Crosscutting Concerns

CSE 503 – Software Engineering

Miryung Kim

16 May 2006

Recap of “Information Hiding” Principle

- What is the “Information Hiding Principle?”
  - hide design decisions that are likely to change.
  - separating interface from implementation.
  - hierarchical design decisions.

Recap of “Information Hiding” Principle

- What is the “Information Hiding Principle?”
  - using C++ instead of C?
  - using private instead of public?
  - abstract the behavior and data?
  - reduce dependencies between modules?

Parnas[72]
Hide design decisions that are likely to change.

≈ Identify design decisions that are unlikely to change and fixate them.

Any Problems with IHP?

- you don't know what to hide
- increase complexity by adding more layers.
- performance cost
- how can we anticipate what are likely to change.

Parnas[72]
Hide design decisions that are likely to change.

≈ Identify design decisions that are unlikely to change and fixate them.

Any Problems with IHP?

- 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?
  - What creates data and who's reading the data.
  - Scope design decisions.
  - Layered architecture.

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?
  - architectural design decisions (e.g. pipeline architecture, layered architecture)
  - class hierarchy in OO programs

Primary vs. Secondary Design Decisions?

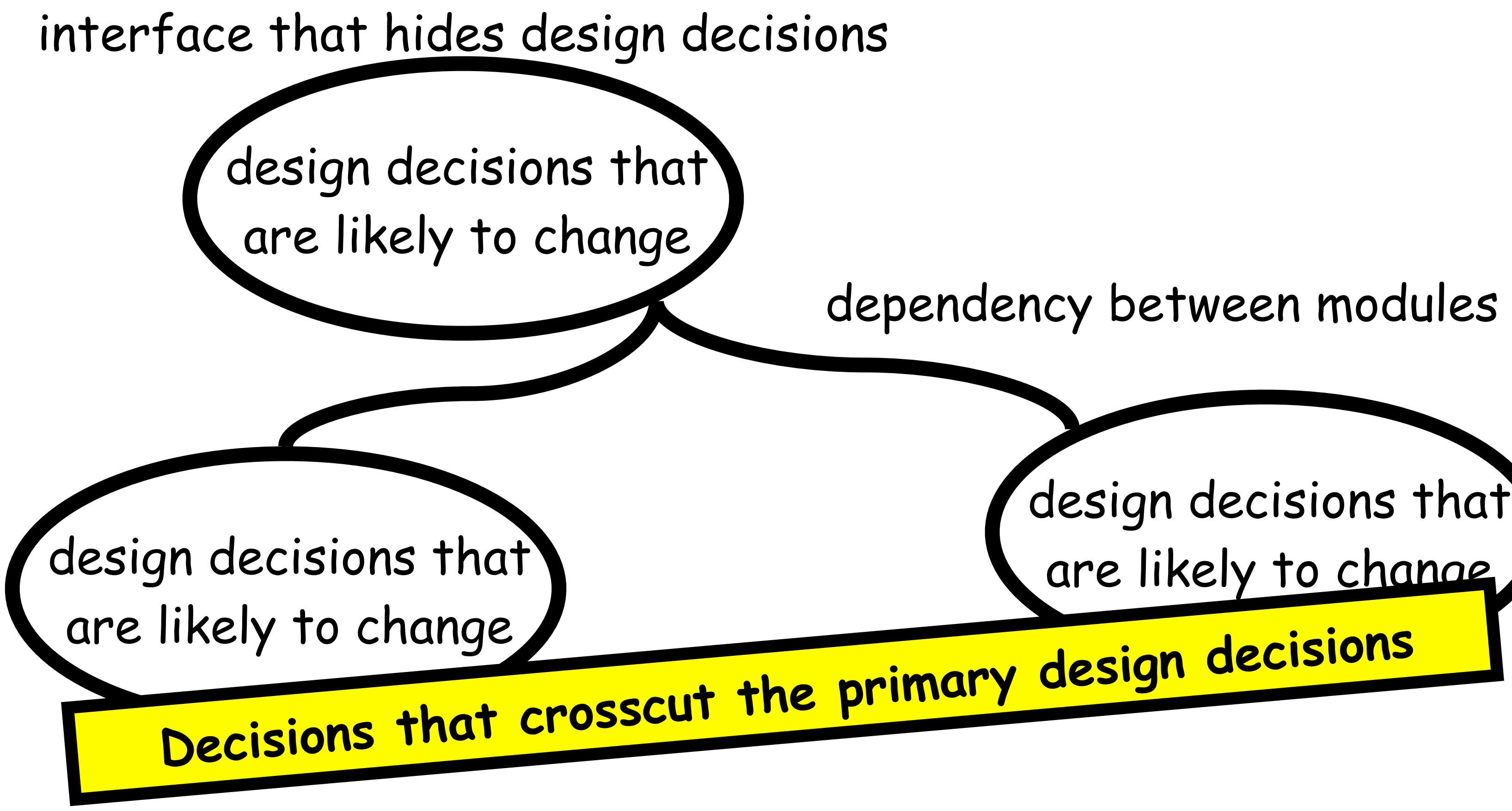
- Secondary design decisions
  - Less important than primary decisions
  - Decisions that architects did not anticipate in the beginning of system design.
- Examples?
  - dependency that are added later given a layered architecture.
  - performance (indirection)
  - data format, security,

