Lecture 16 & 17

Crosscutting Concerns
N-dimensional separation of concerns, AspectJ, Mixin, Concern Graph, etc.

This week's Agenda

- Presentations: Arasi Saravanan (skeptic)
- Problem Space
  - N dimensional separation of concerns by Peri Tarr et al.
  - Canonical example of multiple dimensions of concerns: e.g. abstract syntax tree example
  - Writing the AST example in a functional programming language
  - Writing the AST example code in an object-oriented language
  - QUIZ--Listen closely this lecture as you should be able to answer all questions based on this lecture.

This week's Agenda

- Solution Space
  - Language extension to support crosscutting concerns: AspectJ [Kiczales et al. 1997]
  - Language-based approach (language tweaking): Mixins, Using C++ templates to support flexible feature composition [VanHilst and Notkin 1996], etc.
  - Tool-based approaches: Concern graphs [Robillard and Murphy 2003], AspectBrowser [Griswold et al. 01], CME [Tarr et al.], etc.

Class Presentations

- Arasi (Skeptic)

Before we start our lecture

- Have you ever programmed in a functional programming language?
  - ML, Ocaml, Scheme, etc?
- If you haven't, after today's lecture, please review some web tutorials on ML or Ocaml.

Recap of "Information Hiding Principle"

- What is the Information Hiding Principle?
  - reduce unnecessary sharing or coupling
  - hide decisions that are likely to change into a module
  -

Recap of "Information Hiding Principle"

- What is the Information Hiding Principle?
  - Using C++ instead of C?
  - Using private fields instead of public?
  - Abstract the behavior and data?
  - Reduce dependencies between modules?

- Parnas' Information Hiding Principle
  - Hide design decisions that are likely to change

- Parnas' Information Hiding Principle
  - Hide design decisions that are likely to change
  - ≈ identify design decisions that are unlikely to change and fix them.

Any problems with the IH design principle?

- Difficult to identify what are likely to change?
- Widely spread impact?

Any problems with the IH design principle?

- How can you anticipate which design decisions are likely to change?
- What if there are multiple design decisions?

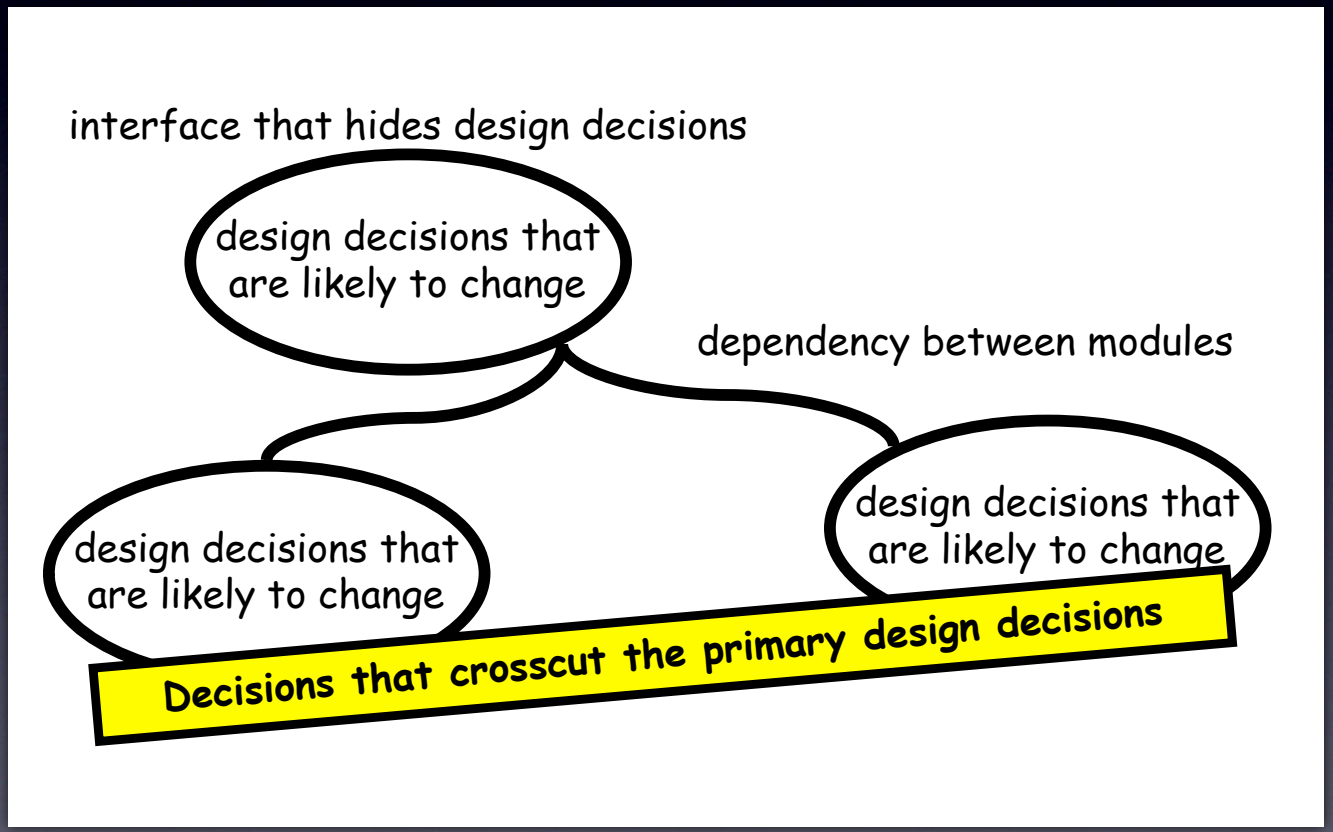
Primary vs. Secondary Design Decisions

- Primary design decisions:
  - Decisions that architects consider as the most important decisions
  - Decisions that are very unlikely to change
- Examples?
- layered architecture, pipe-line architecture.
- (Security, transaction)

