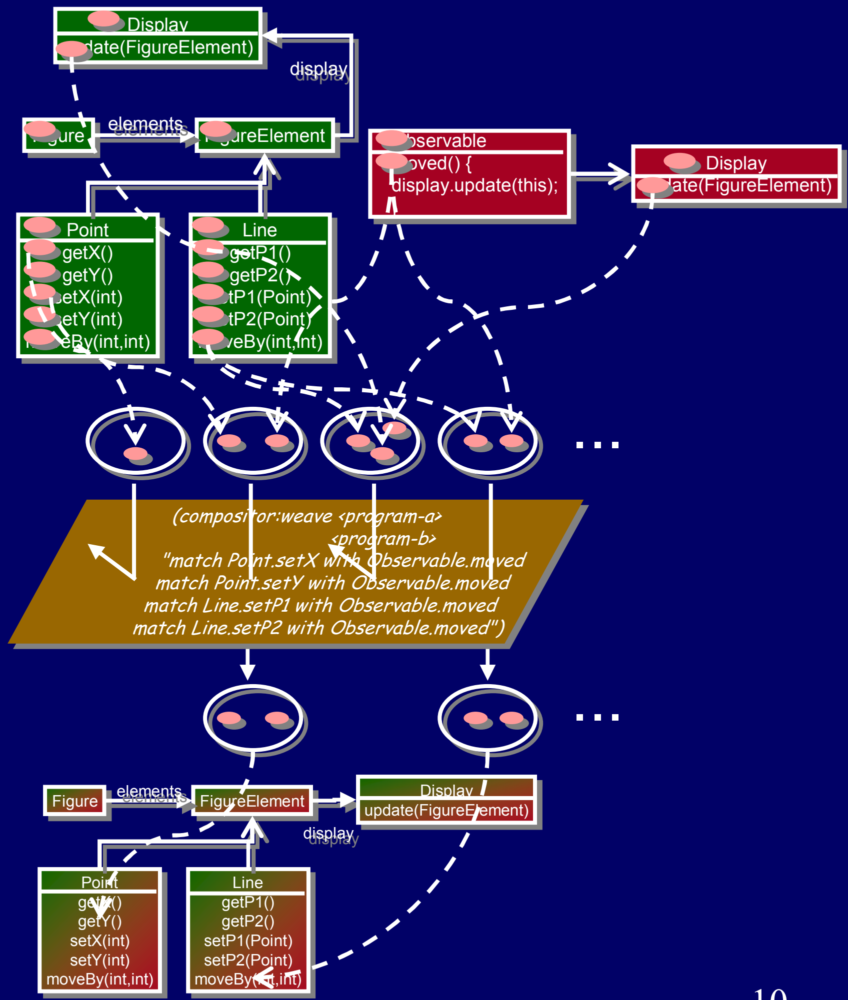


COMPOSITOR: implementation

source-to-source translation

1. computes all possible combinations *
2. determines whether each should be merged
3. merges bodies & adds to program

(* very naïve approach; just for explanation)



COMPOSITOR: implementation

```
(define compositor:weave
 (lambda (pgm-a pgm-b relationships)
  (let loop ((pgm   (make-program '()))
             (seeds (compute-seeds pgm-a pgm-b)))
   (if (not (null? seeds))
     (let ((signature (all-match (car seeds)
                                 relationships)))
       (if signature
         (let* ((jp     (car seeds))
                (decl (merge-decls jp relationships)))
          (loop (add-decl-to-pgm decl pgm signature)
                (remove-subsets jp (cdr seeds))))
         (loop pgm (cdr seeds))))
     pgm))))
```

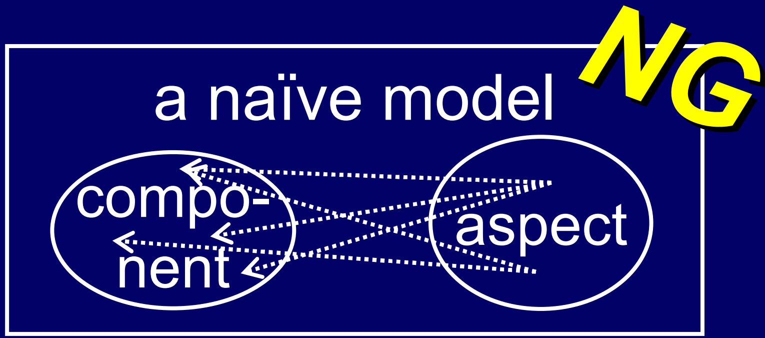
generate combinations of methods

test all methods have matching signatures

merge method bodies and install

observations from COMPOSITOR implementation

- no dominant modularity
  ➔ symmetric model



- join points are not only from pgm-a (nor pgm-b)
  ➔ “weaving into components” is not good
  ➔ weaving in the third space
- matching rule can be modified
  ➔ weaving parameters