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?
  - insertion of additional features or operations
  - system performance improvement
  - logging or tracing system execution

Primary Design Decisions
+ Secondary Design Decisions


interface that hides design decisions
design decisions that are likely to change
dependency between modules
design decisions that are likely to change
design decisions that are likely to change
Decisions that crosscut the primary design decisions

Crosscutting Concerns

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

Functional vs. Data Concerns

- Example: Operations on Abstract Syntax Tree

	Statement	Expression	Method Invocation	Assignment	....
Typecheck					
Evaluate					
....					
....					

How would you write this in ML?

Datatype

```
type ASTnode =
Statement| Expression | FunctionCall| Assignment..
```

Operation

```
let rec typecheck ctxt n =
   match n with
   Statement -> ....
  | Expression -> ....
  | MethodInvocation -> ...
  | Assignment ->...
```

Operation

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

- Any problems with changeability?
  - you cannot change AST.
  - it's difficult to add more datatype.
  - it's easy to add more operations.

How would you write this in Java?

```
Datatype
class ASTnode {
  Operation
  boolean typecheck(Context c){
  ...}
  int evaluate(Context c){
  ...}
  void setParent(ASTnode n) {
  ...}
  ASTnode getParent() {
  ...}
}
```

```
class Expression extends ASTnode {
   boolean typecheck(Context c) {
   ...}
   int evaluate(Context c) {
   ...}
}
class FunctionCall extends ASTnode{
  boolean typecheck(Context c) {
  ...}
  int evaluate(Context c) {
  ...}
}
...
```

- Any problems with changeability?
  - inverse of the other one.
  - difficult to add operations easy to add data type

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.

Crosscutting Concerns

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

Solution Space

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

Recap of OO Design

- Language constructs
  - methods, inheritances, 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.

