Programming Language Design

2015

Week #6: Beyond Hierachical Decomposition

Instructor: Hidehiko Masuhara

Quiz(1/2) Design a class hierarchy for bank customers (10 min.)

Axis 1: There are three types of customers: corporate, elite and individual. A corporate customer has a company name and a corresponding person. An elite or individual customer has a name and a birthday. Those customers are different in:

- calculating transaction fees, and
- writing recipient names on messages

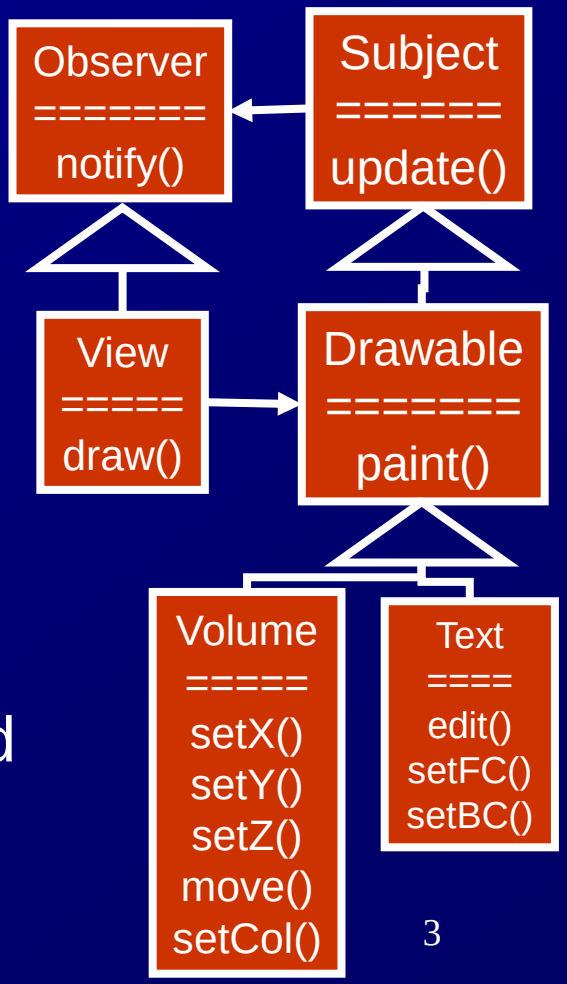
Axis 2: There are two types of customers: domestic and international. A domestic customer's address includes a ZIP number, a prefecture, a city, and a house number. An international customer's address includes a country name and a (free-form) address text. Those customers are different in:

- sending postal notes,
- calculating taxes, and
- calculating transaction fees (additional fees for international transactions).

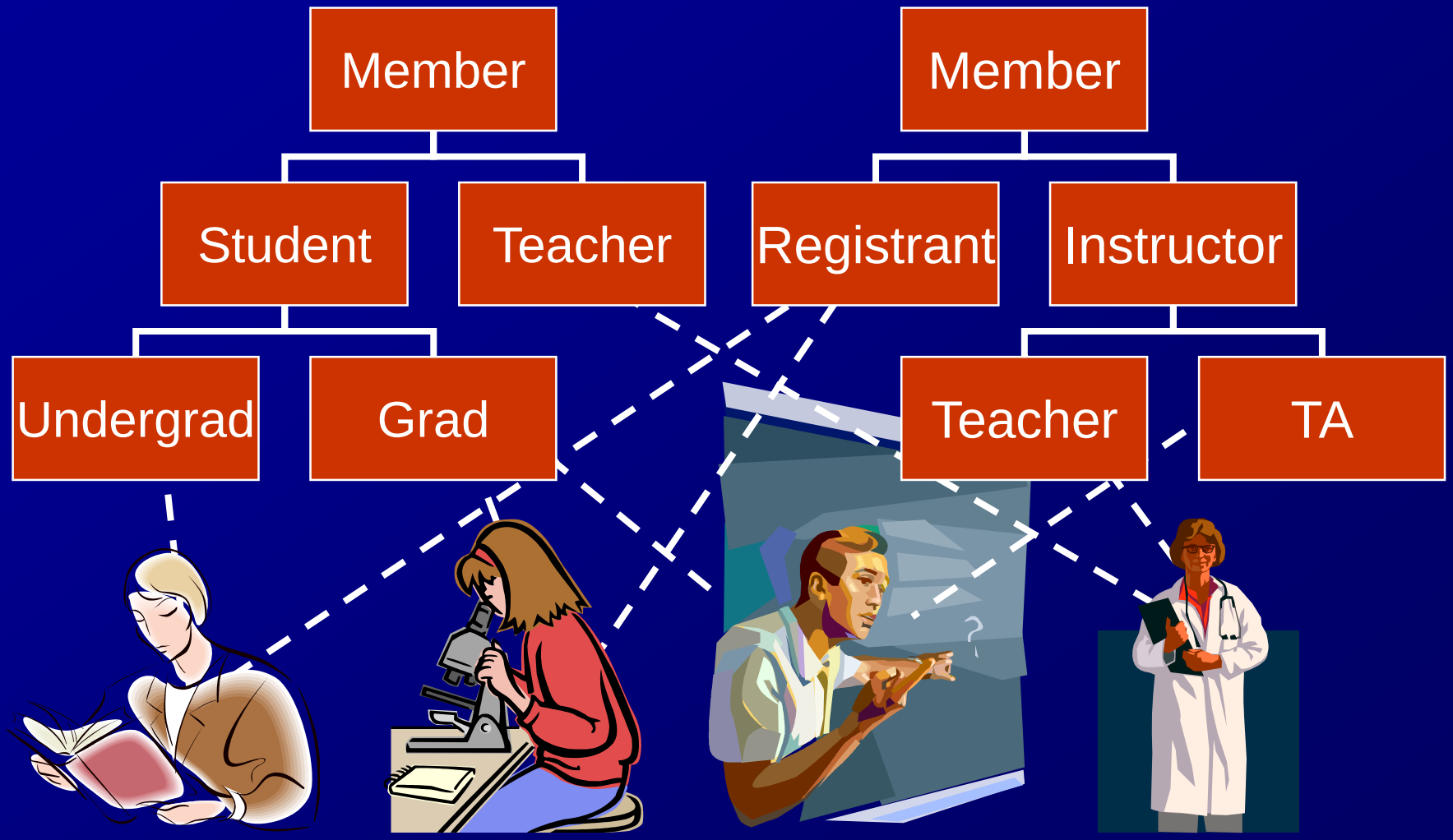
Quiz(2/2) Critique of the Observer Pattern (10min.)