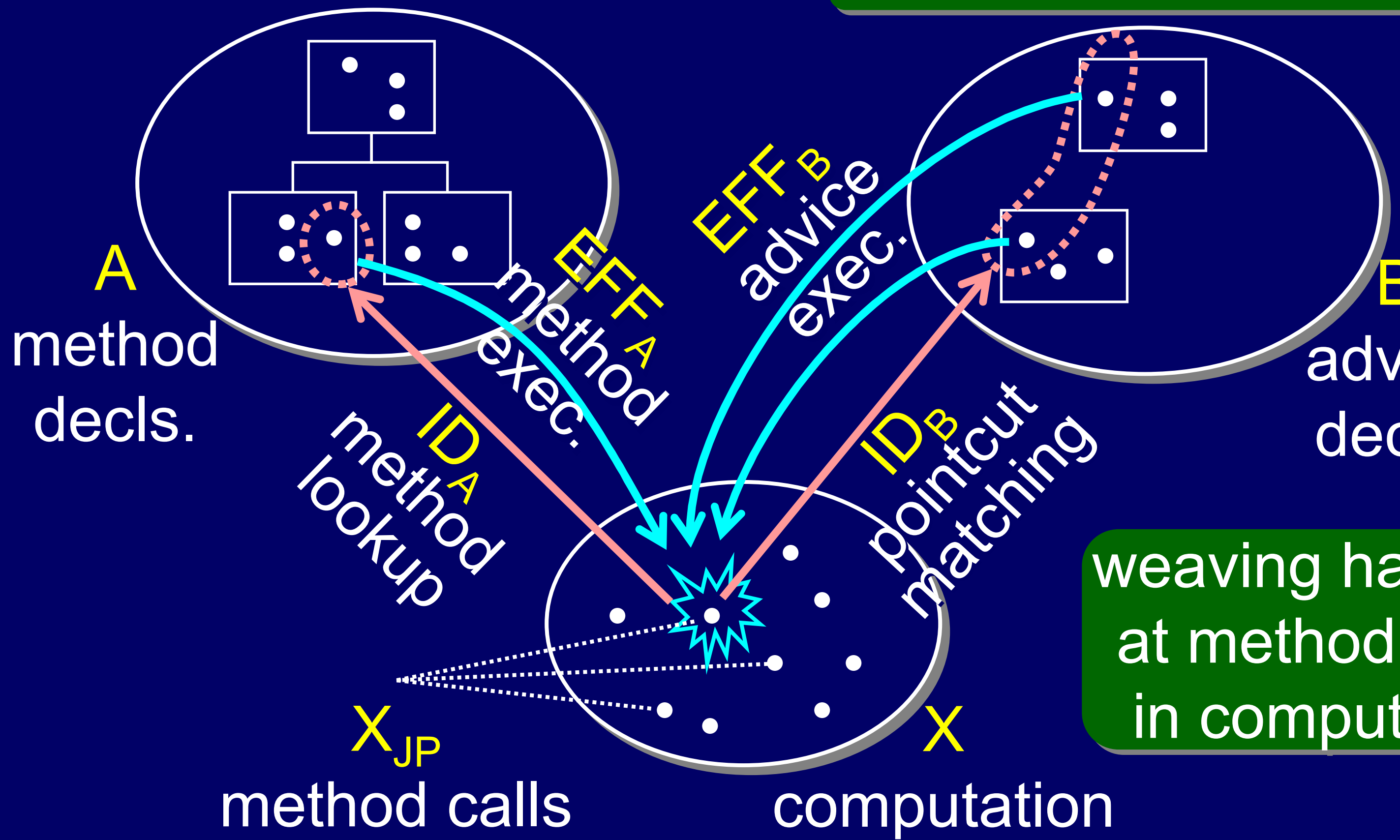
TRAV & OC

similarly implemented

- TRAV: Demeter/DemeterJ/DJ [Liberrherr97], etc.
  - traversals through object graphs
    - modular specification: “where to go” & “what to do” otherwise scattered over classes
- OC: AspectJ’s introductions or ITD [Kiczales01] (also in Flavors, etc. [Cannon82]...)
  - can declare methods/fields outside of the class declarations