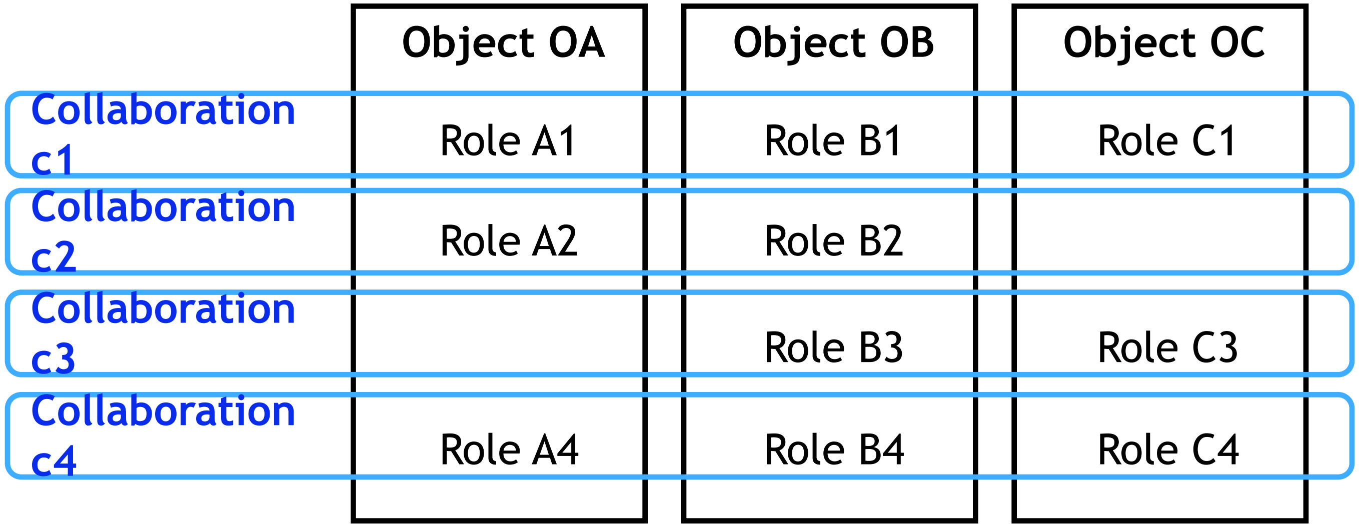
Role-based Model

- 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



Solution Space

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

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*
  - Do some extra operations on the existing role m1.

Problem 1. Difficulty of Adding Roles

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

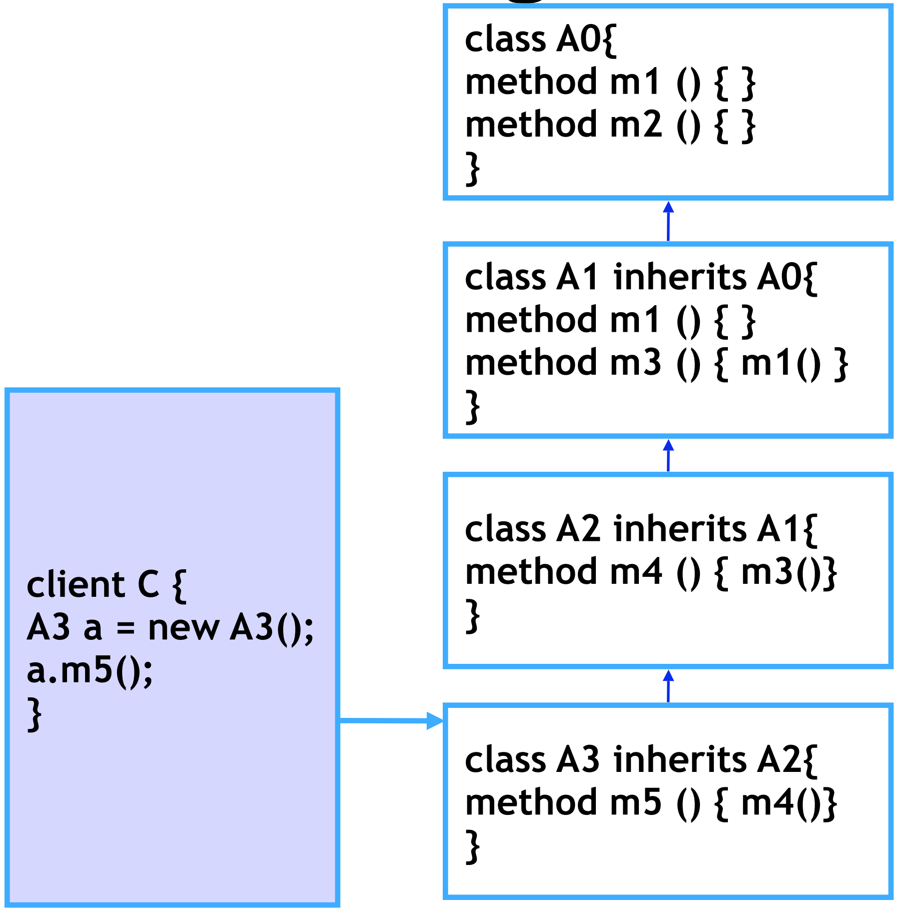
```
client C {
A a = new A1();
a.m1();
a.m2();
}
```

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

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

- Any problems?
  - made changes to m2()
  - class hierarchy no longer reflect what design you wanted to have.