Critisize this design based on the Observer Pattern

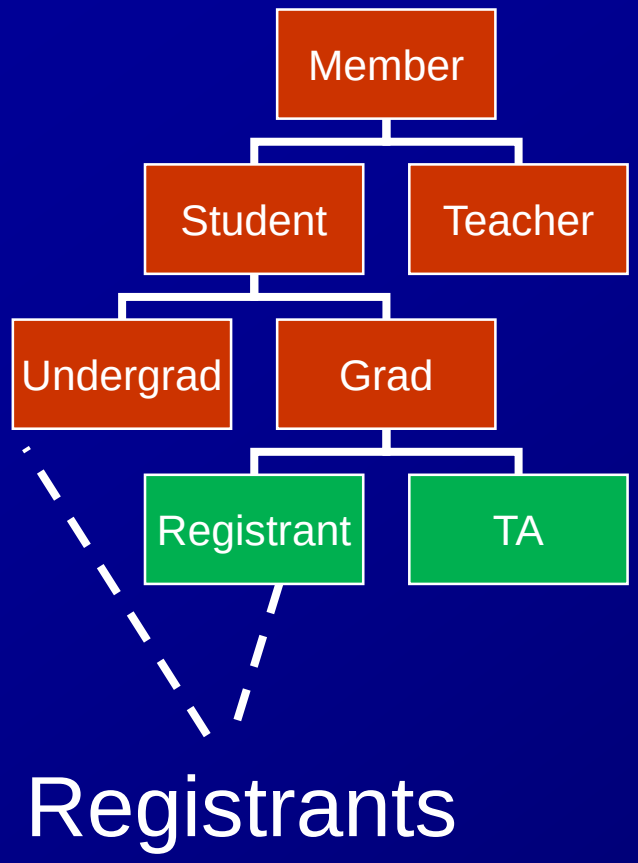
- There are more Drawable sub-classes other than Volume, and Text
- setX, setY, setZ, move of Volume and edit of Text calls update
- Possible future extension: fast edit mode---in this mode, screen will not be redrawn if only color is changed

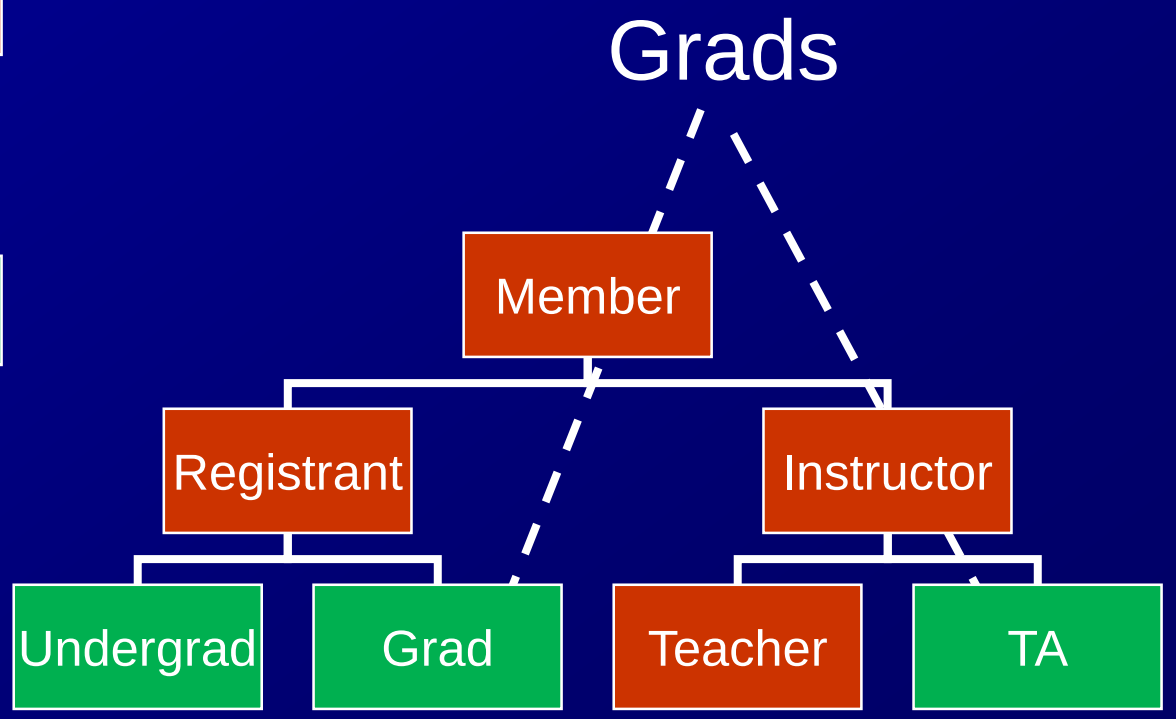


Hierarchical Decompositions from Different Viewpoints [HO93]