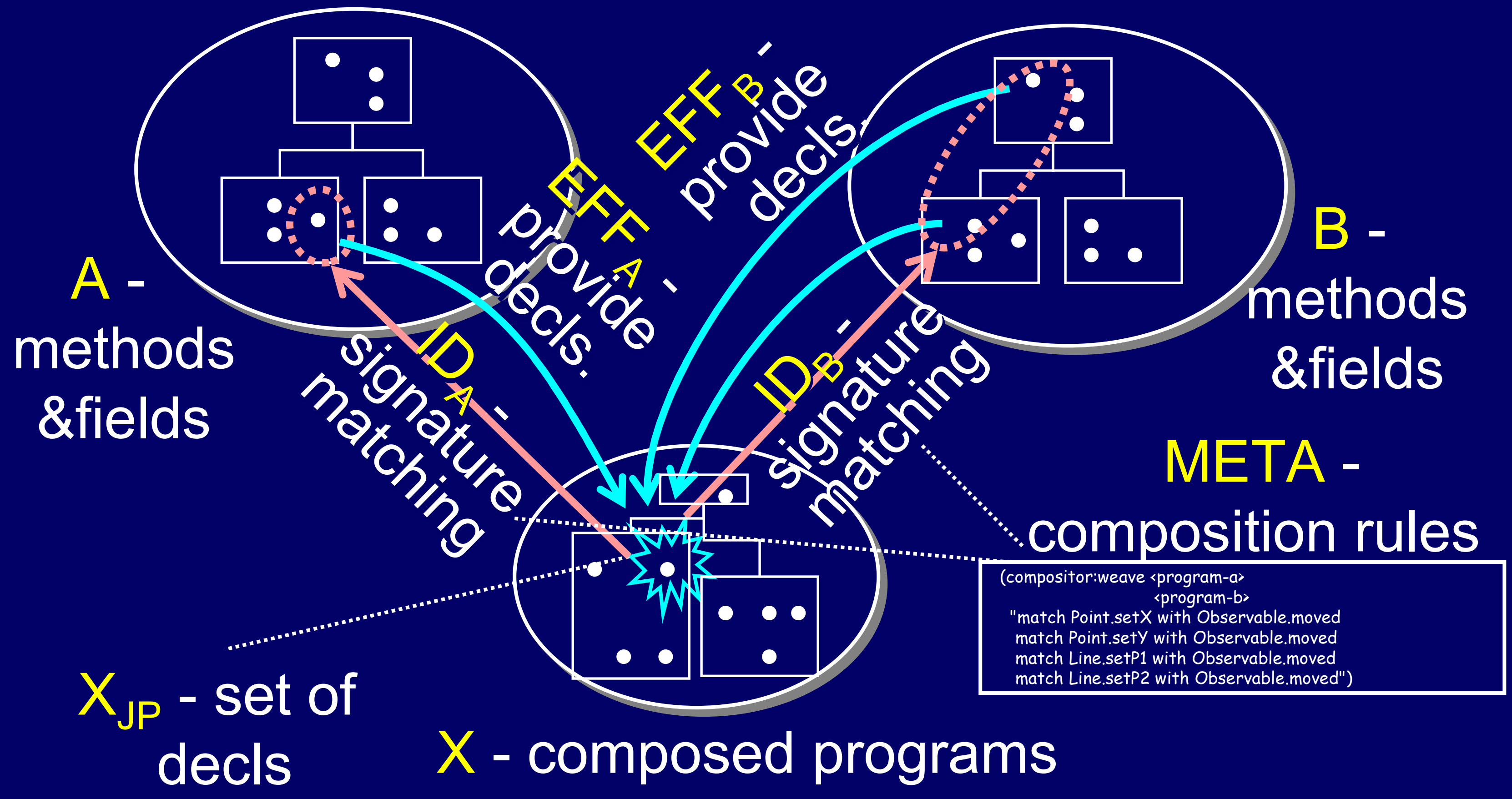
the modeling framework: PA's case

method & advice are parallel

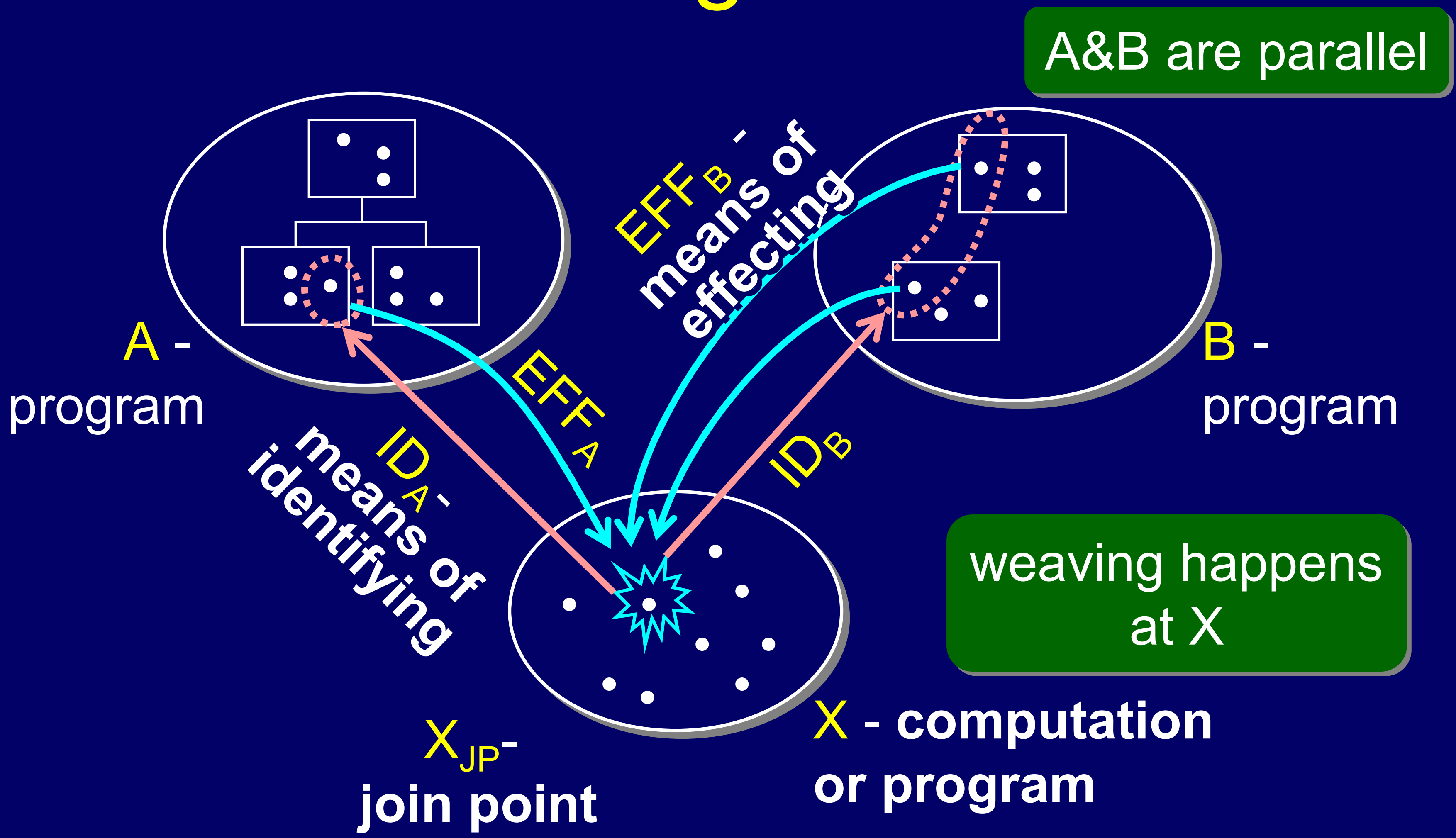


weaving happens at method calls in computation

modeling framework: COMPOSITOR's case



the modeling framework



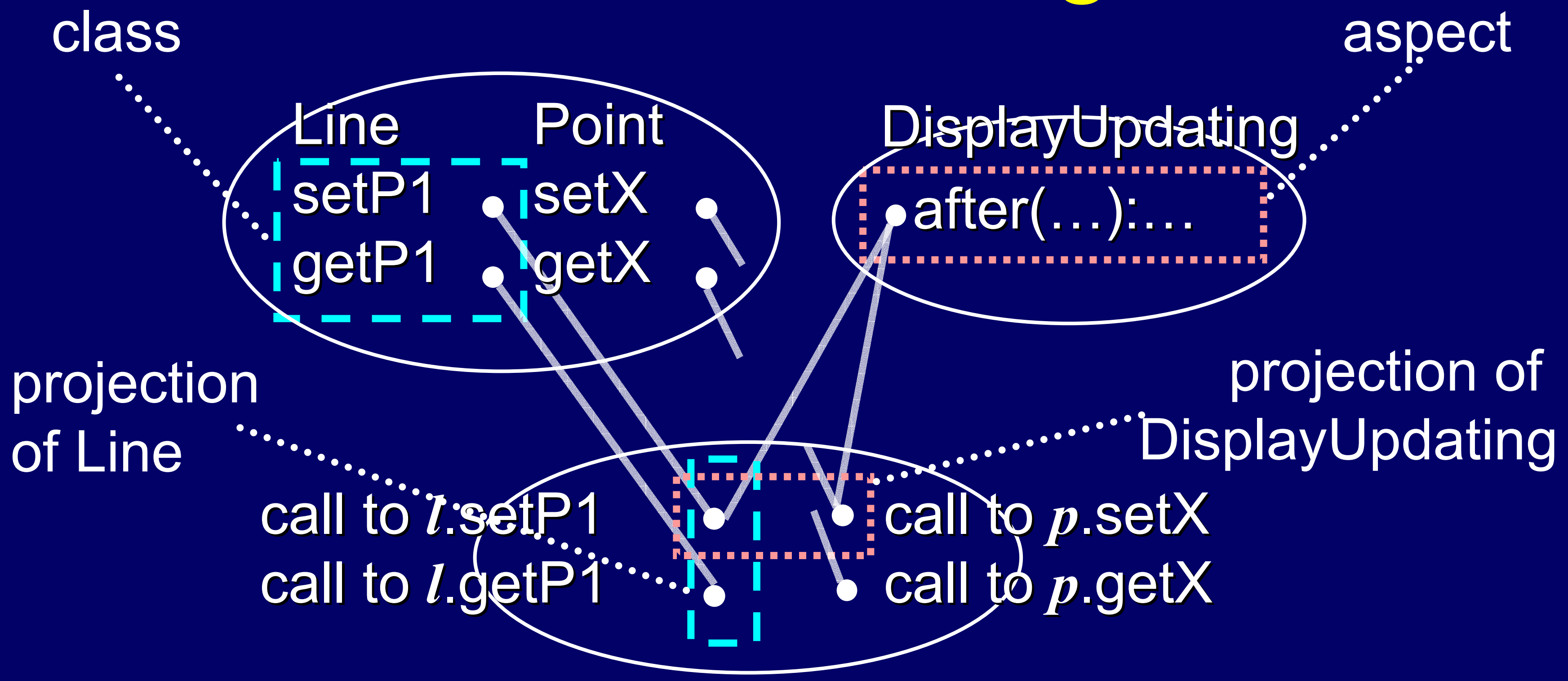
models for 4 mechanisms

	PA	TRAV	COMPOSITOR	OC
X	program execution	traversal execution	composed program	combined program
X_{JP}	method calls	arrival at each object	declarations in X	class declarations
A	c, m, f declarations	c, f declarations	c, m, f declarations	c declarations w/o OC declarations
A_{ID}	m signatures, etc.	c, f signatures	c, m, f signatures	method signatures
A_{EFF}	execute method body	provide reachability	provide declarations	provide declarations
B	advice declarations	traversal spec. & visitor	(= A)	OC method declarations
B_{ID}	pointcuts	traversal spec.	(= A_{ID})	effective method signatures
B_{EFF}	execute advice body	call visitor & continue	(= A_{EFF})	copy method declarations
META	none	none	match & merge rules	none

what's modular crosscutting?