Primary vs. Secondary Design Decisions

- Secondary design decisions
  - Less important than primary decisions
  - Decisions that architects did not anticipate in the beginning of system design
- Examples?
  - memory management
  - synchronization and logging

Primary Design Decisions
+ Secondary Design Decisions



Crosscutting Concerns

- Problem space:
  - What are examples of crosscutting concerns?
- Solution space:
  - To deal with crosscutting concerns during software evolution, what kinds of approaches do we have?

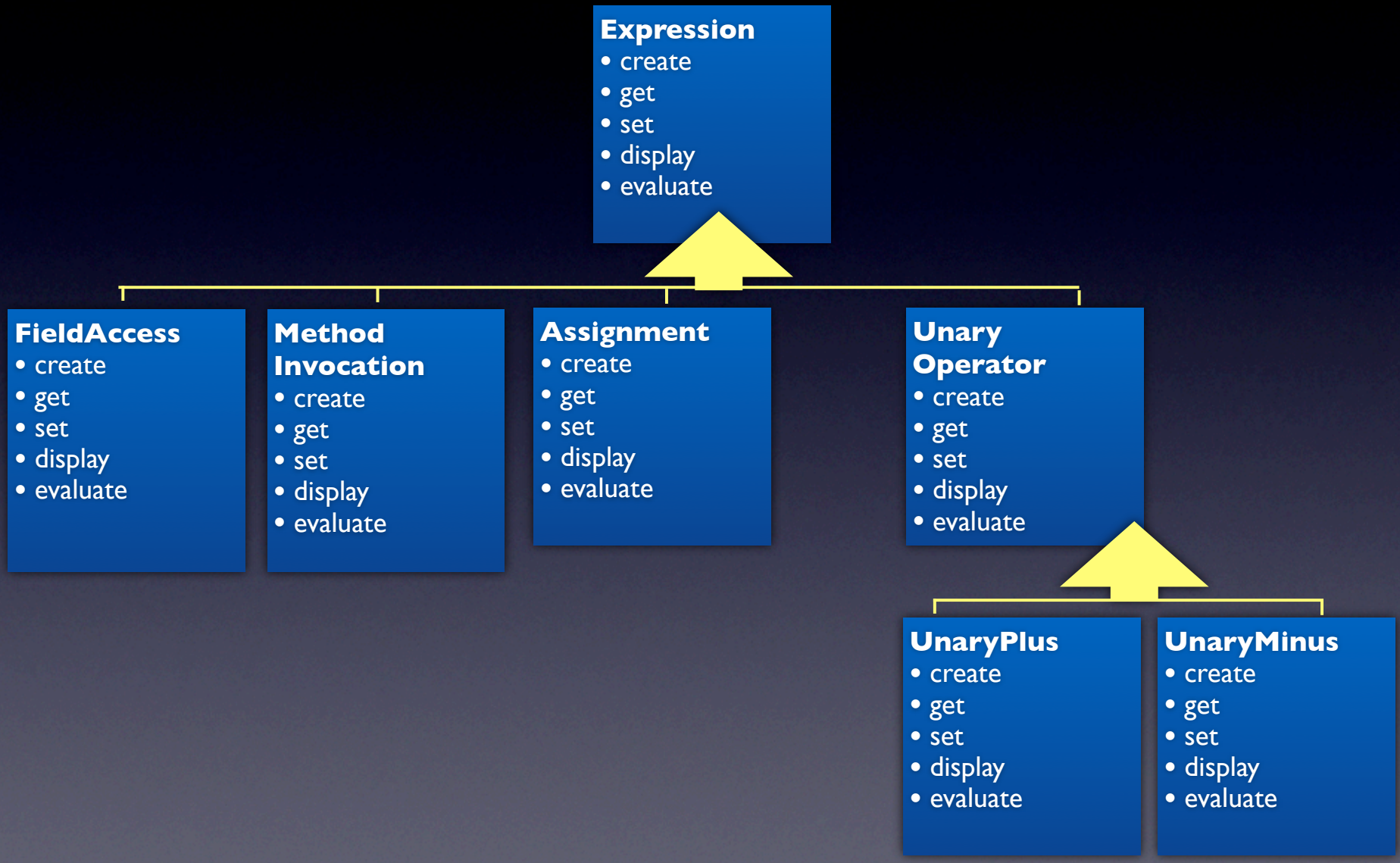
Example: Operations on Abstract Syntax Tree

Requirements:
The SEE supports the specification of expression programs. It contains a set of tools that share a common representation of expressions.
The initial toolset should include:

(1) an evaluation capability, which determines the result of evaluating an expression;

(2) a display capability, which depicts an expression textually; and

SEE in UML


Expression
• create
• get
• set
• display
• evaluate
FieldAccess
• create
• get
• set
• display
• evaluate
Method Invocation
• create
• get
• set
• display
• evaluate
Assignment
• create
• get
• set
• display
• evaluate
Unary Operator
• create
• get
• set
• display
• evaluate
UnaryPlus
• create
• get
• set
• display
• evaluate
UnaryMinus
• create
• get
• set
• display
• evaluate

How would you write this in Java?