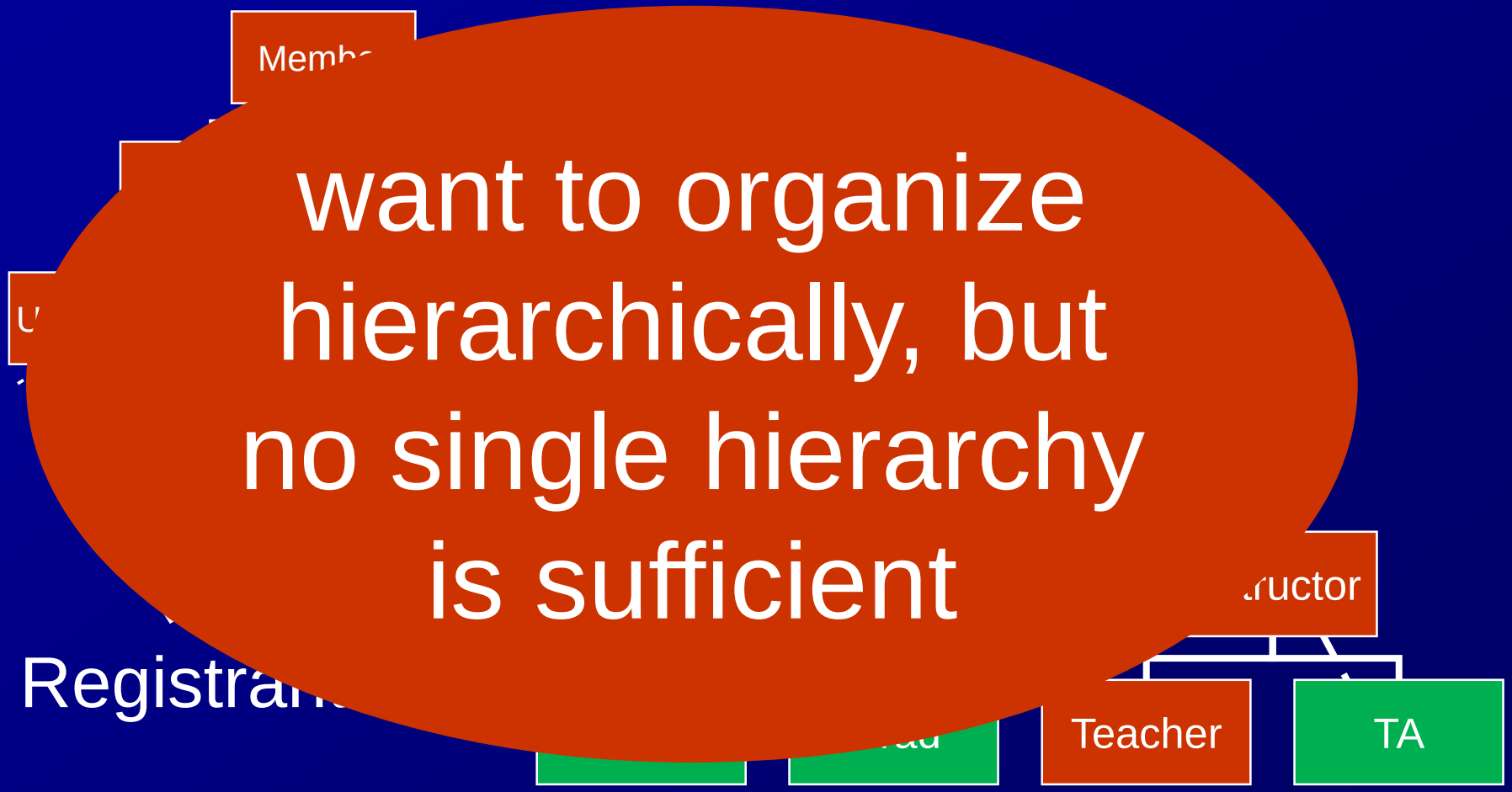


Hierarchical Decompositions from Different Viewpoints [HO93]





Hierarchical Decompositions from Different Viewpoints [HO93]



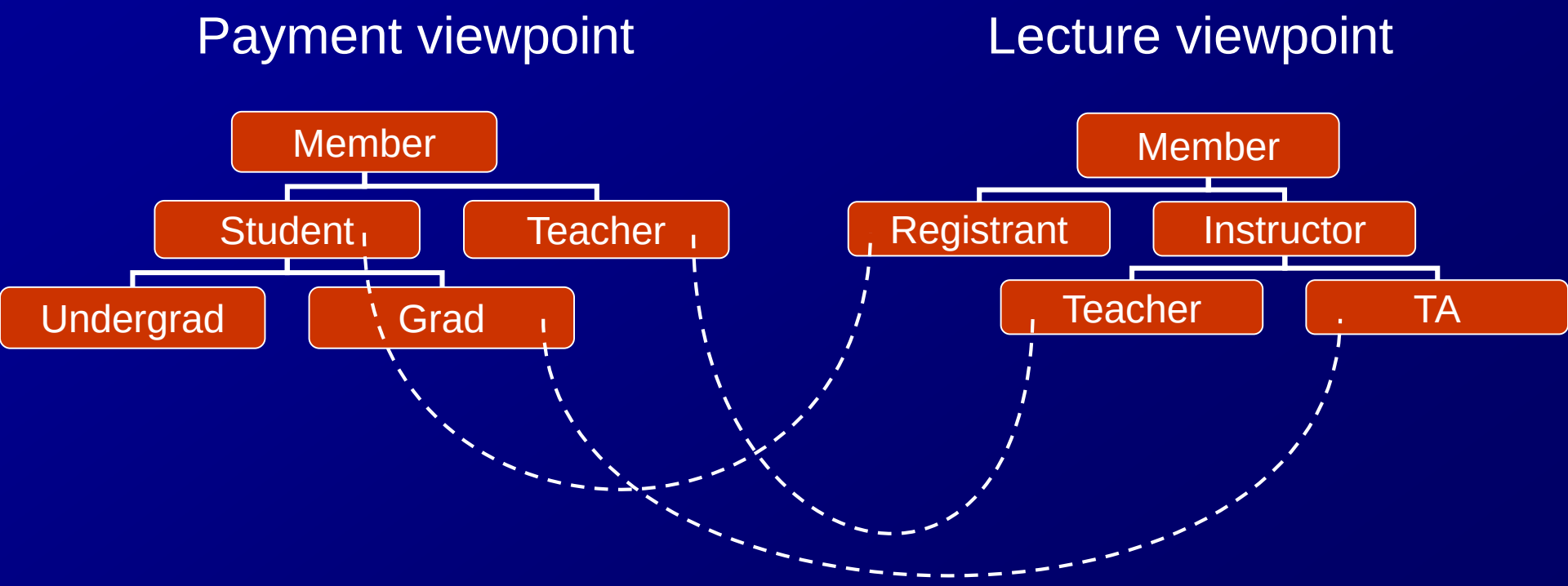
Subject-Oriented Programming [HO93]

- Multiple hierarchical decompositions from different viewpoints
- Approach: composition of hierarchies
- Language: Hyper/J

Hyper/J Language

- Subject-oriented extension to Java
- Programmers construct a class hierarchy for each viewpoint
  - A class in a hierarchy may appear in another hierarchy
  - allows many-to-many correspondence between classes
- Compiler composes class hierarchies, given specifications of
  - correspondence between classes
  - correspondence between methods (eg, overriding, serializing)
- (Tool to extract a partial class hierarchy from an existing program)

School Management System in Hyper/J



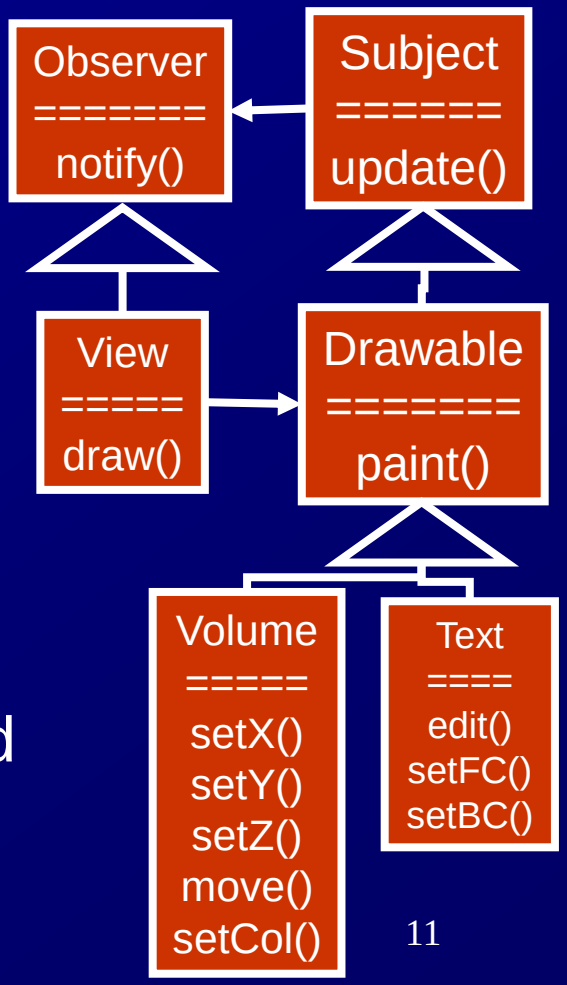
Problems in Subject-Oriented Programming