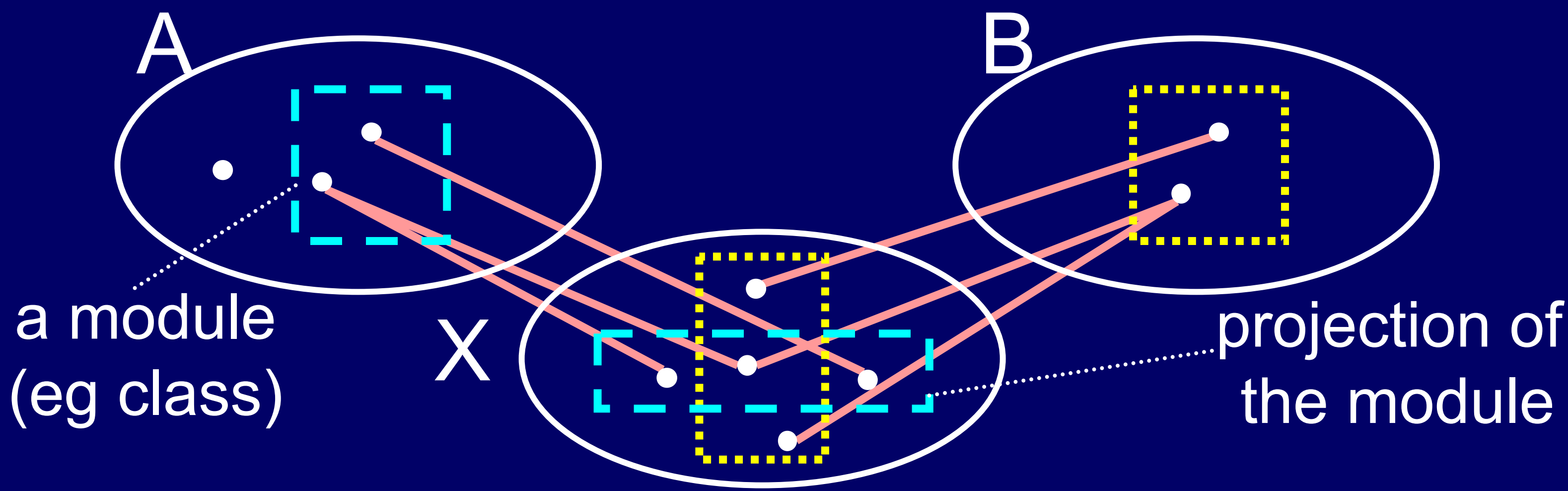
- it is said:
  "*AOP supports modular crosscutting*"
  but what is it?
- the modeling framework can explain:
  two modules in A&B crosscut when
  projections of the modules
  into X intersect and
  neither is subset of the other

an example of modular crosscutting in PA



"Line and DisplayUpdating crosscut in the execution"

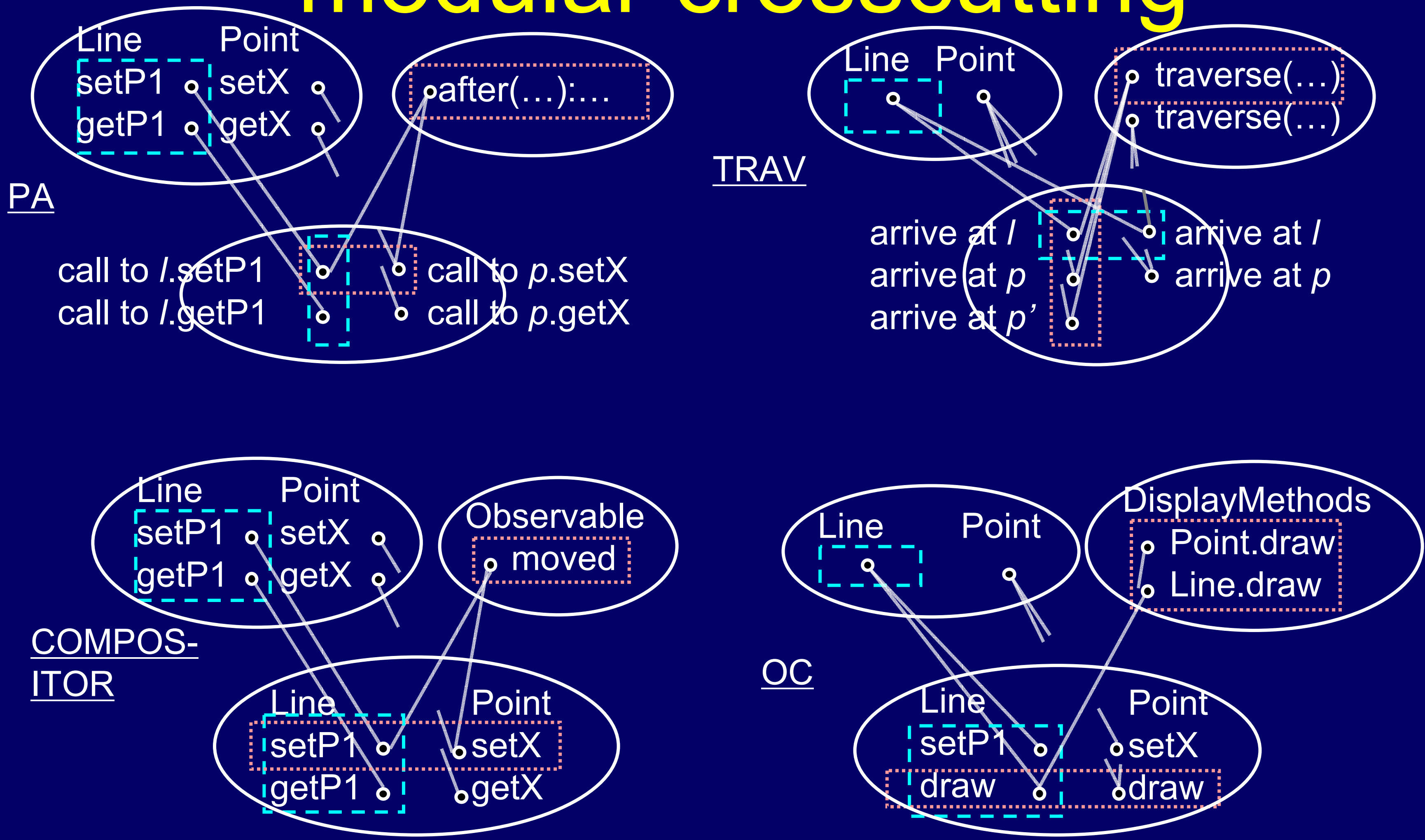
what's modular crosscutting?