-

How would you write this in Java?

Datatype

```
class ASTnode {
  Operation
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
```

```
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}

class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
class Assignment extends Expression{
  int evaluate(Env e) {
  …}
}

…
```

•

How would you write this in ML?

How would you write this in ML?

Datatype

```
type ASTnode =
FieldAccess| Expression | MethodInvocation| Assignment..
```

Operation

```
let rec evaluate env n =
   match n with
   FieldAcess -> ….
  | Expression -> ….
  | MethodInvocation -> …
  | Assignment ->…
```

Operation

```
let rec display n =
   match n with
   FieldAccess -> ….
  | Expression -> ….
  | MethodInvocation -> …
  | Assignment ->…
```

Evolving Requirements

(1) Add a new type of expression, ConditionalExpr.
(2) Expression should be optionally persistent;
(3) Style checking should be supported as well as syntax and semantic checking. It should be possible to check expression against multiple styles. Any meaningful combination of checks (e.g. syntax only; syntax plus style) should be permitted.

Adding a new expression type, ConditionalExpression in Java?

- Datatype
  class ASTnode {
    Operation
    int evaluate(Env e){
    …}
    void set(ASTnode n) {
    …}
    ASTnode get() {
    …}
    String display() {
    }
  }

```
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}

class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
class Assignment extends Expression{
  int evaluate(Env e) {
  …}
}

…
```

Adding a new expression type, ConditionalExpression in Java?

Datatype

```
class ASTnode {
  Operation
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
```

```
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}

class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
class Assignment extends Expression{
  int evaluate(Env e) {
  …}
}
 class ConditionalExpr extends Expression{
  int evaluate (Env e) {
}

…
```

Adding Typecheck function in Java?

Datatype

```
class ASTnode {
  Operation
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
```

```
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}

class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
class Assignment extends Expression{
  int evaluate(Env e) {
  …}
}

…
```

Adding Typecheck function in Java?

Datatype

```
class ASTnode {
  Operation
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
  boolean typecheck(Context c) {
  …}

}
```

```
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
  boolean typecheck(Context c) {
  …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
  boolean typecheck(Context c) {
  …}
}
class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
  boolean typecheck(Context c) {
  …}
}
class Assignment extends Expression{
  int evaluate(Env e) {
  …}
  boolean typecheck(Context c) {
  …}
```

Adding Persistence feature in Java?

Datatype

```
class ASTnode {
  Operation
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
```

```
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}

class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
class Assignment extends Expression{
  int evaluate(Env e) {
  …}
}

…
```

Adding Persistence feature in Java?

```
Datatype
class ASTnode {
  Operation
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
    if (notPersisted) {
       f=openFile();
       f.write(n.serialize());
       closeFile(f);
    }
    ASTnode get() {
     // getLocalCopy
     // if persisted, deserialize from a file
  …}
  String display() {
  }
  String serialize() {
  }
  void deserialize(String) {
  }
…
```

```
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
  String serialize() {
  …}
  void  deserialize(String s) {
  }
}
class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
  String serialize() {
  …}
  void deserialize(String s) {
  }
}
…
```

Adding Typecheck function in ML?

- **Datatype**

```
type ASTnode =
FieldAccess| Expression | MethodInvocation| Assignment..
```

Operation

```
let rec evaluate env n =
   match n with
   FieldAcess -> ….
   | Expression -> ….
   | MethodInvocation -> …
   | Assignment ->…
```

Operation

```
let rec display n =
   match n with
   FieldAccess -> ….
   | Expression -> ….
   | MethodInvocation -> …
   | Assignment ->…
```

Adding Typecheck function in ML?

- **Datatype**

```
type ASTnode =
FieldAccess| Expression | MethodInvocation| Assignment..
```

Operation

```
let rec evaluate env n =
   match n with
   FieldAcess -> ….
  | Expression -> ….
  | MethodInvocation -> …
  | Assignment ->…
```

Operation

```
let rec display n =
   match n with
   FieldAccess -> ….
  | Expression -> ….
  | MethodInvocation -> …
  | Assignment ->…
```

```
let rec typecheck context n =
   match n with
   FieldAccess -> ….
  | Expression -> ….
  | MethodInvocation -> …
  | Assignment ->…
```

Adding a new expression type in ML?