- Tedious to write class/method correspondences (no good languages)
- Unclear semantics
  - Composition of two hierarchies may result in a contradicting hierarchy
  - Hard to grasp program behaviors without having a composed hierarchy in mind

Quiz(2/2) Critique of the Observer Pattern (10min.)

Critisize this design based on the Observer Pattern

- There are more Drawable sub-classes other than Volume, and Text
- setX, setY, setZ, move of Volume and edit of Text calls update
- Possible future extension: fast edit mode---in this mode, screen will not be redrawn if only color is changed



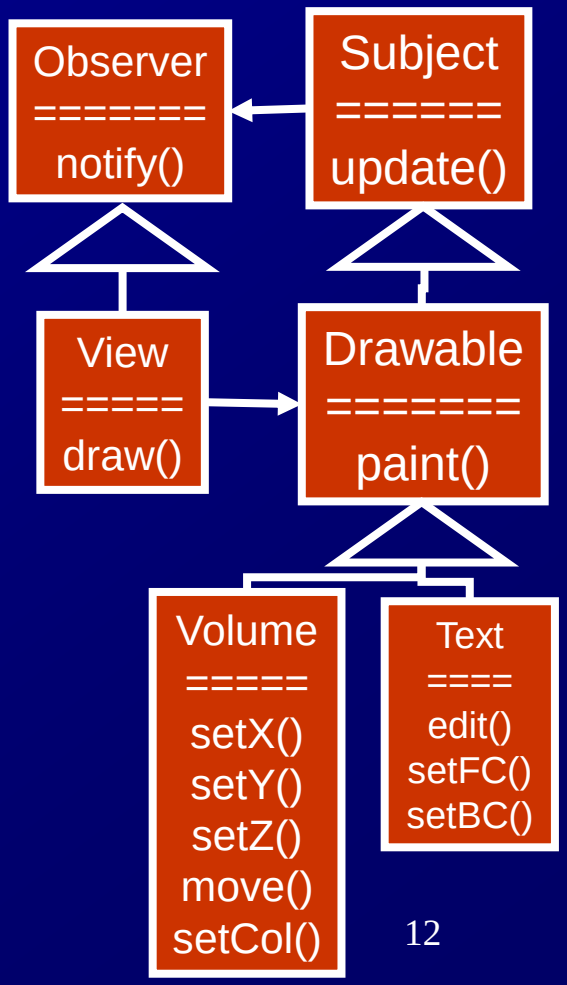
Anti-Hierarchical Extension

- Possible future extension: fast edit mode---in this mode, screen will not be redrawn if only color is changed

→extension to the policy of the update-triggers

"what triggers view updating?"

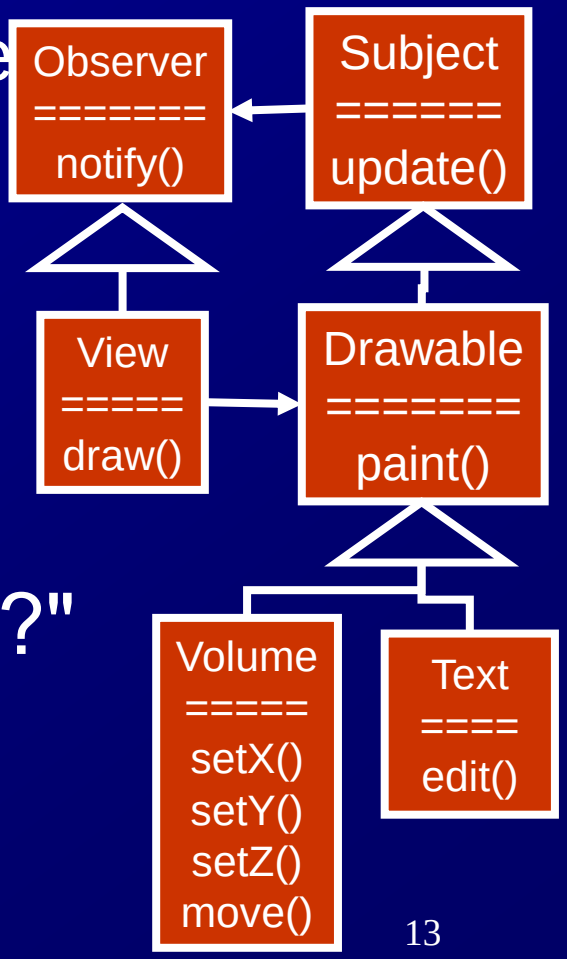
- With the Observer Pattern, the policy is hard coded by embedding update() calls



Anti-Hierarchical Extension

- setX, setY, setZ, move of Volume and edit of Text calls update
  - move calls setX, setY, setZ
  - setX, setY, setZ calls update
  - one move call results in 3 update calls