two modules in A&B crosscut when

projections of the modules
into X intersect and
neither is subset of the other

examples of modular crosscutting



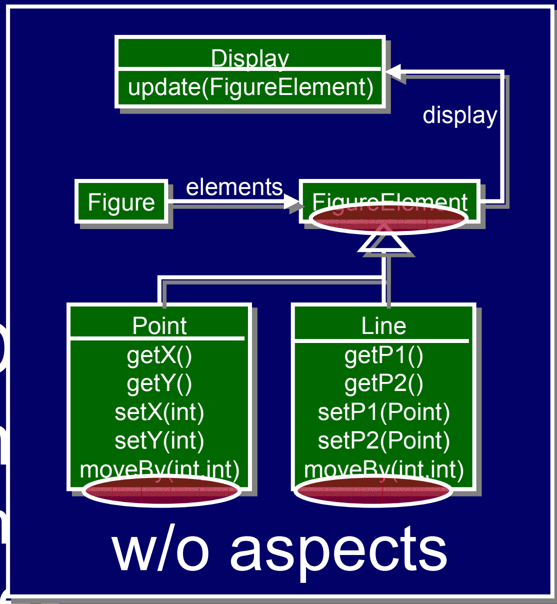
related work

- comparison ***two*** AOP mechanisms;
  e.g., Demeter vs. AspectJ [Lieberherr97]
- formal models for ***particular*** AOP mechanism
  [Wand+01], [Lämmel01], etc.
- Filman-Friedman's claim on non-invasiveness, or "quantified assertions over programs written by oblivious programmers"
  - not explicit in our framework;
    suggesting invasive AOP mechanisms is possible

summary

- 3 part modeling framework
  - elements from A&B meet at JP in X
  - based on executable implementations
    www.cs.ubc.ca/labs/spl/projects/asb.html
- explanation of modular crosscutting
  - in terms of projections of A&B modules into X
- future work:
  - discuss more features in AOP on the framework
    e.g., non-invasiveness, remodularization, …
  - unified implementation and formalization
  - apply to foundational work: semantics[Wand01,02], compilation[Masuhara02,03], new feature designs…

TRAV – traversals



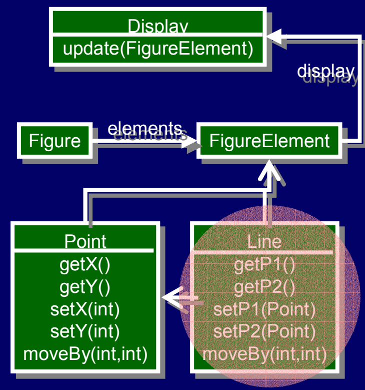
- based on Demeter/DemeterJ/D
- traversals through object graph
  - specification: “where to go” & “wh
  - otherwise scattered among classes
- e.g., counting FigureElements in a Figure

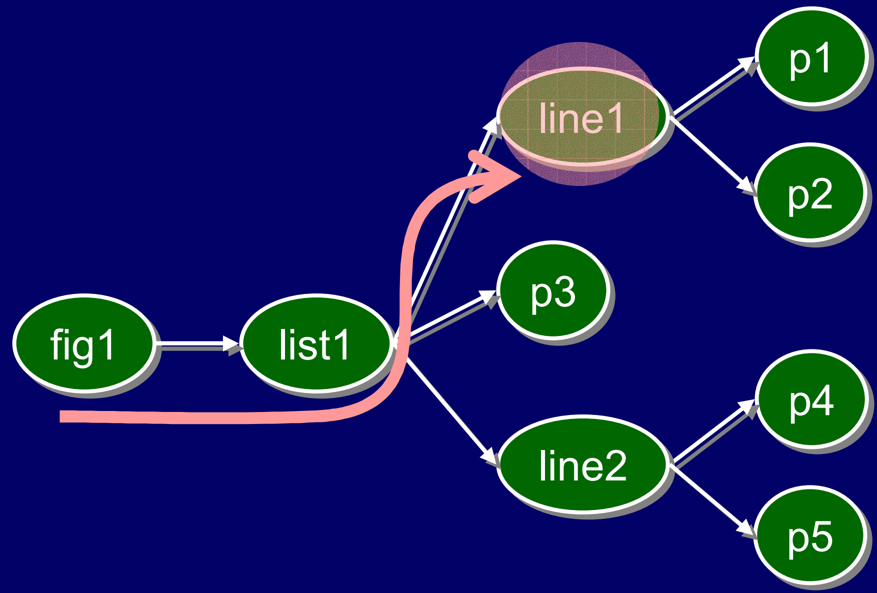
```
Visitor counter = new CountElementsVisitor( );
traverse("from Figure to FigureElement",
         fig,
         counter);
```

implementation of TRAV

- semantics:
  visit all objects
  that can lead to goals
- naïve implementation:
  a traversal engine
  - walks over object graph
  - locates current object
    in the spec.
  - visits & continues
    walking if reachable
    in terms of class graph

```
from Figure
through Line
to FigureElement
```





