Reflection Applied: Aspects

CPSC 501: Advanced Programming Techniques
Fall 2022

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Monday, November 14, 2022

UNIVERSITY OF CALGARY

Intercession via aspects

Introduction

- What is AspectJ?
  - Aspect oriented programming (AOP) extension to Java

- What is an Aspect?
  - a particular part or feature of something.

History

- Developed at Xerox PARC (Palo Alto RC)
- Launched in 1998
- PARC transferred AspectJ to an openly-developed eclipse.org project in December of 2002.

For more info: www.eclipse.org/aspectj

Introduction

- What are goals of AOP?
  1. **Separation of concerns**
  2. **Modularity**
     - No more tangled code
     - Simplicity
     - Maintainability
     - Reusability
  3. **Aspects**
     - encapsulate behaviors that affect multiple classes (OO) into reusable modules.

I’m concerned?

Cross-Cutting Concern

- What is a **cross-cutting concern**?
  - Behavior that cuts across the typical divisions of responsibility**, such as logging or debugging**
  - A problem which a program tries to solve.
  - Aspects of a program that **do not relate to the core concerns** directly, but which **proper program execution nevertheless requires**.

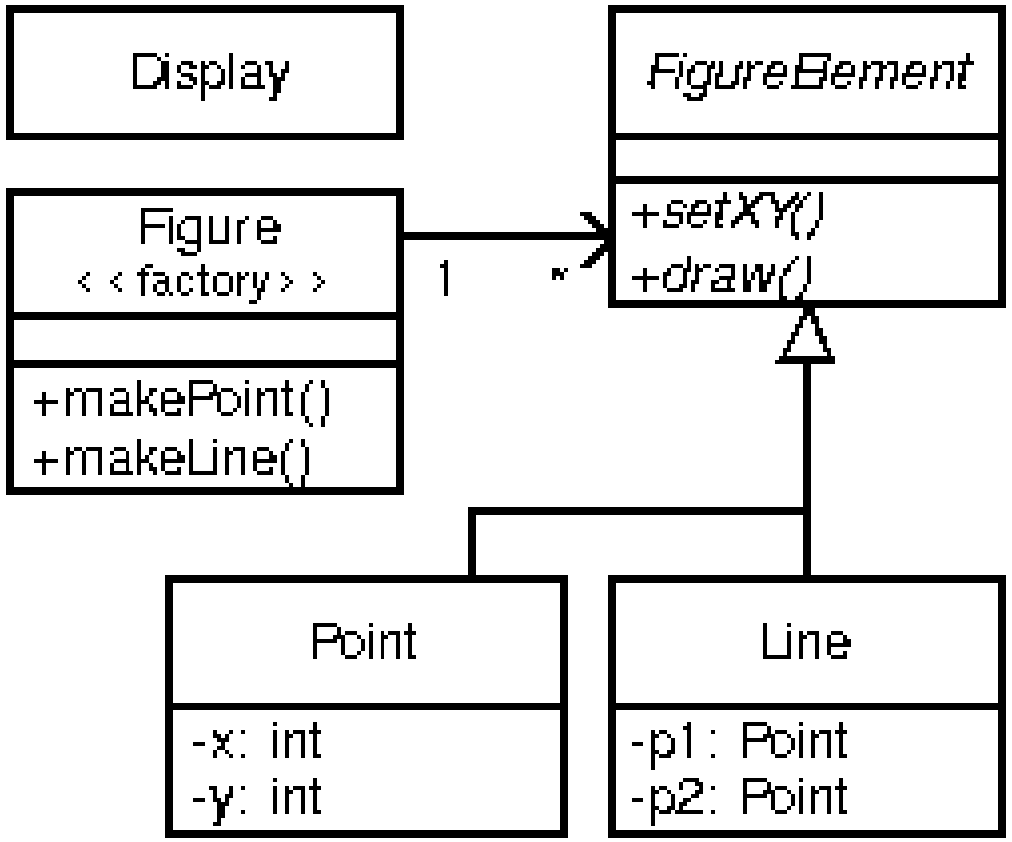
Language: Dynamic VS Static crosscutting