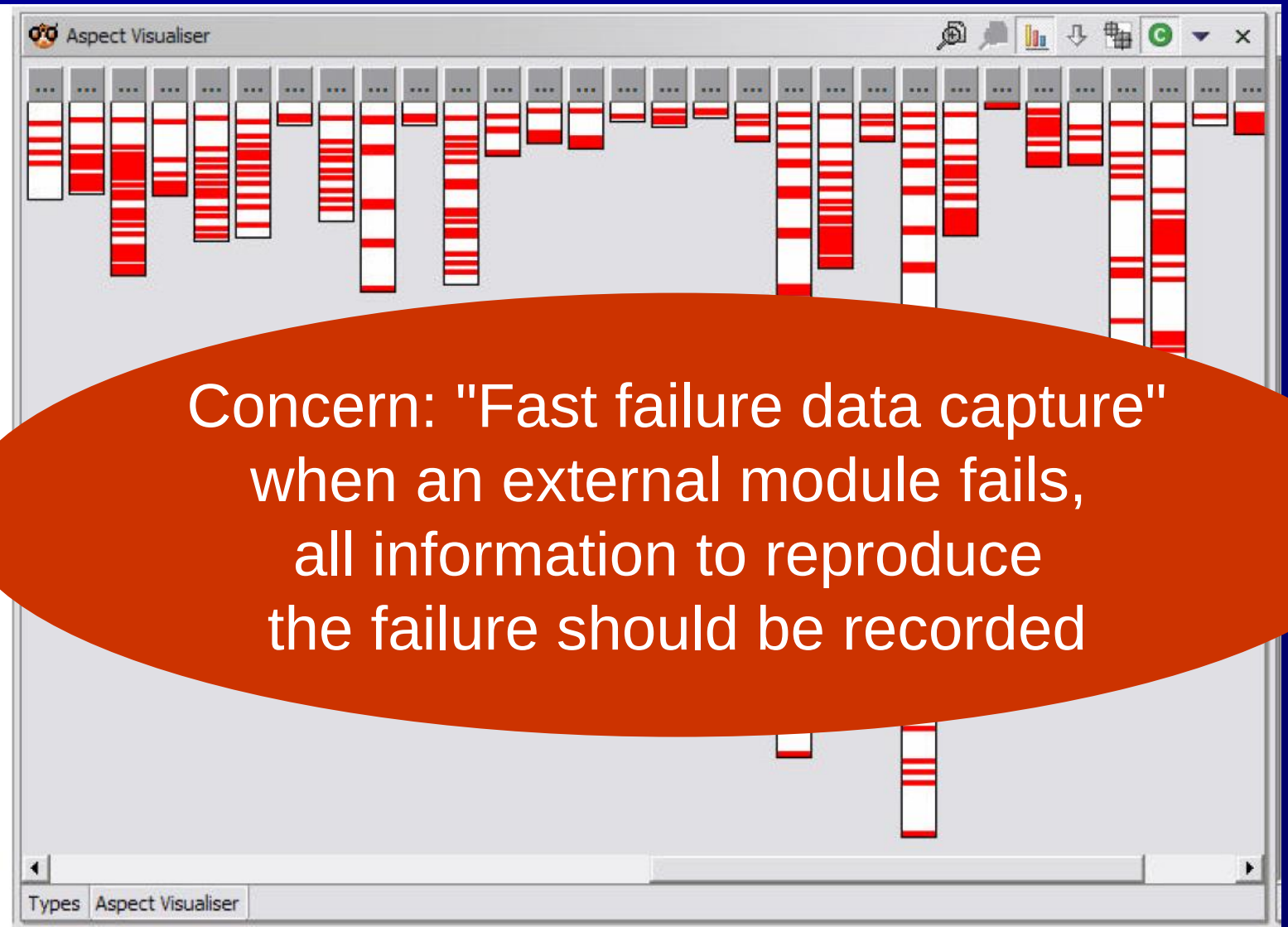
→"what triggers view updating?" is hardcoded



Crosscutting Concerns: Scattering and Tangling of Concerns

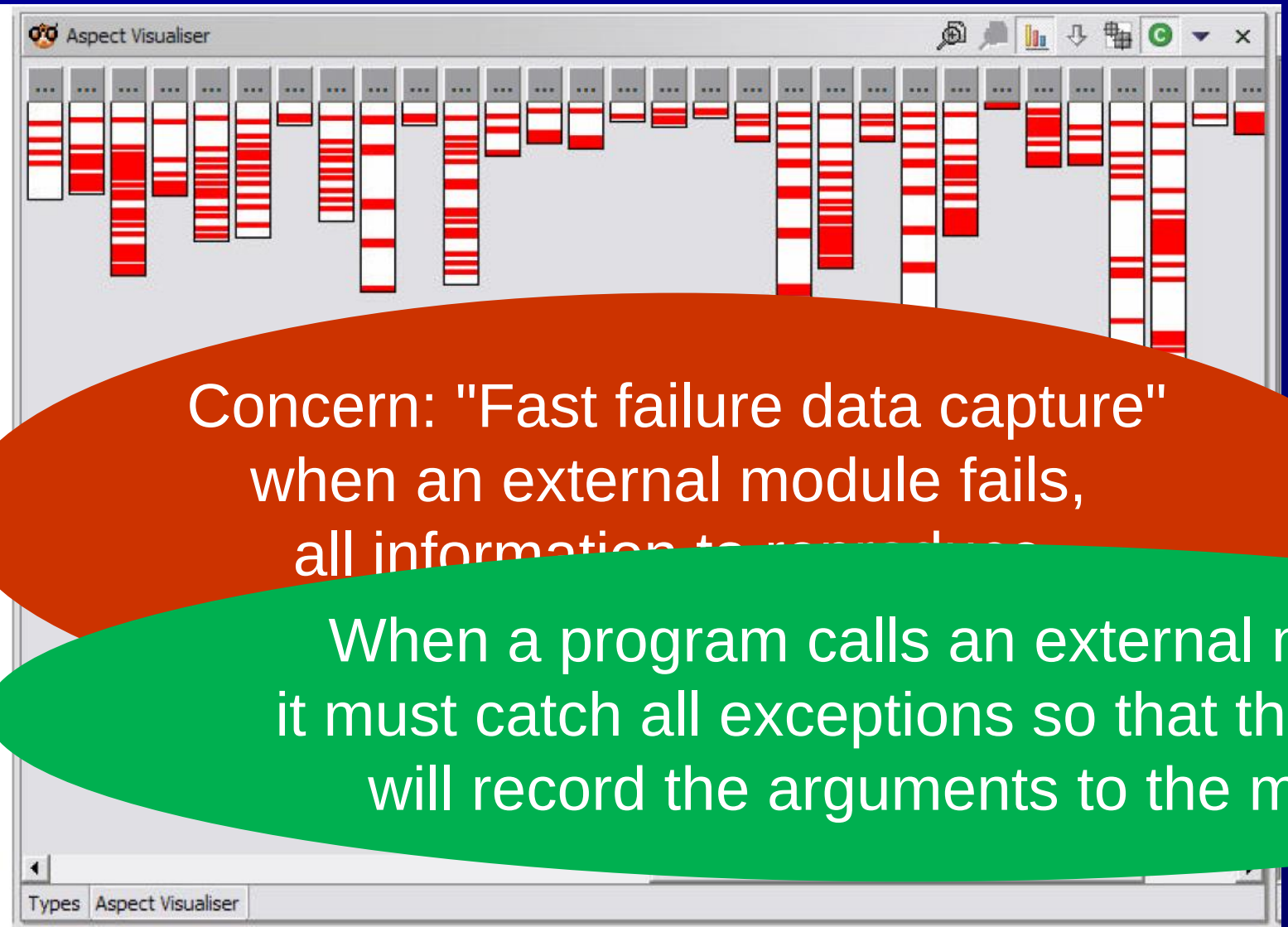
- Crosscutting concerns: a concern whose implementation spans across multiple modules
  - a concern "what triggers view updating?" is hardcoded as update calls in many methods in many classes
- Concern: a group of artifacts in software that a human considers as one (a function, a performance requirement, etc.)
  - eg. "update a screen when an element moves"
- Scattering: a symptom that the implementation of a concern appears in many modules
- Tangling: a symptom that a description of a module involves with multiple concerns