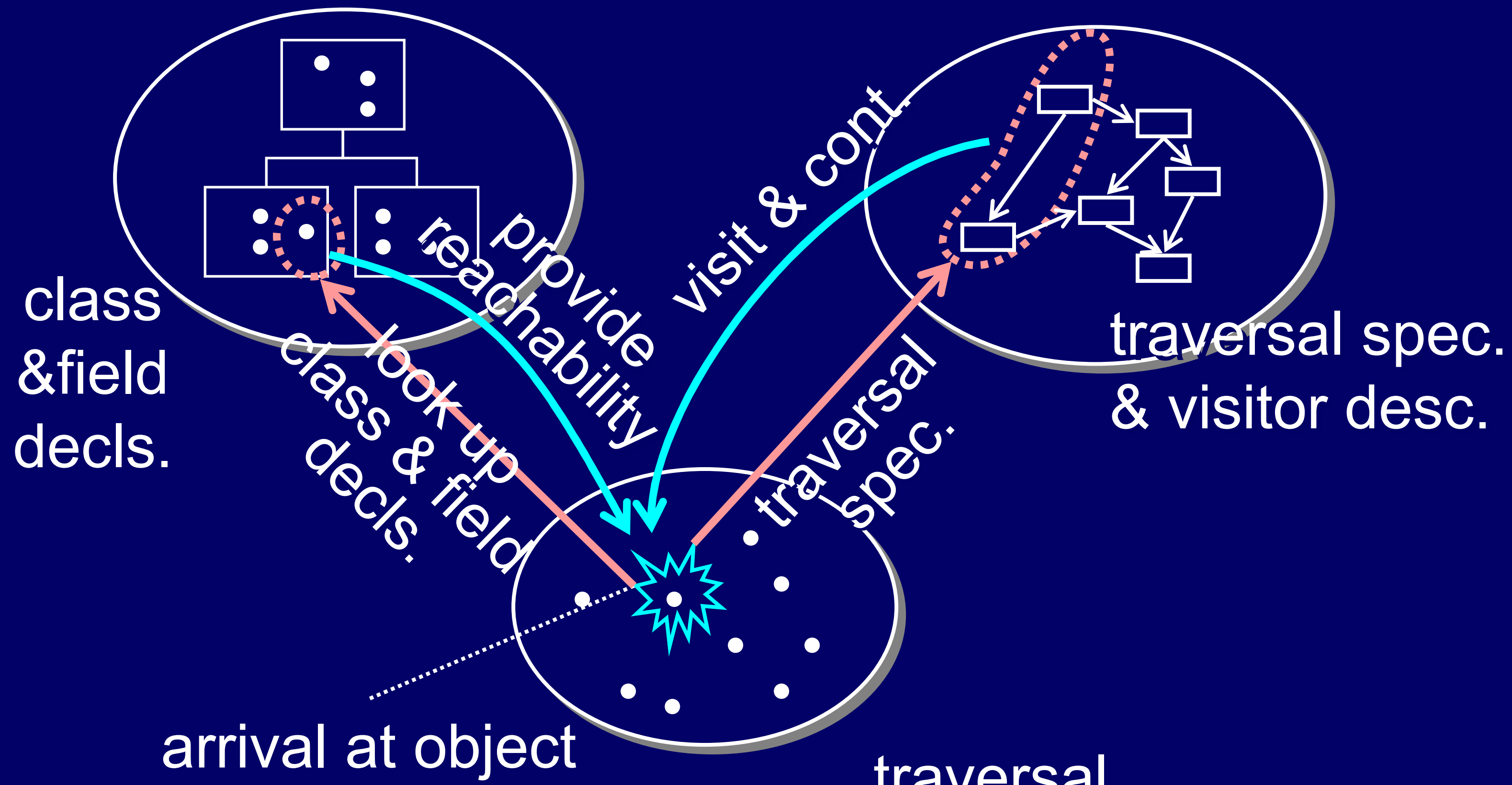
implementation of TRAV

- weaver = traversal engine

```
(define trav:weave
  (lambda (trav-spec root visitor)
    (let arrive ((obj  root)
                 (path (make-path (object-cname root))))
      (call-visitor visitor obj)
      (for-each (lambda (fname)
                  (let* ((next-obj   (get-field fname obj))
                         (next-cname (object-cname next-obj))
                         (next-path  (extend-path path
                                                  next-cname)))
                    (if (match? next-path trav-spec)
                      (arrive next-obj next-path))))
                (object->fnames obj)))))
```

- visit object
- match path vs. spec
- and cont.

model for TRAV



OC – open classes

- based on AspectJ's introductions [Kiczales01]
  Flavors, etc. [Cannon82]...
- can declare methods/fields
  outside of the class declarations
- example: add drawing functionality

```
class DisplayMethods {
 void Point.draw() { Graphics.drawOval(...); }
 void Line.draw()  { Graphics.drawLine(...); }
}
```

implementation of OC

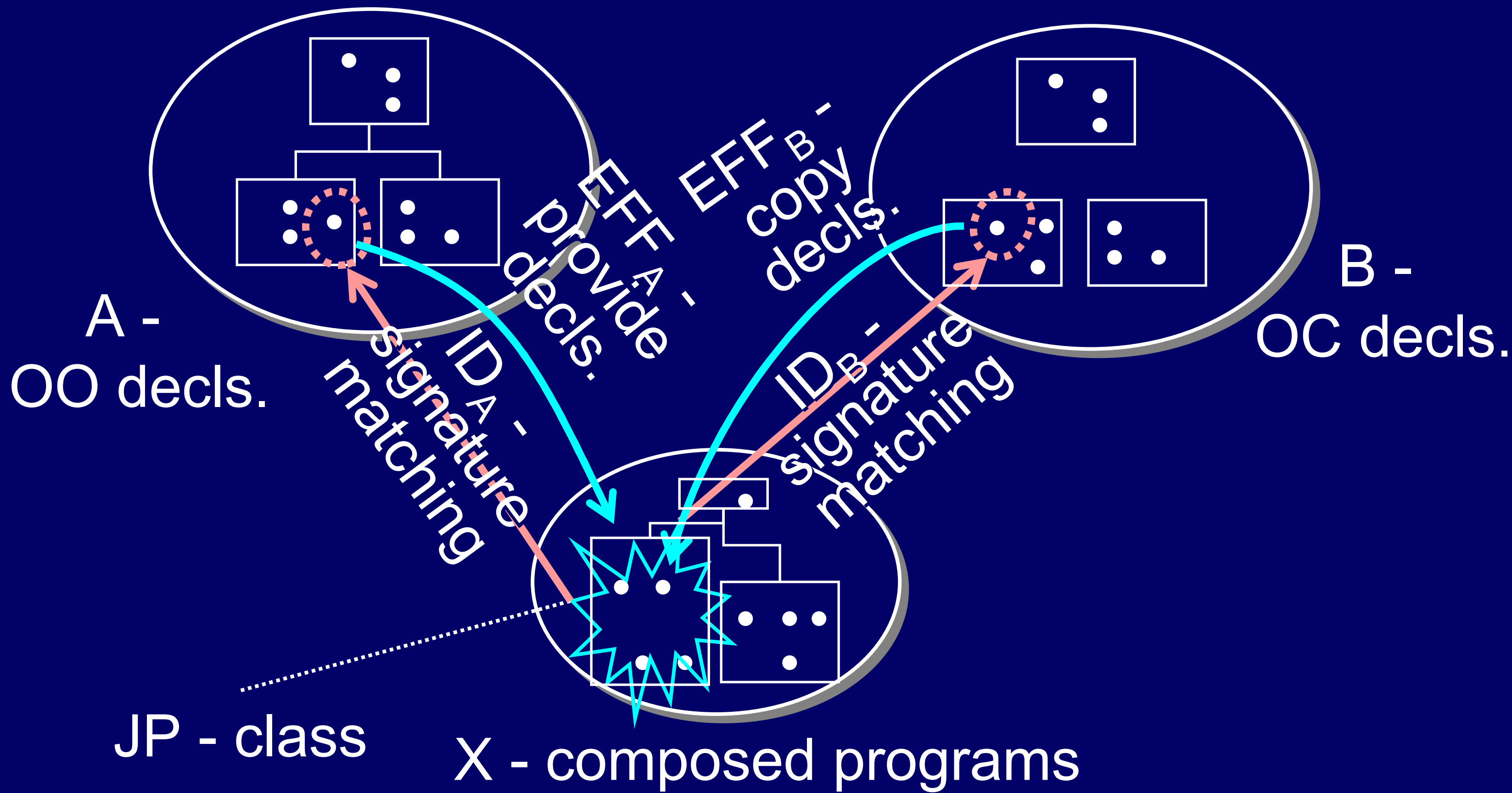
- a special case of COMPOSITOR
  - a source-to-source translator
  - class decls × oc decls → program

```
(define oc:weave
 (lambda (pgm);-> pgm
  (let ((pgm       (remove-oc-mdecls pgm))
        (oc-mdecls (gather-oc-mdecls pgm)))
   (make-pgm
    (map (lambda (cdecl)
          (let* ((cname (class-decl-cname cdecl))
                 (sname (class-decl-sname cdecl))
                 (per-class-oc-mdecls
                   (lookup-oc-mdecls cname oc-mdecls)))
           (make-class-decl cname sname
            (append (class-decl-decls cdecl)
                    (copy-oc-mdecls cname per-class-oc-mdecls)))))
         (pgm-class-decls pgm))))))
```