- **Dynamic crosscutting**
  - define **additional behavior** to run at certain well-defined **points** in the **execution** of the program

- **Static crosscutting**
  - **modify the static structure** of a program (e.g., adding new methods, implementing new interfaces, modifying the class hierarchy)

We’ll build around this

Reference Object Structured for Following



Join In

Language: Join Points

- **Join Points:** well-defined points in the execution of a program

  - Method call, Method execution
  - Constructor call, Constructor execution
  - Static initializer execution
  - Object pre-initialization, Object initialization
  - Field reference, Field set
  - Handler execution
  - Advice execution

Language: Join Points

Method-execution

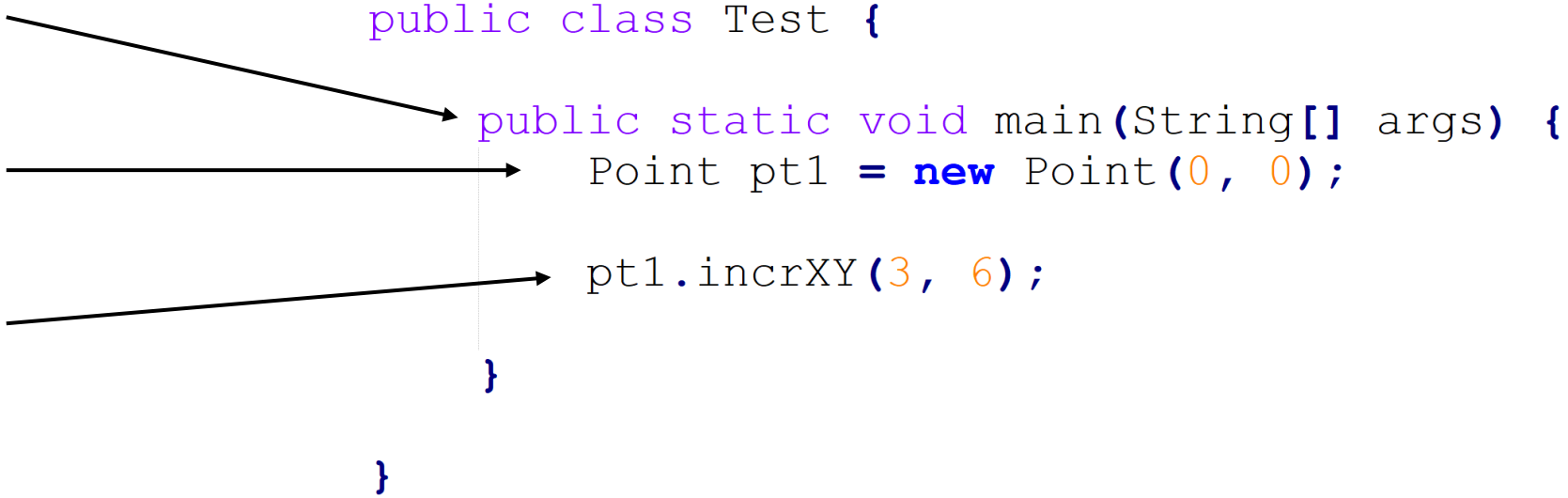
"Test.main(..)"

Constructor-call

"Point(..)"