Scattering [Colyer+03]



Concern: "Fast failure data capture"
when an external module fails,
all information to reproduce
the failure should be recorded

Scattering [Colyer+03]

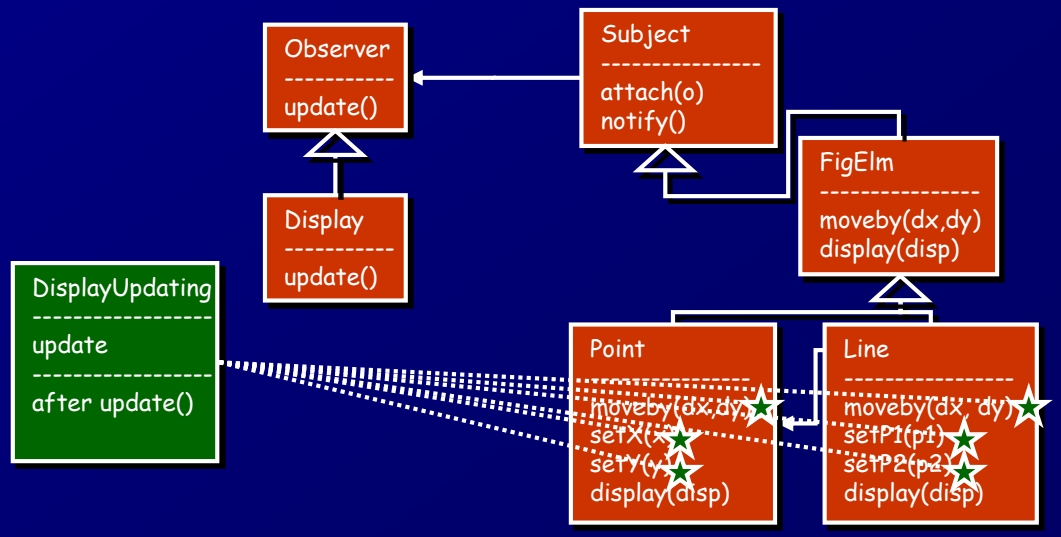


Concern: "Fast failure data capture"
when an external module fails,
all information to reproduce

When a program calls an external module,
it must catch all exceptions so that the handler
will record the arguments to the module

Aspect-Oriented Programming [Kiczales+97]

- A module that adds/modifies across a class hierarchy; called an aspect
  - Novelty: a pointcut language to describe actions across modules



AspectJ [Kiczales+01]

- The best known AOP language
  - extension to Java
- Used in industry
- Many follow-up languages
  - JBoss AOP, AspectWerks, Spring AOP, Seasar 2 AOP, etc.
  - C, C++, C#, Python, Ruby, Smalltalk, ...

An example of aspect: "update display when figures move"

- Pointcut: "move is when Point:: setX is called, or ..."
- Advice: "execute redraw() after move"
- Inter-type declaration: adds methods & fields to other classes
- Aspect: a module like a class

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

   after() : move() { Display.redraw(); }

   void FigElm.draw(Display d);
   void Point.draw(Display d) { … }
   … }
```

AspectJ basics: advice declarations

Additional/alternative behaviors

- before/after/instead of (modifier)
- an action (pointcut),
- do something (body)
  - Java statement

many variations (later)

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() : move() {
    Display.redraw();
  }

  Display d);
  y d) { ... }
```

after move(),
do redraw()

AspectJ basics: pointcuts

Timings of certain actions

- Kind of actions (method call, etc.)
- Method signature
- Composition

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() : move() { Display.redraw(); }

  void FigElm.draw(Display d);
  void Point.draw(Display d) { … }
  … }
```

when FigElm.moveby, Point.setX, or … is called

AspectJ basics: inter-type declarations

Add declarations to existing modules

- Existing modules
  - class & interface
- Declarations
  - method, field
  - extends/implements clauses

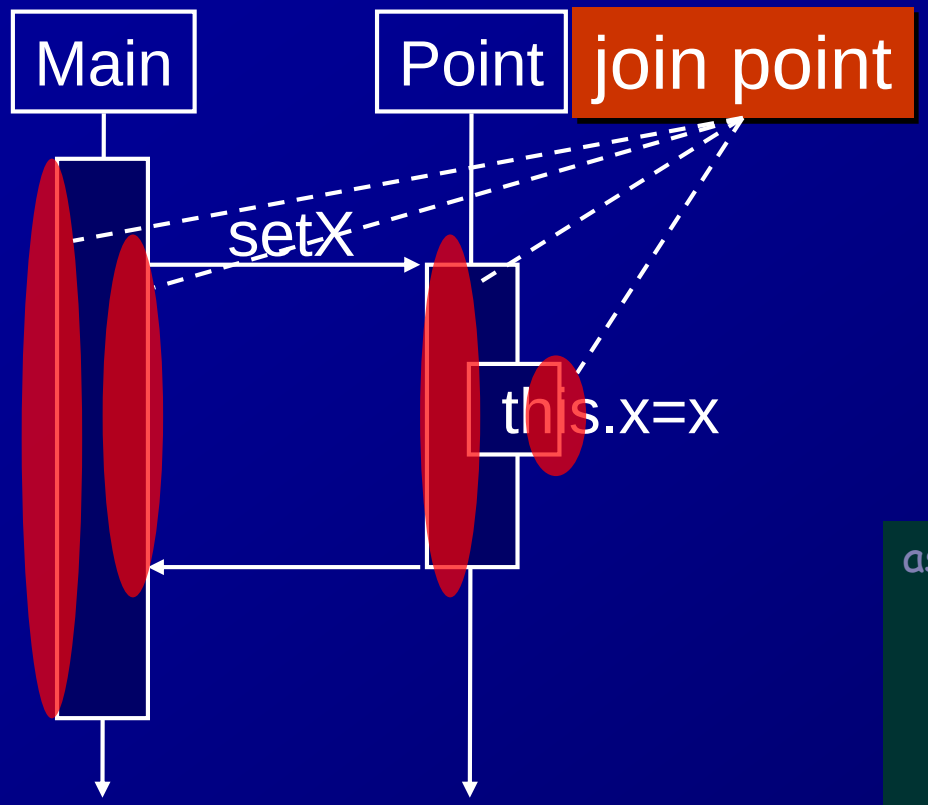
```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.move
    call(
    call(

  after() : move() { Display.redraw(); }

  void FigElm.draw(Display d);
  void Point.draw(Display d) { … }
  … }
```

add draw method to FigElm