- A
- B
- a new program in X
- cdecl is a jp

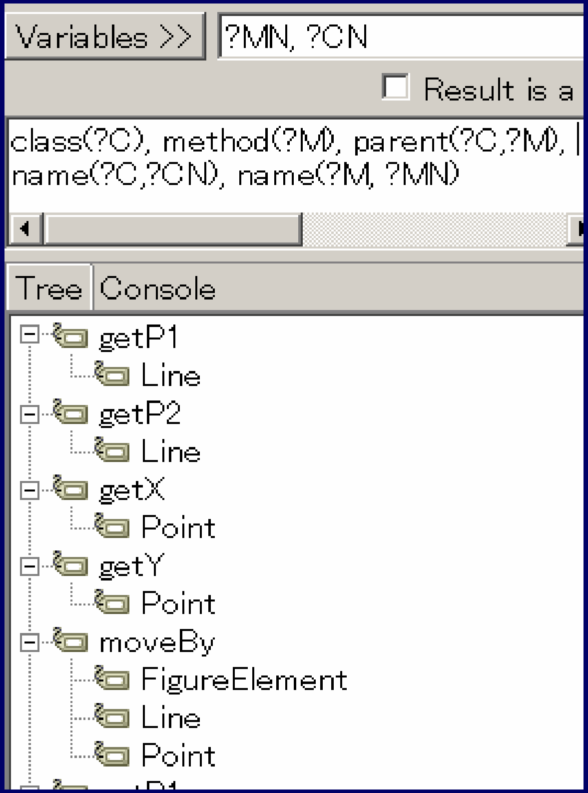
- ID_B
- EFF_A and EFF_B

model for OC

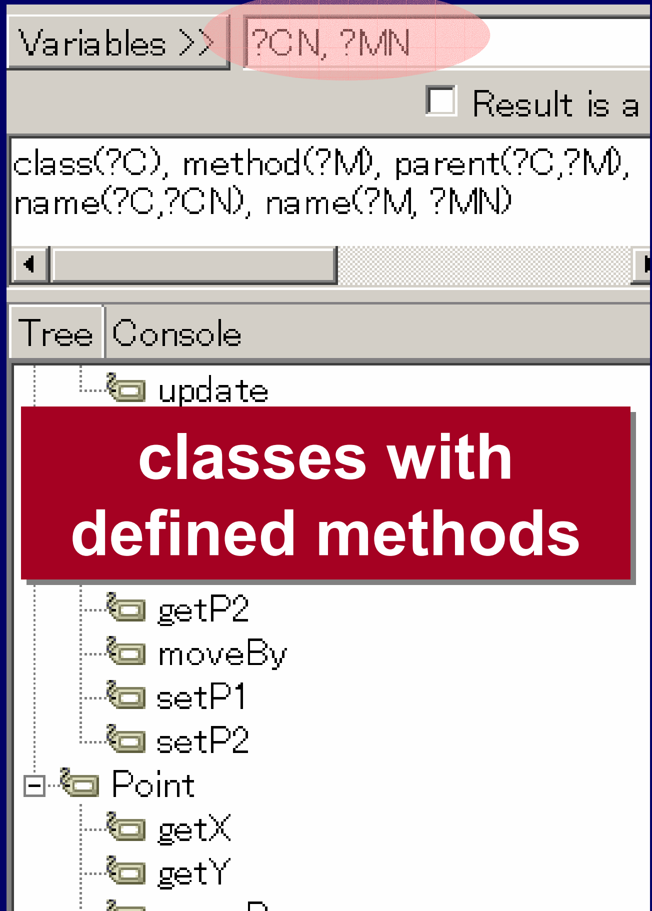

EFF_A - provide decls.
EFF_B - copy decls.
A - OO decls.
B - OC decls.
ID_A - signature matching
ID_B - signature matching
JP - class
X - composed programs

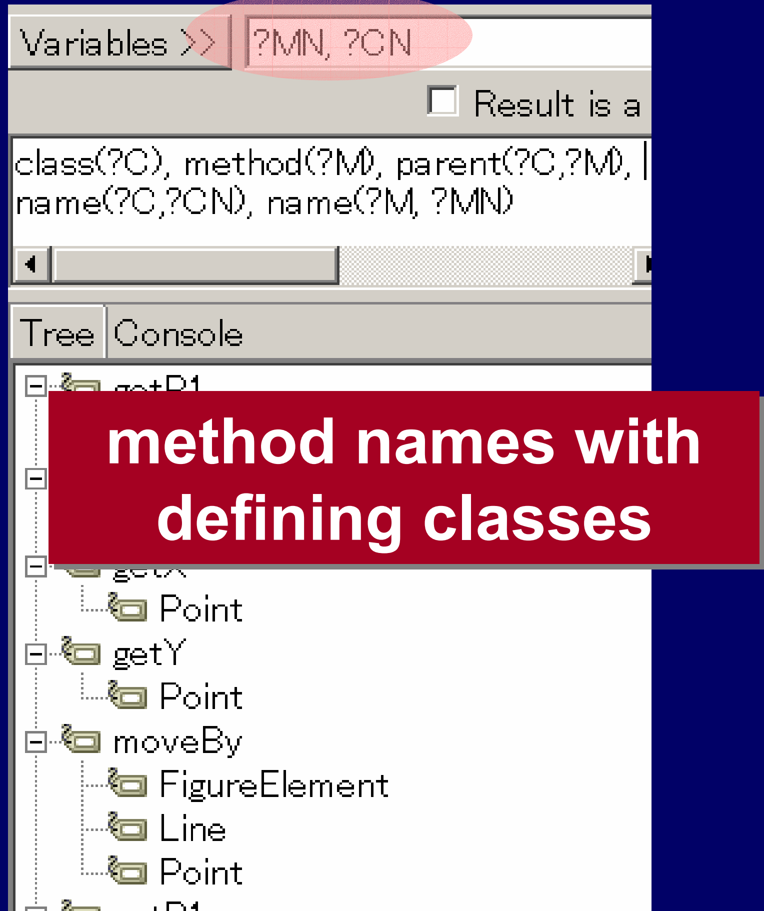
QB – query-based browser

- a customizable code exploration tool [Rajagopolan02]
- takes parameters:
  - properties to extract
  - order of properties
- can give different views of a program;
  e.g., group classes by method names