Datatype

```
type ASTnode =
FieldAccess| Expression | MethodInvocation| Assignment | ConditionalExpr..
```

Operation

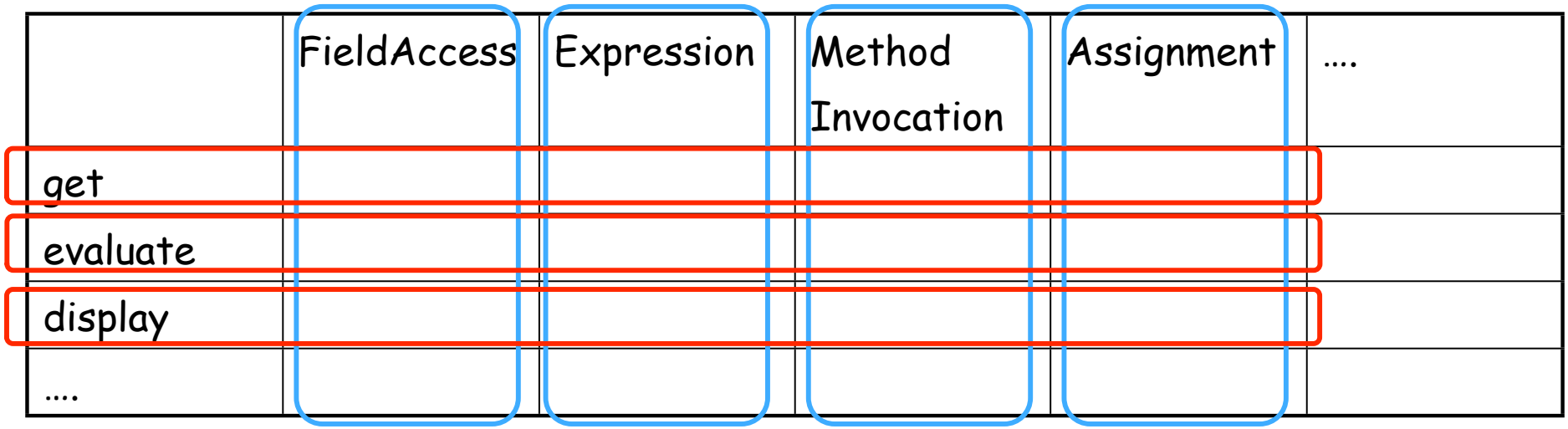
```
let rec evaluate env n =
   match n with
   FieldAcess -> ….
   | Expression -> ….
   | MethodInvocation -> …
   | Assignment ->…
   | ConditionalExpr ->
```

Operation

```
let rec display n =
   match n with
   FieldAccess -> ….
   | Expression -> ….
   | MethodInvocation -> …
   | Assignment ->…
   | ConditionalExpr ->
```

```
let rec typecheck context n =
   match n with
   FieldAccess -> ….
   | Expression -> ….
   | MethodInvocation -> …
   | Assignment ->…
   | ConditionalExpr ->
```

Functional vs. Data Concerns



Visitor Design Pattern

- It allows OO programs to localize functional concerns using double dispatching.

Visitor Pattern

Step 1. Add Visitor Class

```
class ASTnode {
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}
class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
```

```
class ASTNodeVisitor {
  void visitExpression(Expression e) {}
  void visitFieldAccess(FieldAccess f) {}
  void visitMethodInvocation(MethodInvocation m) {}
  void visitAssignment(Assignment a) {}
```

Visitor Pattern

Step 2. Extend the Visitor

```
class ASTnode {
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}
class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
```

```
abstract class ASTNodeVisitor {
  void visitExpression(Expression e) {}
  void visitFieldAccess(FieldAccess f) {}
  void visitMethodInvocation(MethodInvocation m) {}
  void visitAssignment(Assignment a) {}

class TypeCheckVisitor extends ASTNodeVisitor{
  void visitExpression(Expression e) {
  // type checking for Expression }
  void visitFieldAccess(FieldAccess f) {
  // type checking for FieldAccess}
…
```

Visitor Pattern

Step 3. Weave the Visitor

```
class ASTnode {
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
  void accept(Visitor v) { }
}
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
   void accept (Visitor v) {
   v.visitExpression(this); }
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
   void accept (Visitor v) {
   v.visitFieldAccess(this); }
```

```
class ASTNodeVisitor {
  void visitExpression(Expression e) {}
  void visitFieldAccess(FieldAccess f) {}
  void visitMethodInvocation(MethodInvocation m)
  void visitAssignment(Assignment a) {}

class TypeCheckVisitor extends ASTNodeVisitor{
  void visitExpression(Expression e) {
  // type checking for Expression }
  void visitFieldAccess(FieldAccess f) {
  // type checking for FieldAccess}
...

TypeCheckVisitor checker= new
TypeCheckVisitor();
AST ast = getASTRoot();
// control  logic for recursively traversing AST
nodes{
        ASTNode node = …
        node.accept(checker);
}
```

QUIZ

- Identify weakness of this implementation from a program understanding perspective.
- Extend this implementation of the AST example to accommodate two evolution scenarios:
  - Add a new data type, ArrayCreation expression.
  - Add a new operation, prettyPrint.
- Essay question: Discuss strengths and weaknesses of this Visitor Pattern implementation with respect to changeability.

Today's Agenda

- Presentations: Adam and Xin
- Problem Space
  - N dimensional separation of concerns by Peri Tarr et al.
  - Canonical example of multiple dimensions of concerns: e.g. abstract syntax tree example
  - Writing the AST example in a functional programming language
  - Writing the AST example code in an object-oriented language
  - QUIZ--Listen closely this lecture as you should be able to answer all questions based on this lecture.

This week's Agenda

- Solution Space
  - Collaboration-based design / Role-based design
  - Language-based approach (language tweaking): Mixins, Using C++ templates to support flexible feature composition [VanHilst and Notkin 1996], etc.
  - Language extension to support crosscutting concerns: AspectJ [Kiczales et al. 1997]
  - Tool-based approaches: Concern graphs [Robillard and Murphy 2003], AspectBrowser [Griswold et al. 01], CME [Tarr et al.], etc.