```
declare parent MyTask:
   implements Runnable;
public void MyTask.run() { init(); }
```

let MyTask implement Runnable

AspectJ basics: join point model

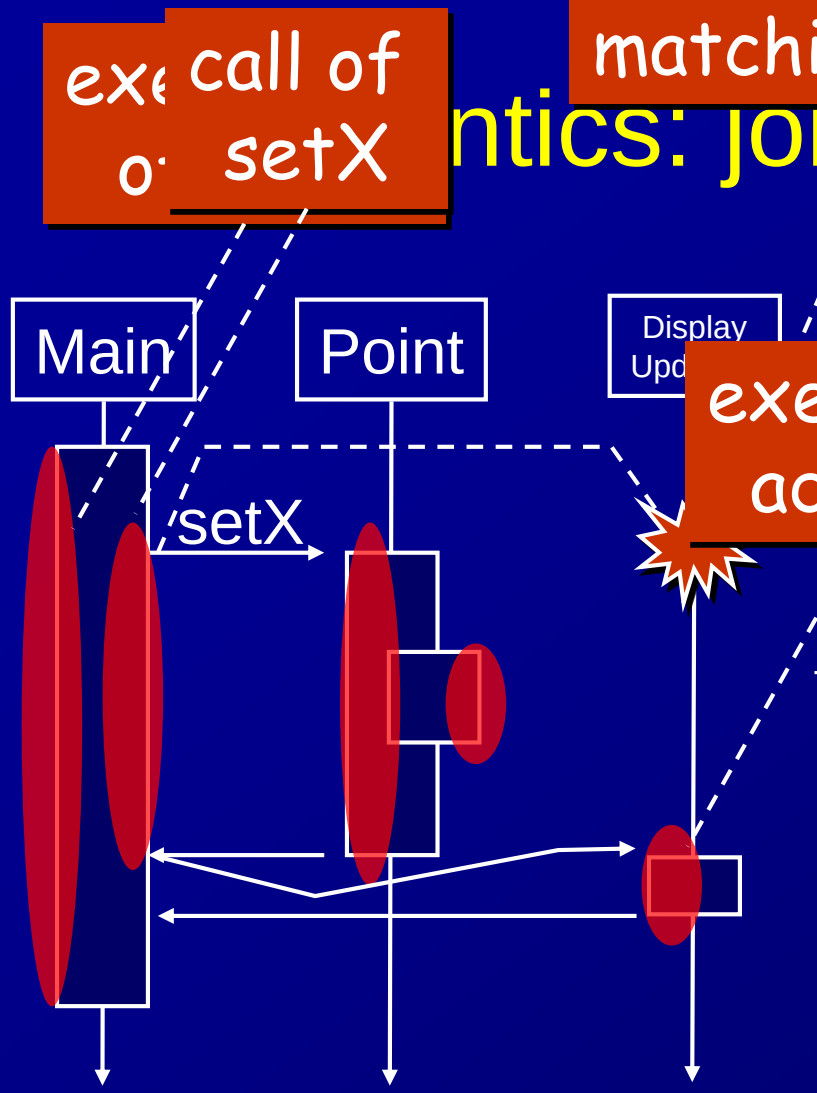


= runtime ***actions*** like method call, execution, field assignment

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) || …;
  after() : move() { … }
  void FigElm.draw(Display d);
  void Point.draw(Display d) { … }
  … }
```

ntics: join point model

conceptual model



- Whenever a join point is ed, collect advice that have matching pointcuts

→run body of before; body of original join point; body of after

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) || …;
  after() : move() { … }
  void FigElm.draw(Display d);
  void Point.draw(Display d) { … }
  … }
```

Pointcuts

- Novel in AOP
- Crosscutting abstraction of join points
  - Crosscutting: related to multiple modules
  - Abstraction:
    - disregarding details of join points
    - giving names
- Many variations

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() : move() { Display.redraw(); }
  … }
```

Kinds of pointcuts

- Specifies kinds and signatures of join ponts:

  call, execution, set, get, initialization, preinitialization, staticinitialization, handler, adviceexecution
- Checks & extracts of runtime type/context information:

  this, target, args, @annotation
- Confines in scope: within, withincode, cflow
- Arbitrary expression: if
- Logical Operator: &&, ||, !

Pointcut: wildcard

- Movement of a figure element: "when moveBy of FigElm, or any method begins with "set" of any FigElm subclass is called"

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(* FigElm+.set*(..));