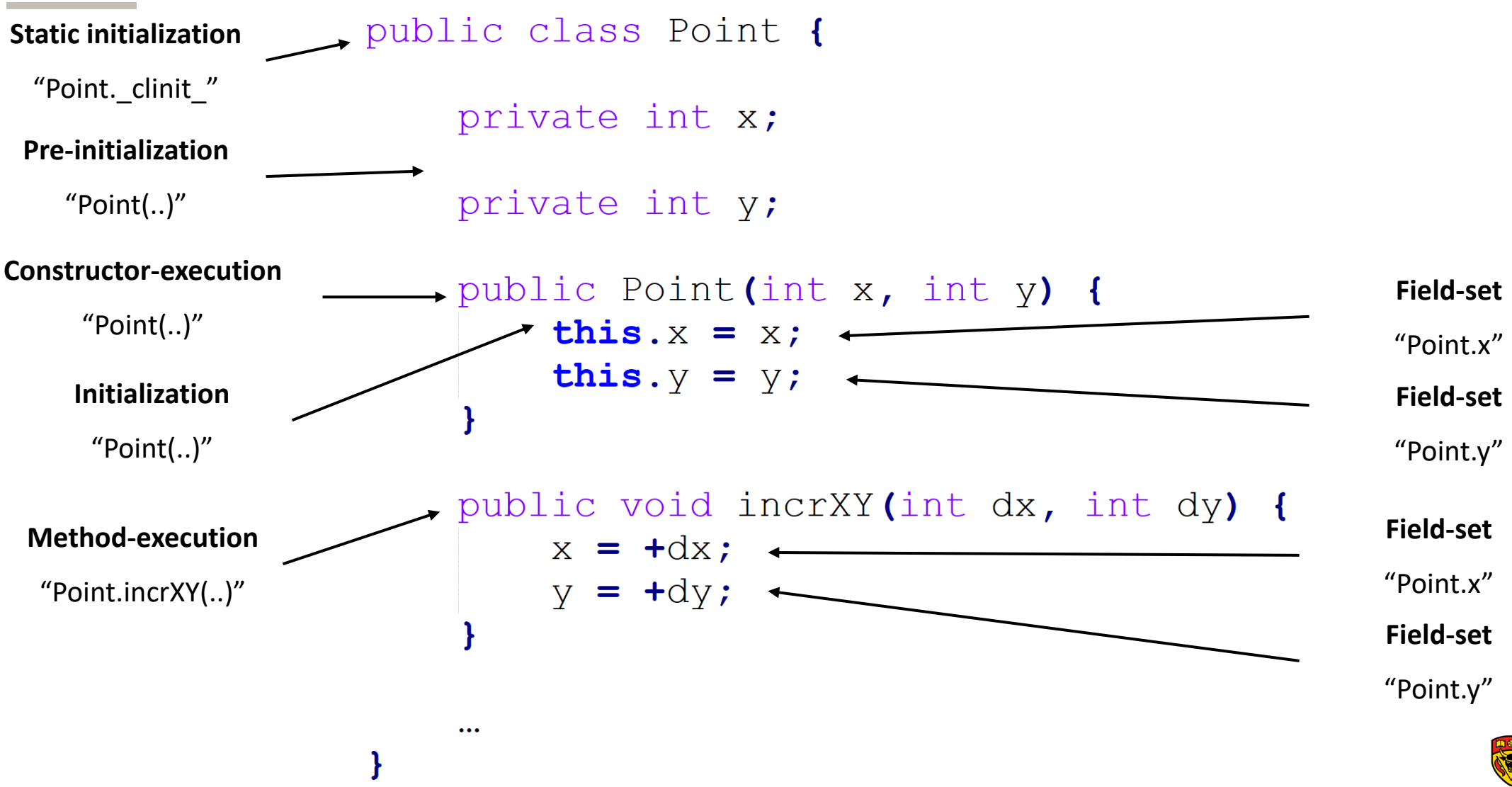
Method-call

"Point.incrXY(..)"



```
public class Test {

    public static void main(String[] args) {
        Point pt1 = new Point(0, 0);

        pt1.incrXY(3, 6);

    }

}
```

Language: Join Points



Cut in

Language: Pointcuts

- A set of join point, plus, optionally, some of the values in the execution context of those join points.
- Can be composed using boolean operators || , &&
- Matched at runtime

Language

Pointcut examples

Matches if the join point is a method call with this signature.

```
call(public void Point.setX(int))
```

Matches if the join point is a method call to any kind of FigureElement.

```
call(public void FigureElement.incrXY(int,int))
```

Matches any call to setX OR setY

```
call(public void Point.setX(int)) || call(public void Point.setY(int))
```

Language

Pointcut examples

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

- There is a cross-cutting concern here relating to moving
- We can capture these in our own user defined pointcut

Language: Wildcards and cflow

```
call(void Figure.make*(..))
```

A void method on Figure whose name begins with make regardless of parameters (both makePoint and makeLine)

```
call(public * Figure.* (..))
```

Each call to all Figure's public methods

```
cflow(move())
```

Identify all join points that occur between when move() is called and it returns (in dynamic flow of move())

When to cut in?