Example: Logging concern

- Where do you have to change to add the logging concern?
- How can you modularize logging concerns?
  - Log4J?

Other crosscutting Concerns

- Runtime checking of invariants
- Tracing executions
- Serializing
- Database transaction
- Security
- Performance enhancement, etc.

Solution Space

- OO Design technique and methodology
  - Role-based modeling
- Programming language tweaking
  - Mixins
- Programming language approach
  - AspectJ
- Software engineering tool approach
  - FEAT, AspectBrowser, CME, etc.

Recap of OO Design

- Language constructs
  - methods, inheritance, packages, types (classes and interfaces), access modifiers, etc.
- Good at supporting for ADT
  - separate a particular data representation choice from other parts of a program in a source file
  - hide the representation choice behind an interface

Role-based Model [Anderson et al. 92]

- OO design technique to achieve separation of concerns
  - Also called as "responsibility-driven" design and "collaboration-based" design.
  - Behavioral requirement is implemented by a set of communicating objects.
  - For each behavior requirement, separate the role of each object from irrelevant details.

Role-based Model

- What is a role?
  - A particular responsibility of an object
- What is a role model?
  - The unit of collaboration
  - The concept of communicating objects (roles)

Role-based Model

Design Methods:

- Identify collaboration among objects
- Assign a role to each object in the collaboration that you model
- Synthesize roles in several role models

	Object OA	Object OB	Object OC
Collaboration c1	Role A1	Role B1	Role C1
Collaboration c2	Role A2	Role B2	
Collaboration c3		Role B3	Role C3
Collaboration c4	Role A4	Role B4	Role C4

Mixin [Bracha, Cook 90]

- Template<T> class C inherits T {…}

- Implementation technique for role models
  - A mixin is an abstract subclass whose superclass is not determined.

Recap of Java Style Inheritance

- Support reuse of the implementation provided by a superclass.
- A subclass has a control.

Problem 1.
Difficulty of Adding Roles

```
client C {
A a = new A();
a.m1();
a.m2();
}
```

```
class A {
    method m1()  {
            …
    }
    method m2() {
       m1();
       …
    }
}
```

- Change Scenario:
  - Add an additional role in A

Problem 1.
Difficulty of Adding Roles

```
client C {
A a = new A();
a.m1();
a.m2();
}
```

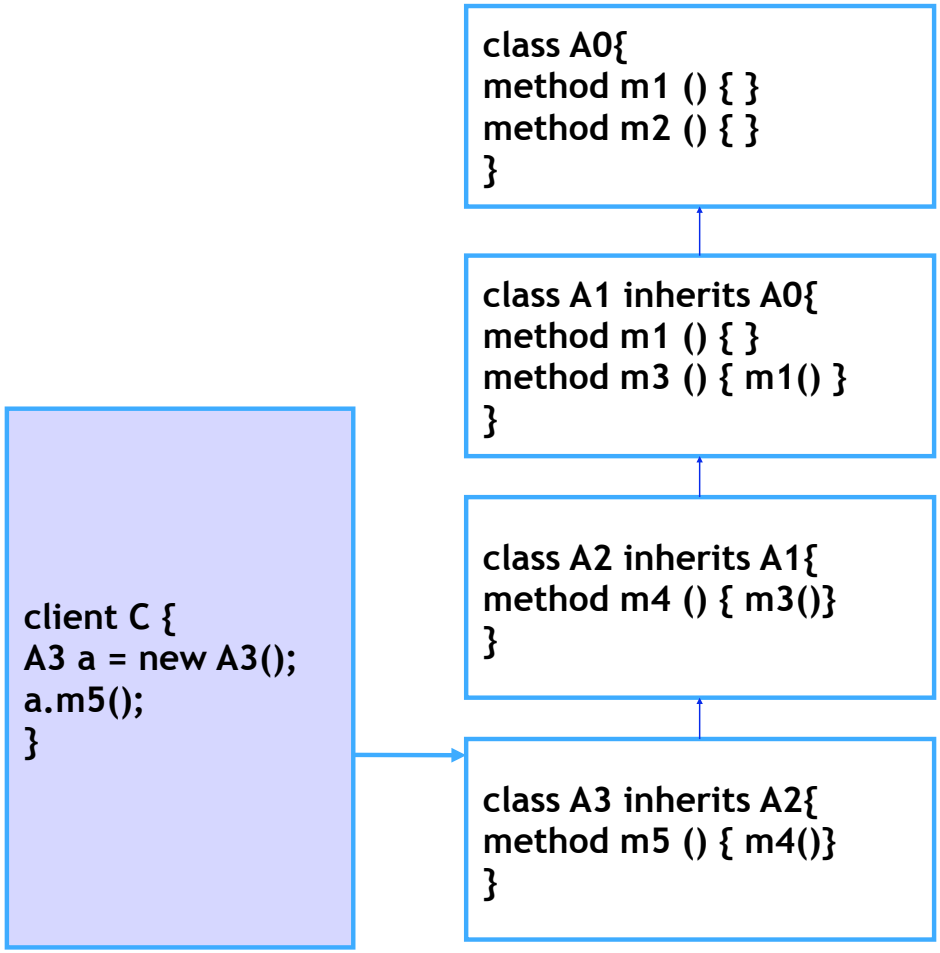
```
client C {
A a = new A1();
a.m1();
a.m2();
}
```

```
class A {
    method m1()  {
            …
    }
    method m2() {
       m1();
       …
    }
}
```

```
class A1 inherits A {
   method m1() {
      … // override m1.
   }
   method m3() {
      … // extra role
   }
}
```

- Any problems?

Problem 2.
Fragile Class Hierarchy

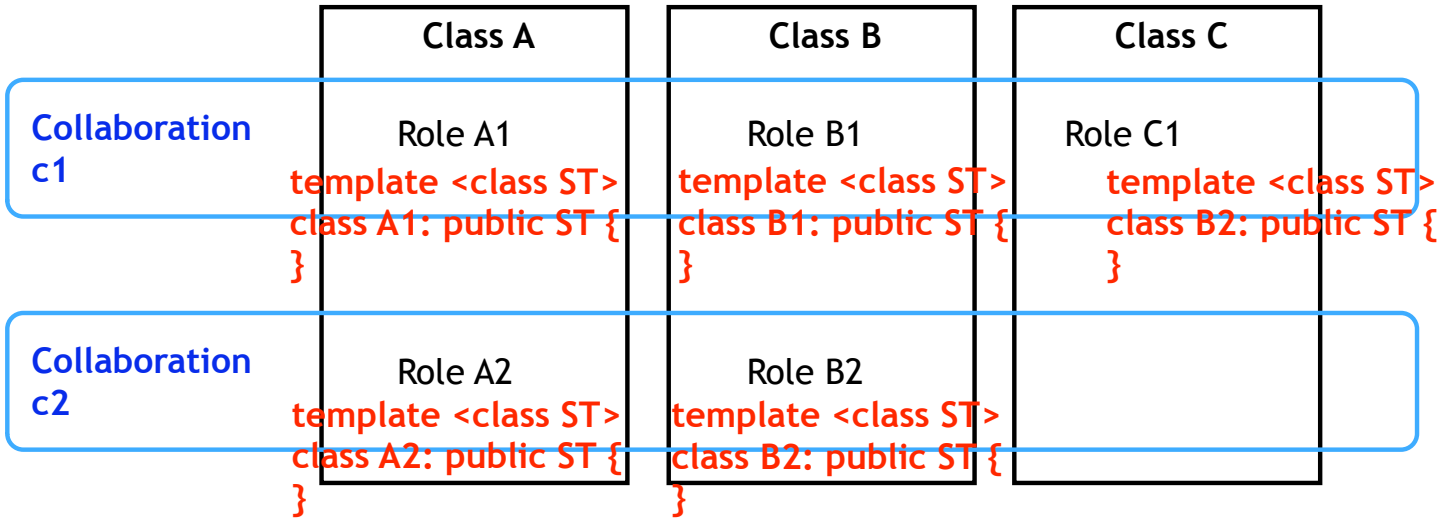


- Change Scenario:
  - Change the behavior of m3().
- Any problems?
  -

Mixin for Role-based Model [VanHilst, Notkin 96]

- Implementation technique for role models
  - A mixin is an abstract subclass whose superclass is not determined.
  - A role as a class, including all the relevant variables and methods
  - Roles are composed by inheritance
  - To make roles reusable, the superclass of a role is specified in a template argument of C++.

Mixin using C++ template



Composition Statement

```
class a1: public A1<empty> {};
class A:  public A2<a1> {};
class b1: public B1<emtpy> {};
class B:  public B2<b1> {};
class C:  public C1<empty> {};
```

Role based model via inheritance, static binding, and type parameterization

Example

```
template <class SuperType>
class Shifter: public SuperType {
  public :
     void shiftLine (int l) {
         int num_words=words(l);
         for (int w=0; w<num_words; w++)
            addShift(l,w,num_words);
     }
    void initializeShift() {
         int num_lines = lines ();
         resetShift();
         for (int l=0; l<num_lines; l++)
            shiftLine(l);
    }
};
```

Evaluation of Mixin Approach

+ Roles can be added to a single base class incrementally.
+ Fine grained decomposition/ flexible composition
+ No run time overhead
- There is NO direct support for adding a set of roles to multiple base classes together.
- Composition orders matter. Classes composed later can only use classes composed earlier.
- Relying on C++ type safety – not a good idea
- Reduced understandability

AspectJ [Kiczales et al. 01]

- Extension of Java that supports crosscutting concerns
- An aspect is a module that encapsulates a crosscutting concern.
  - Joint point: the moment of method calls and field references, etc.
  - Point cut: a mean of referring to a set of joint points
  - Advice: a method-like construct used to define additional behavior at join points

Join point and Pointcut

- Name based

```
pointcut move ():
    call (void FigureElment.moveBy(int,int)) ||
    call (void Point.setX(int) ||
    call (void Point.setY(int) ||
    call (void Line.setP1(Point) ||
    call (void Line.setP2(Point) );
```

- Pattern based

```
pointcut move () :
    call (void Figure.make*.(...))
    // starting with "make," and which take any number of
    parameters
```

Advice

- after: the moment the method of a joint point has run and before the control is returned
- before: the moment a join point is reached
- around: the moment a join point is reached and has explicit control over whether the method itself is allowed to run at all

How to Retrieve Execution Context

- pointcut parameters
  - advice declaration values can be passed from the pointcut designator to the advice.

```
before (Point p, int val) : call (void p.setX(val)) {
        System.out.println("x value of"+p+ "will be set to" + val+".";
}
pointcut gets(Object caller) : instanceof (caller) && (call(int Point.getX()) );
```