  after() : move() { Display.redraw(); }
  … }
```

Pointcut: other conditions

- Various types of conditions
  - within(myapp.db..*): within DB package
  - set(int Point.x): assignment to x of a Point
  - withincode, execution, get, handler, initialization, static initialization
- Combinations
  - call(* javax.swing..*(..)) && !within(myapp.ui..*)):
    use of Swing framework outside from UI package

Pointcut: Extracting Context Information

- Context info.: values inside a join point
  - eg: caller, receiver, arguments
- Advice can use them as args.

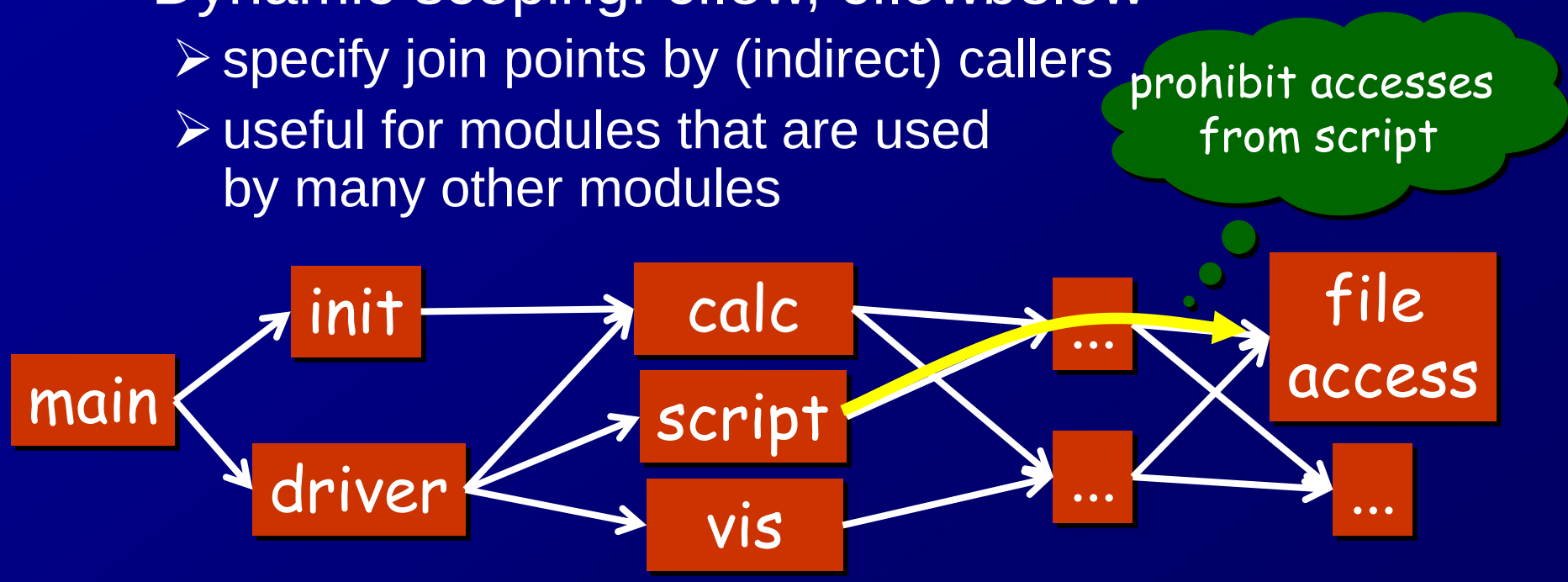
```
aspect DisplayUpdating {
  pointcut move(FigElm fig) :
    (call(int FigElm.moveby(int,int)) ||
     call(void FigElm+.set*(..)))
    && target(fig);

  after(FigElm fig) : move(fig) {
    Display d = fig.getDisp();
    d.redraw(fig);
  }
}
… }
```

update only displays that are showing the moved figure

Bind the receiver object to fig

Pointcut: Scoping

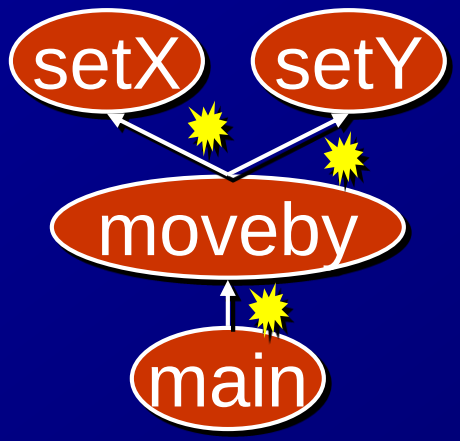
- Static scoping: within, withincode
  - specify join points generated in specific packages, classes or methods
- Dynamic scoping: cflow, cflowbelow
  - specify join points by (indirect) callers
  - useful for modules that are used by many other modules



Poincut: Dynamic Scoping (1/2)

- p.moveby(2,3); calls redraw 3 times

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(* FigElm+.set*(..));

  after() : move() { Display.redraw(); }
  … }
```

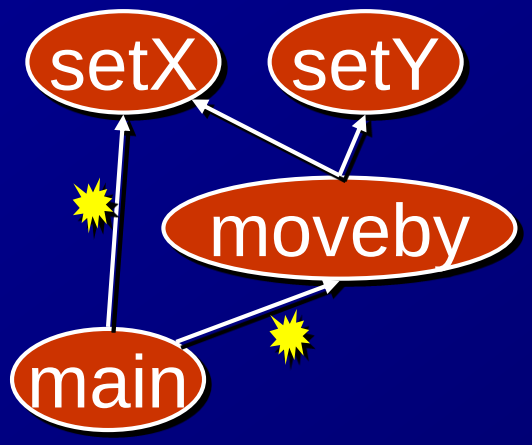



```
class Point … {
  void moveby(int dx, int dy) {
    setX(getX()+dx);
    setY(getY()+dy); } }
```

Pointcut: Dynamic Scoping (2/2)

- Restrict only when called from "outside" = "when no setX, setY & moveBy exist in the caller stack"

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(* FigElm+.set*(..));
  after() : move() &&
        !cflowbelow(move()) { … }
  … }
```




```
class Point … {
  void moveby(int dx, int dy) {
    setX(getX()+dx);
    setY(getY()+dy);  } }
```

Expressive Pointcuts

- Conditional pointcuts: specify join points based on arbitrary expressions
  - "when the argument to setX is negative", replace the argument with zero.
  - "when the receiver of compare is null", replace the receiver with a DefaultComparator
- Dataflow pointcuts [MK03]: specify join points based on relationship between values
  - "when an argument to HTML.append() depends on a return value from UserRequest.get()", quote special characters in the argument string
- Tracematch [Allan+05]: specify join points based on a regular expression over a sequence of join points
  - "when a FileReader object is called methods with this pattern: open(); (read() | write())*; close(); !close()"

Reusing Aspects: Abstract Aspect

- can define a general aspect and reuse it
  - by specifying ***what*** actions (advice) to be added, but
  - not by specifying ***where*** they are applied to

cf. abstract classes and abstract methods

  - specify ***the interface*** of existing actions, but
  - not specify ***what*** actions will be performed

Reusing Aspects: A Logging Aspect

- Abstract aspect:
  - specifies how to log
  - but not specifies where to log (via abstract pointcut)
- Concrete aspect:
  - instantiates a pointcut
  - no mention about the logging impl'n.

```
abstract aspect AbstractLogging {
  abstract pointcut logPoint();
  after() : logPoint() {
     …logging action… }
  … }
```

```
aspect DBLogging
        extends AbstractLogging {
  pointcut logPoint():
        call(* myapp.db..*.*(..));
}
```

References

[HO93] Harrison and Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In OOPSLA'93. pp.411-428. 1993.

[Colyer+03] Colyer, et al. Using AspectJ for component integration in middleware. In Practitioner Report, OOPSLA'03, 2003.

[Kiczales+97] Kiczales et al. Aspect-Oriented Programming. In ECOOP'97. pp.220-242, 1997.

[Kiczales+01] Kiczales et al. An Overview of AspectJ. In ECOOP’01. pp.327-353, 2001.

[MK03] Masuhara and Kawauchi, Dataflow Pointcut in Aspect-Oriented Programming, In APLAS'03, pp.105-121, 2003.

[Allan+05] Allan et al., Adding trace matching with free variables to AspectJ, In OOPSLA’05, pp. 345—364, 2005.