Language: Advice

- Method-like mechanism used to declare that certain code should execute at each of the join points in the pointcut.
- Advice:
  - before
  - around
  - after
    - after
    - after returning
    - after throwing

Language: Advice

```
before(): move() {
    System.out.println("about to move");
}


after() returning: move() {
    System.out.println("just successfully moved");
}
```

Language: Exposing context

We can also interact with parameters of pointcut

```
after(FigureElement fe, int x, int y) returning:
        ...SomePointcut... {
    System.out.println(fe + " moved to (" + x + ", " + y + ")");
}
```

Filling in an applicable pointcut

```
after(FigureElement fe, int x, int y) returning:
        call(void FigureElement.setXY(int, int))
        && target(fe)
        && args(x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ")");
}
```

Cut in all over the place

Language: Inter-type declarations

Suppose we want to have Screen objects observe changes to Point objects

where Point is an existing class.

We can implement this by writing an aspect

Each point has an instance field **observers** that keep track of the Screen objects that are observing Points.

Language: Inter-type declarations

Suppose we want to have Screen objects observe changes to Point objects

where Point is an existing class.

We can implement this by writing an aspect

Each point has an instance field **observers** that keep track of the Screen objects that are observing Points.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}
```

Language: Inter-type declarations

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    ...
}
```

Language: Inter-type declarations

```
pointcut changes(Point p): target(p) && call(void Point.set*(int));

after(Point p): changes(p) {
    Iterator iter = p.observers.iterator();
    while ( iter.hasNext() ) {
        updateObserver(p, (Screen)iter.next());
    }
}

static void updateObserver(Point p, Screen s) {
    s.display(p);
}
```

All together now

Language: Aspects

- Mix everything we've seen up to now and put it one or more modular units called Aspects.
- Looks a lot like a class!
- Can contain pointcuts, advice declarations, methods, variables ….
- Single instances (default behavior)

Log example

Language: Aspects

```
aspect Logging {
    OutputStream logStream = System.err;

    before(): move() {
        logStream.println("about to move");
    }
}
```

The methods we weave

Implementation

- Aspect weaving: makes sure that applicable advice runs at the appropriate join points.
- In AspectJ, almost all the weaving is done at compile-time to expose errors and avoid runtime overhead.
- cflow (and maybe others) require dynamic dispatch.

Developmental Aspects

Developmental Aspects

- What are some places Aspects can assist developmental processes
- Exist in along-side but apart from existing coding
- Tracing, profiling, logging, pre-post conditions, contract enforcement

Tracing

- Enabling tracing as an ‘weaved’ in process that doesn’t exist in production

```
aspect SimpleTracing {
    pointcut tracedCall():
        call(void FigureElement.draw(GraphicsContext));

    before(): tracedCall() {
        System.out.println("Entering: " + thisJoinPoint);
    }
}
```

Profiling and Logging

- Although many sophisticated profiling tools are available, and these can gather a variety of information and display the results in useful ways, you may sometimes want to profile or log some very specific behavior.

```
aspect SetsInRotateCounting {
    int rotateCount = 0;
    int setCount = 0;

    before(): call(void Line.rotate(double)) {
        rotateCount++;
    }

    before(): call(void Point.set*(int))
              && cflow(call(void Line.rotate(double))) {
        setCount++;
    }
}
```