- access to return value

```
after (Point p) returning (int x) : call(int p.getX()) {
        System.out.println(p+ "returned" + x + "from getX().";  }
```

- thisJointPoint

Aspect Code: Tracing

```
aspect SimpleTracing {
   pointcut traced():
      call (void Display.update()) ||
      call (void Display.repaint());
   before ()  : traced() {
      println("Entering:" + thisJointPoint);
   }
   after ()  : traced() {
      println("Exiting:" + thisJointPoint);
   }

   void println(String str) {
   ...// write to the appropriate stream
  }
}
```

Aspect Code: Runtime Invariant Checking

```
aspect PointBoundsInvariantChecking {
  before (Point p, int x) : call (void p.setX(x)) {
     checkX(p,x);
  }
  before (Point p, int y) : call (void p.setY(y)) {
     checkY(p,x);
  }
  before (Point p, int x, int y) : call (void p.moveBy(x,y)) {
     checkX(p,p.getX()+x);
     checkY(p,p.getY()+y);
  }
  void checkX(Point p, int x) {…//check an invariant}
  void checkY(Point p, int y) {…//check an invariant}
}
```

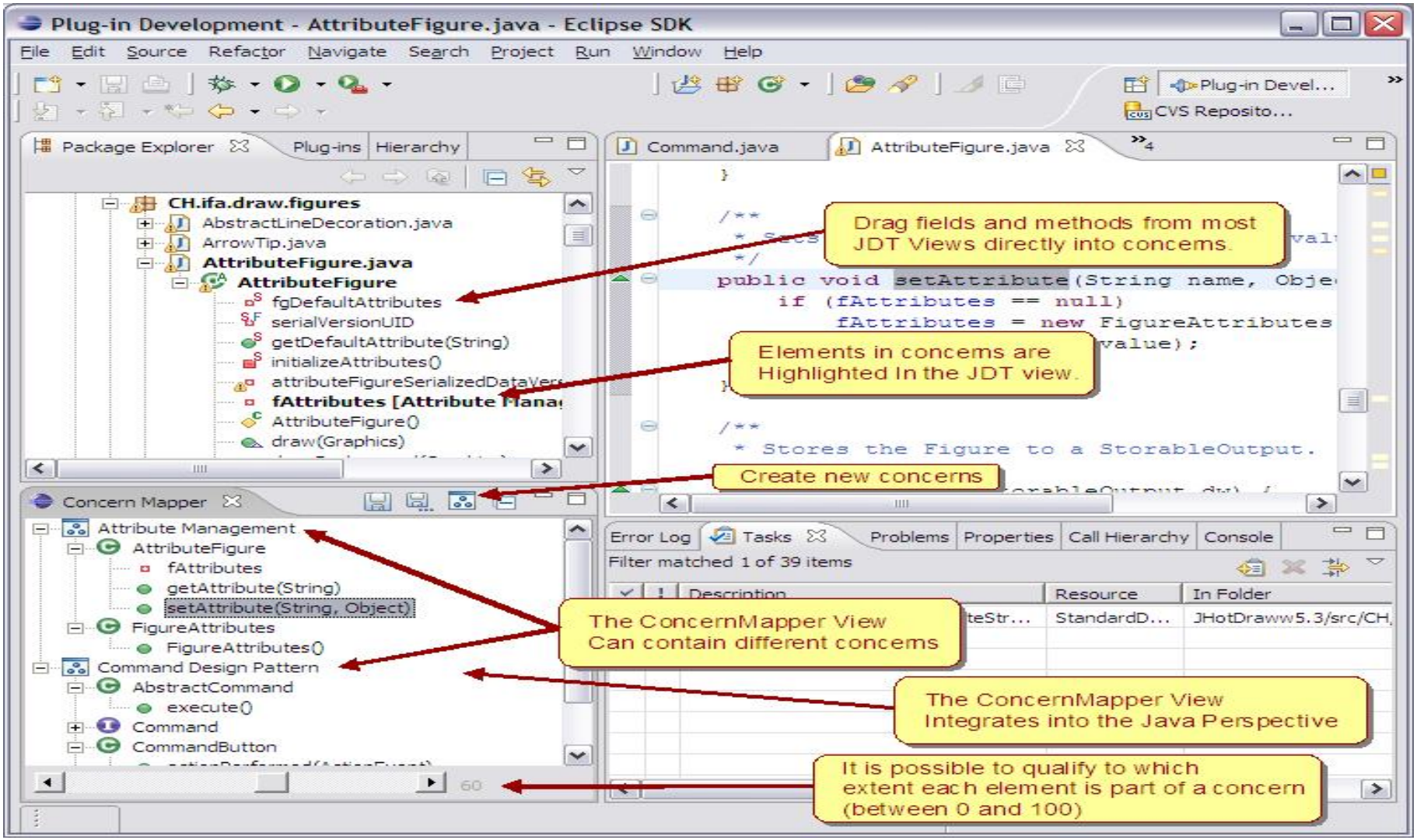
Evaluation of AspectJ

+ Dynamic crosscutting mechanism helps aspect code to be invoked implicitly
+ Reduce code duplication
- AspectJ style differentiates the base code from aspect code.
- Unidirectional reference from AspectJ code to base code
- AspectJ code may end up reflecting the base class hierarchy.
- Base code sometimes needs to be restructured to expose suitable join points.