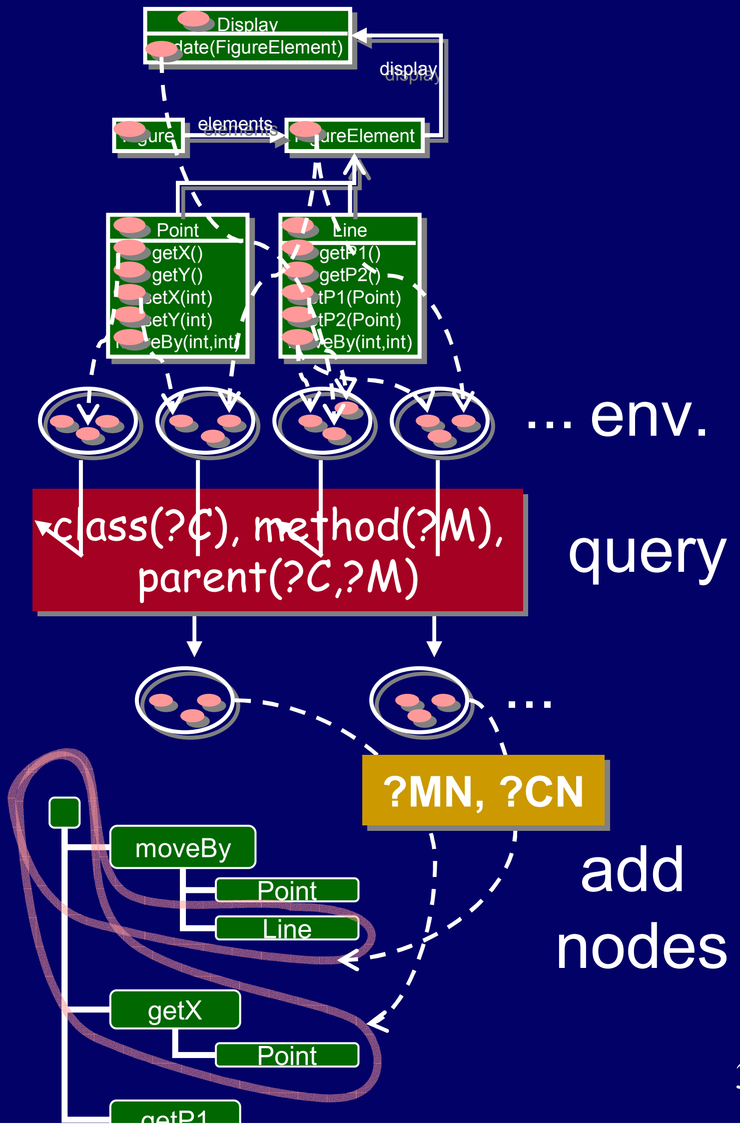
QB – query-based browser





implementation of QB

1. extract metaobjects
2. build envs.
3. test query against each env
4. add nodes to tree guided by the var. list

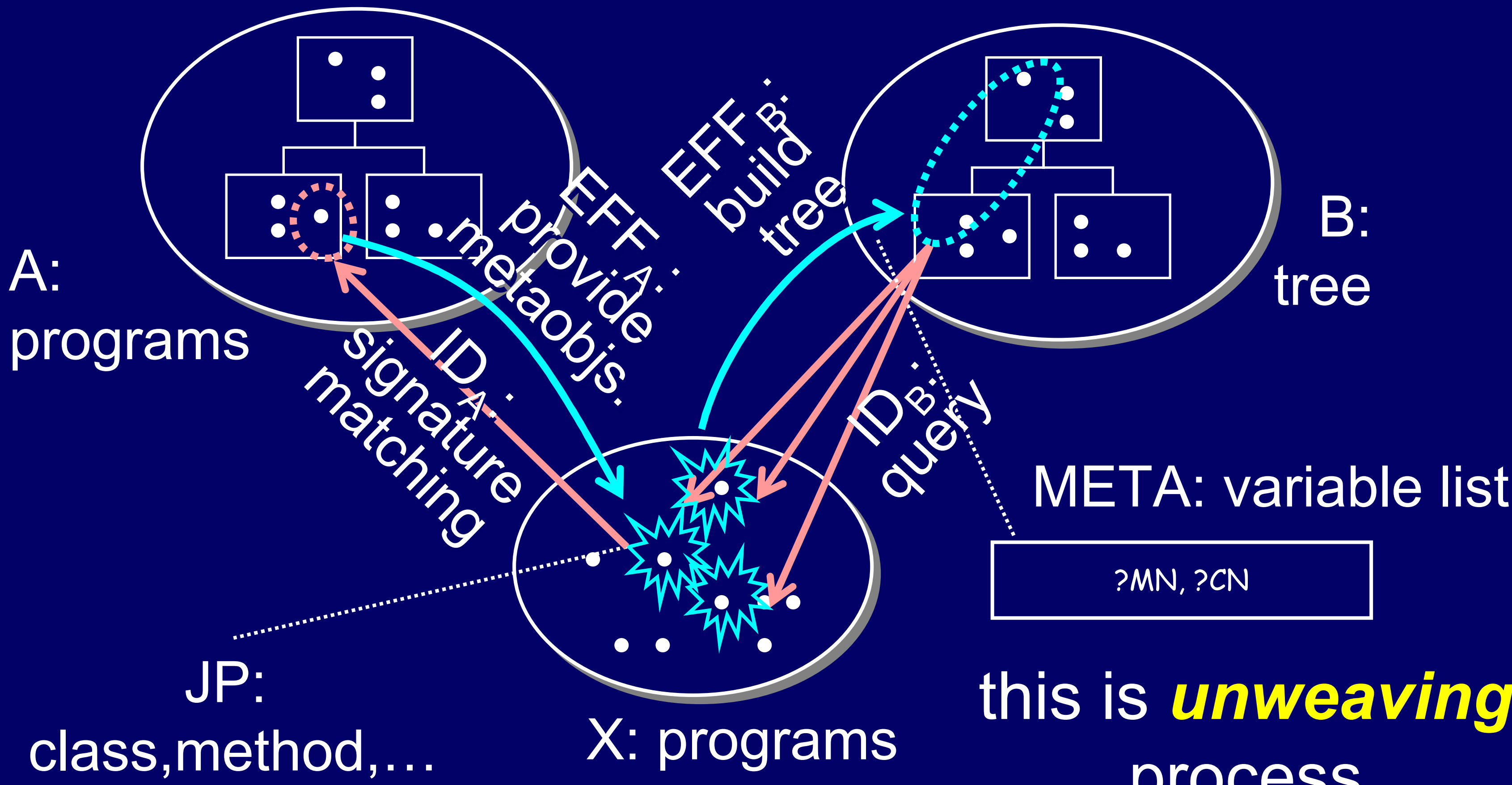


implementation of QB

```
(define qb:unweave
  (lambda (pgm query tree-vars);->tree
    (let* ((metaobjects (elaborate-program pgm))
           (all-envs    (possible-envs (query-vars query)
                                       metaobjects))
           (tree        (make-empty-tree)))
      (for-each (lambda (env)
                  (if (match? env query)
                      (let ((vals (map (lambda (var)
                                         (lookup-var var env))
                                       tree-vars)))
                        (add-to-tree! vals tree))))
        all-envs)
      tree)))
```

- A to isomorphic X
- tuples of jps
- B
- ID_B
- EFF_B
- return B

model for QB



this is ***unweaving*** process