Pre-Post Condition Checking

- "Design by Contract" style where explicit pre-conditions test that callers of a method call it properly and explicit post-conditions test that methods properly do the work they are supposed to.

```
aspect PointBoundsChecking {

    pointcut setX(int x):
        (call(void FigureElement.setXY(int, int)) && args(x, *))
        || (call(void Point.setX(int)) && args(x));

    before(int x): setX(x) {
        if ( x < MIN_X || x > MAX_X )
            throw new IllegalArgumentException("x is out of bounds.");
    }

}
```

Contract Enforcement

- The property-based crosscutting mechanisms can be very useful in defining more sophisticated contract enforcement.
- One very powerful use of these mechanisms is to identify method calls that, in a correct program, should not exist.

```
aspect RegistrationProtection {

    pointcut register(): call(void Registry.register(FigureElement));

    pointcut canRegister(): withincode(static * FigureElement.make*(..));

    before(): register() && !canRegister() {
        throw new IllegalAccessException("Illegal call " + thisJoinPoint);
    }
}
```

Production Aspects

Production Aspects

- What are some places Aspects can assist production code
- Expected to be enabled and in operation
- Change monitoring, Context passing, Consistent Behaviour,

Change Monitoring

- Sometimes simple functionality is hard to do explicitly.
- Ex. maintain a dirty bit associated with object having moved since last display occurred

```
aspect MoveTracking {
    private static boolean dirty = false;

    public static boolean testAndClear() {
        boolean result = dirty;
        dirty = false;
        return result;
    }

    pointcut move(): ... ;

    after() returning: move() {
        dirty = true;
    }
}
```

Change Monitoring: Cross-cutting concern

Consider implementing this functionality with ordinary Java:

1. there would likely be a helper class that contained the dirty flag, the testAndClear method, as well as a setFlag method.
2. Each of the methods that could move a figure element would include a call to the setFlag method.
3. Those calls, or rather the concept that those calls should happen at each move operation, are the crosscutting concern in this case.

AspectJ Advantages over Standard Implementation

1. The structure of the crosscutting concern is captured explicitly.
2. Evolution is easier
3. The functionality is easy to plug in and out.
4. The implementation is more stable.

Context Passing

- Consider implementing functionality that would allow a client of the figure editor to set the color of any figure elements that are created.
- Typically this requires passing a color, or a color factory, from the client, down through the calls that lead to the figure element factory.

Context Passing

- The following code adds after advice that runs only when the factory methods of Figure are called in the control flow of a method on a ColorControllingClient.

```
aspect ColorControl {
    pointcut CCClientCflow(ColorControllingClient client):
        cflow(call(* * (..)) && target(client));

    pointcut make(): call(FigureElement Figure.make*(..));

    after (ColorControllingClient c) returning (FigureElement fe):
            make() && CCClientCflow(c) {
        fe.setColor(c.colorFor(fe));
    }
}
```