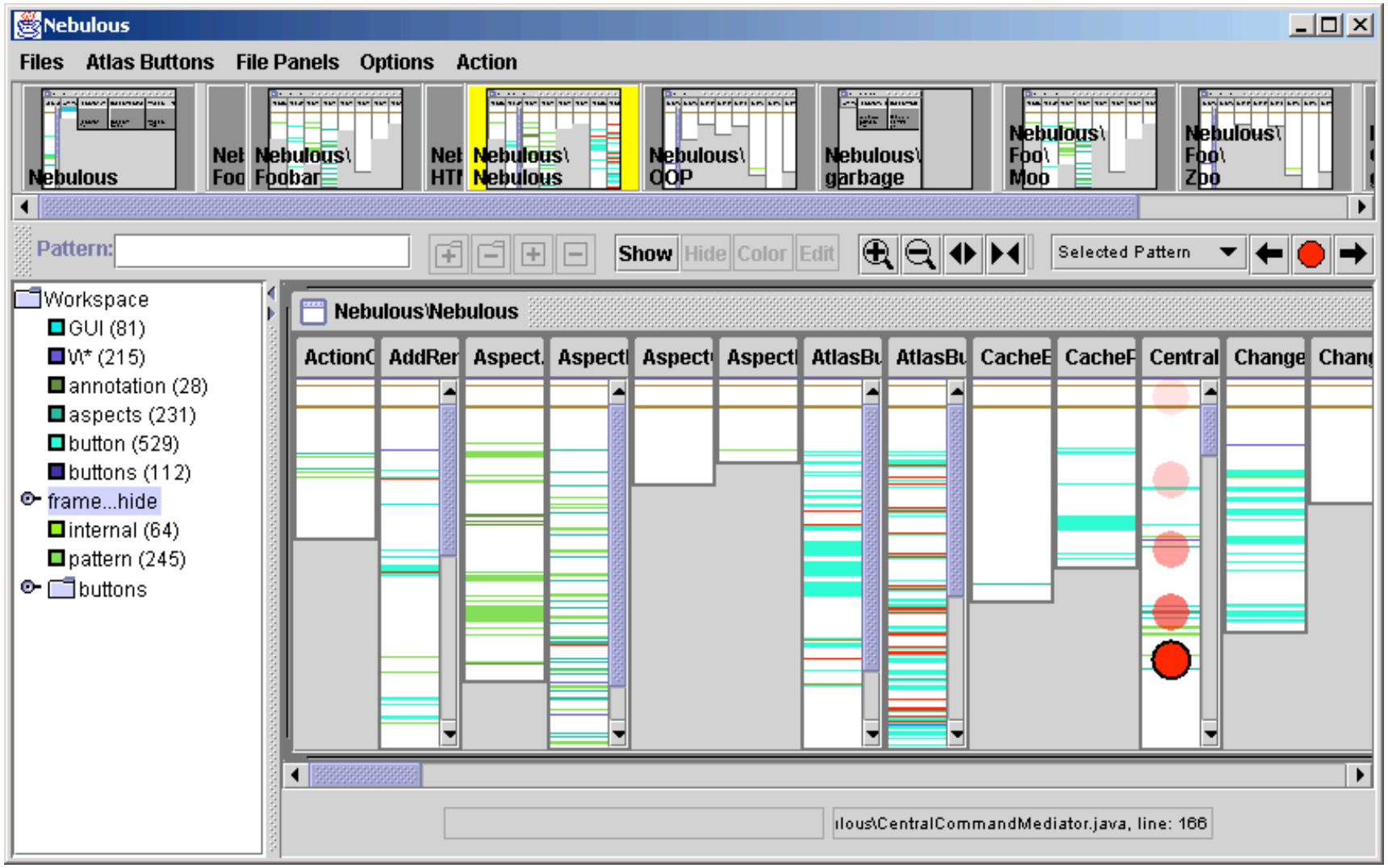
Lightweight Tool Support

- Finding aspects and managing crosscutting concerns
  - FEAT (Concern Graph) [Robillard et al.03]
- Lexical search tools
  - grep, STAR tool
  - Aspect Browser [Griswold et al.01]

FEAT [Robillard et al. 03]



Aspect Browser [Griswold et al. 01]



Other Lightweight Tools

- Navigation and Management
  - CME: Crosscutting Concern Modeling Environment [IBM]
  - JQuery [De Volder 03]
- Crosscutting Concern Mining Tool
  - Based on topology of structural dependencies [Robillard 05]
  - Based on code clones [Shepherd et al. 05]
  - Based on event traces [Breu et al. 04]

Recap of Today's Lecture

- Mixin
  - + good at adding functional concerns that cross-cut the boundary between classes
  - - complex PL tweaking -> difficulty in program understanding
- AspectJ
  - + good at adding functional concerns
  - + good at intercepting control flow
  - - difficulty in program understanding
- Lightweight tool approaches
  - + can be easily integrated into development practices
  - - only good at discovering code with particular symptoms
  - - human in the loop

If you are interested in more,

- Good news! a lot more interesting research out there
  - design patterns
  - open implementation, meta object protocol, composition filters, hyperslices, etc
  - programming languages
  - many light-weight tools
  - many design methodologies
  - validation of existing approaches and tools
- Open problems, open solution space