Problem 2. Fragile Class Hierarchy



```
class A0{
method m1 () { }
method m2 () { }
}
```

```
class A1 inherits A0{
method m1 () { }
method m3 () { m1() }
}
```

```
class A2 inherits A1{
method m4 () { m3()}
}
```

```
class A3 inherits A2{
method m5 () { m4()}
}
```

```
client C {
A3 a = new A3();
a.m5();
}
```

- Change Scenario:
  - Change the behavior of m3().
- Any problems?
  - m4, m5

Mixin [Bracha, Cook 90]

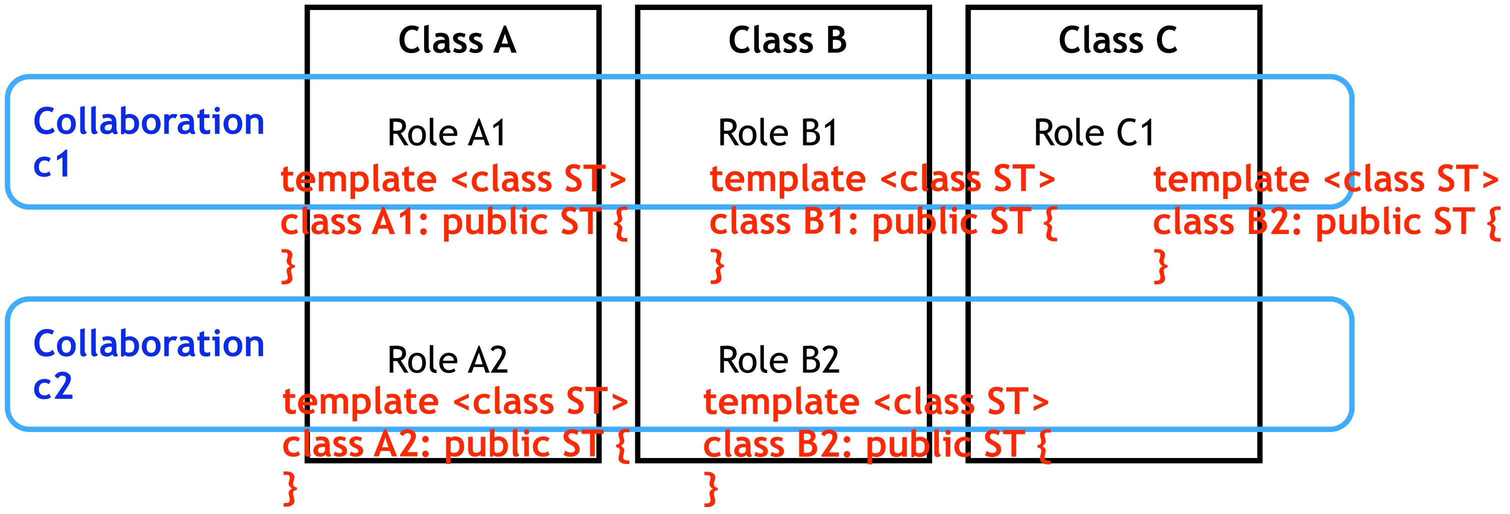
- Template<T> class C inherits T {…}

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

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

Evaluation of Mixin Approach

+ Roles can be added to a single base class incrementally.

Evaluation of Mixin Approach

+ Roles can be added to a single base class incrementally.
+ Fine grained decomposition/ flexible composition

Evaluation of Mixin Approach

+ Roles can be added to a single base class incrementally.
+ Fine grained decomposition/ flexible composition
+ No run time overhead

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.

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.

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

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

Solution Space

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

AspectJ [Kiczales et al.]

- 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 point
  - Advice: a method like constructs 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
    call (public * Display.*(…))
    // any call to a public method defined on Display
```

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

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
 }
}
```

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().”; }
```

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

Evaluation of AspectJ

+ Dynamic crosscutting mechanism helps aspect code to be invoked implicitly

Evaluation of AspectJ

+ Dynamic crosscutting mechanism helps aspect code to be invoked implicitly
+ Reduce code duplication

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.

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

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.

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.

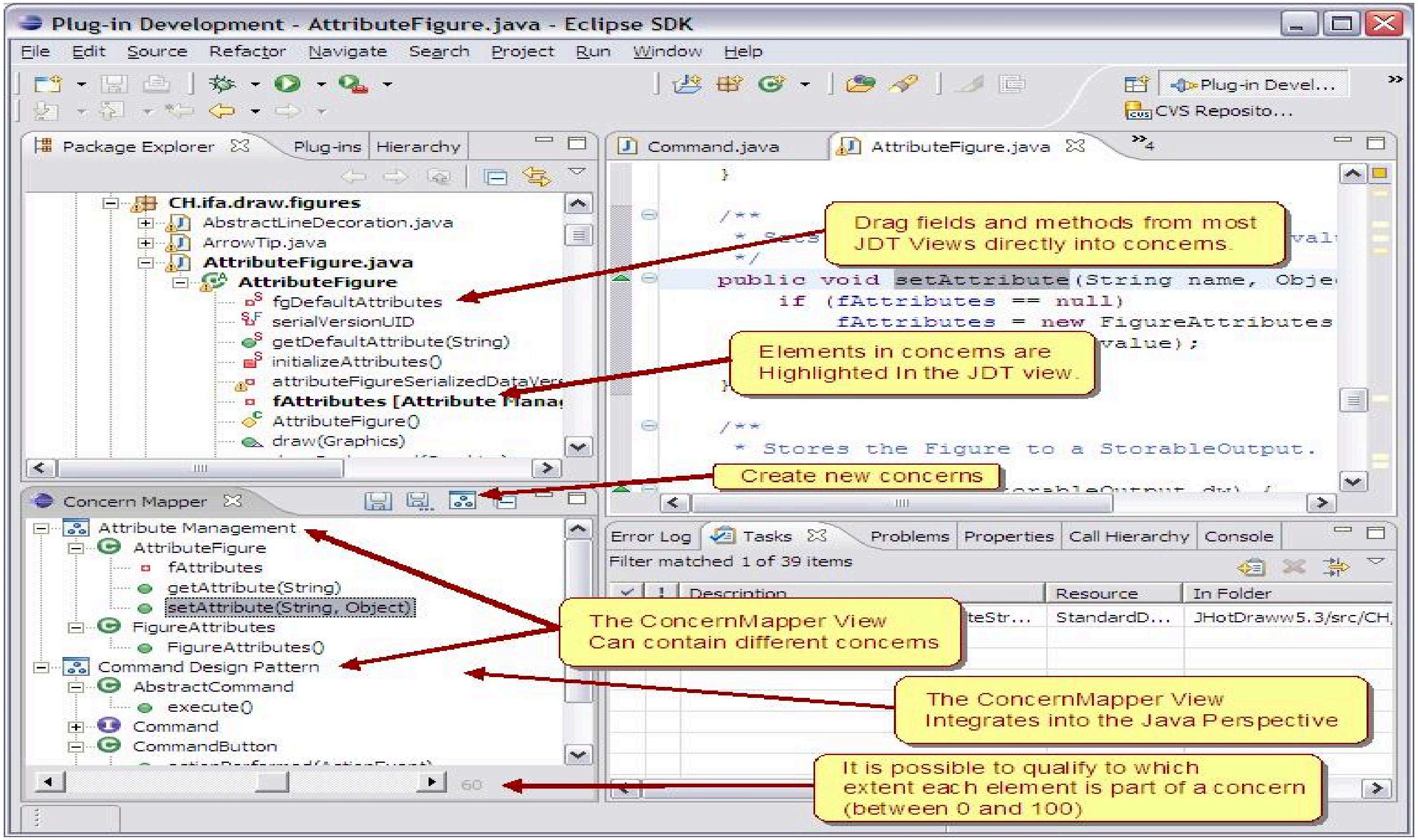
Solution Space

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

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]



```
}

/**
 * Sets
 */
public void setAttribute(String name, Obje
    if (fAttributes == null)
        fAttributes = new FigureAttributes
                              value);
}

/**
 * Stores the Figure to a StorableOutput.
```

Drag fields and methods from most JDT Views directly into concerns.

Elements in concerns are Highlighted In the JDT view.

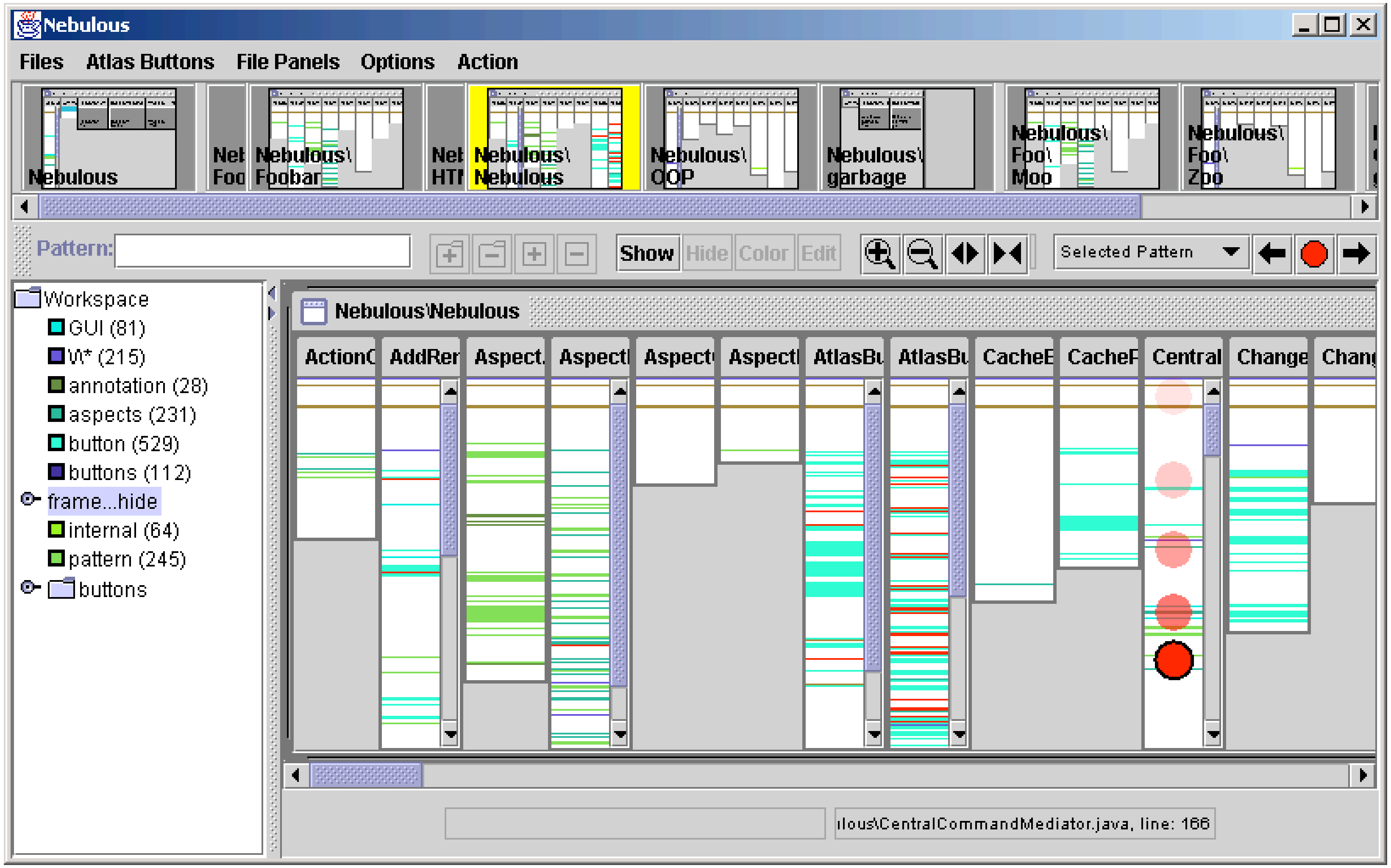
Create new concerns

The ConcernMapper View Can contain different concerns

The ConcernMapper View Integrates into the Java Perspective

It is possible to qualify to which extent each element is part of a concern (between 0 and 100)

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