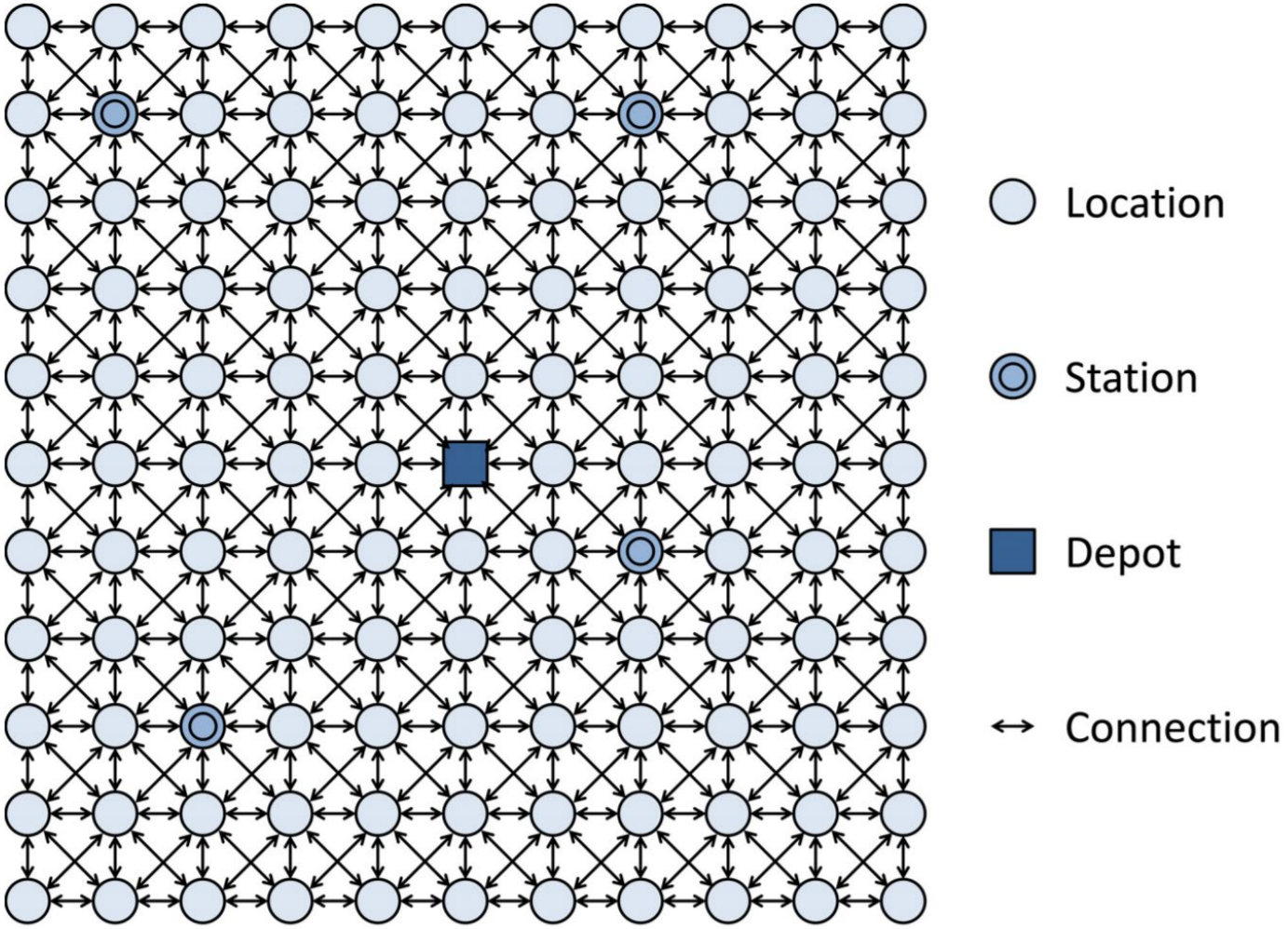
Providing Consistent Behavior

- This aspect ensures that all public methods of the com.bigboxco package log any Errors they throw to their caller

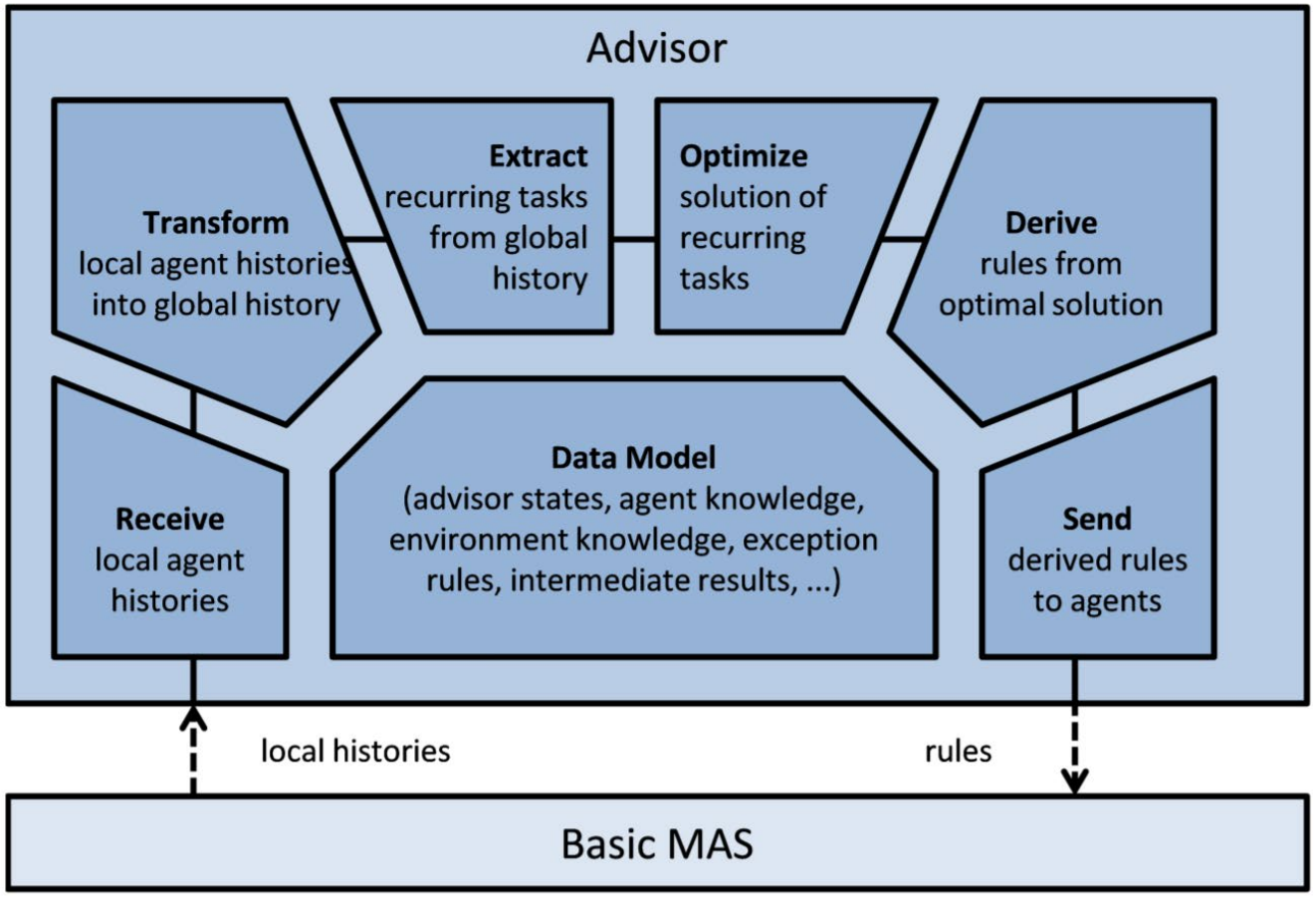
```
aspect PublicErrorLogging {
    Log log = new Log();

    pointcut publicMethodCall():
        call(public * com.bigboxco.*.*(..));

    after() throwing (Error e): publicMethodCall() {
        log.write(e);
    }
}
```

Complex Example from Research

My Experience: Advised Multi-Agent System



My Experience : Advised Multi-Agent System



My Experience: Advised Multi-Agent System

- Advisor monitors each agent when actions happened (collecting histories)
- From histories environment reconstructed, as well as agent behaviour
- Agent behaviour compared to optimal (ish)
- Rules to attempt to make agents act like optimal
- Rules added to agent

My Experience: Advised Multi-Agent System

- Aspects:
  - Advisor Aspect that hooks onto Agents when actions occur and records them (methods are called)
  - Also is able to notice when simulation runs have finished and do its number crunching to extract info, optimize, derive rules, and communicate them
  - Aspect around each agent to store advisor communicated rules and inter-cede in methods to change their behaviour decisions based on rules

My Experience: Advised Multi-Agent System

1. The MAS designer never had to change his code
2. The distributed aspect concerns related to the advisor were all centralized into very few classes, despite their interaction with code base being distributed
3. Could be flagged on and off at runtime

- The negative was a negligible runtime cost of hooking in aspects (the optimization AI step was much longer)
- Code always had to be run with additional configuration setup than basic Java code

Onward to … optimization.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY