Adding Open Modules to AspectJ

Neil Ongkingco, Pavel Avgustinov, Julian Tibble,
Oege de Moor, Ganesh Sittampalam
Programming Tools Group, University of Oxford, UK

Laurie Hendren
Sable Research Group, McGill University, Canada

The Trouble with Aspects

A simple aspect...

```
aspect ReplayAspect {
  pointcut translate(int dx, int dy): call(* Figure.translate(int, int)) && args(dx,dy);
  LinkedList moves = new LinkedList();

  before(int x, int y, Figure fig) : translate(x,y) && target(fig) {
    //Store fig, x and y in the moves list
  }
}
```

Applied to a simple class...

```
public class Figure {
  List /*<Point>*/ elements;
  public Figure translate(int dx, int dy) {
    for (Iterator iter = elements.iterator(); iter.hasNext();) {
      Point elem = (Point) iter.next();
      elem.translate(dx,dy);
    }
    return this;
  }
}
```

The Trouble with Aspects

But things rarely stay simple...

```
public class Figure {
 List /*<Figure>*/ elements;
 public Figure translate(int x, int y) {
  for (Iterator i = elements.iterator();
       iter.hasNext(); ) {
   Figure elem = (Figure) i.next();
   elem.translate(x,y);
  }
  return this;
 }
}
   aspect ReplayAspect {
    pointcut translate(int dx, int dy):
         call(* Figure.translate(int, int)) && args(dx,dy);
    LinkedList moves = new LinkedList();

    before(int x, int y, Figure fig) : translate(x,y) && target(fig) {
     //Store fig, x and y in the moves list
    }
   }
```

- The class Figure now contains other Figures
- ReplayAspect now behaves incorrectly
- The translate pointcut also matches the *internal* calls to translate(), causing double entries

The Trouble with Aspects

The problem:

- Changing the base code can break the aspects
  - subtle, undetectable at compile-time
  - check all aspects on every change (impossible)
- Aspects have unrestricted access to base code
  - Need to look at aspects to determine behavior
  - Makes it hard to enforce invariants on base code

A Solution

Open Modules (Aldrich)

- Interface (signature) between aspects and base code
- Specifies which events can be advised
- Internal advice has full access
- Module inclusion
- Defined for a small functional aspect language
  - call() only primitive
  - Module inclusion restricts exposed events

A Solution

Open Modules in Java terms:

```
public class Figure : expose (* call Figure.translate(..) &&
                                 !within(Figure))
                        friend DebugAspect {
  List /*<Figure>*/ elements;
  public Figure translate(int x, int y) {
    for (Iterator i = elements.iterator();
           iter.hasNext(); ) {
      Figure elem = (Figure) i.next();
      elem.translate(x,y);
    }
    return this;
  }
}
```

This visibility signature exposes only *external* calls to translate()

Friend aspects have full access to Figure's joinpoints

This is an *internal* call which aspects can not advise

Our Goal

- Adapt open modules to aspectJ
- Have the effect of visibility signatures
- Prevent scattering of signatures (not class annotations)
- Consistency with AspectJ

Open Modules in AspectJ

```
module FigureModule {
    class Figure;
    friend DebugAspect;
    advertise : call(Figure Figure.translate(int, int));
    expose to tracingaspects.* : call(* *(..));
    constrain FigureUtils;
}
```

- A module construct that contains classes and their visiblity/friend aspects
- Three main components:
  - Member classes and friend aspects
  - Visibility specification
  - Included modules

Open Modules in AspectJ

```
module FigureModule {
    class Figure;
    friend DebugAspect;
    advertise : call(Figure Figure.translate(int, int));
    expose to tracingaspects.* : call(* *(..));
    constrain FigureUtils;
}
```

- *class members* define the set of classes affected by visibility, friends
- Is an aspectJ class pattern (may contain wildcards, ..)

Open Modules in AspectJ

```
module FigureModule {
    class Figure;
    friend DebugAspect;
    advertise : call(Figure Figure.translate(int, int));
    expose to tracingaspects.* : call(* *(..));
    constrain FigureUtils;
}
```

- *friend* aspects have full access
- Order of the friends list also define precedence
- The list ***may not*** contain wildcards

Open Modules in AspectJ

```
module FigureModule {
   class Figure;
   friend DebugAspect;
   advertise : call(Figure Figure.translate(int, int));
   expose to tracingaspects.* : call(* *(..));
   constrain FigureUtils;
}
```

- Visibility specification limits the set of joinpoints accessible by *external* aspects
- Applies to joinpoints owned by class members
- Two forms:
  - **advertise**: only the external matches of the pointcut
  - **expose**: all matches

Open Modules in AspectJ

```
module FigureModule {
    class Figure;
    friend DebugAspect;
    advertise : call(Figure Figure.translate(int, int));
    expose to tracingaspects.* : call(* *(..));
    constrain FigureUtils;
}
```

- Included modules are affected by the module's friend aspects and signature
- Two forms
  - **constrain**: restricts the included module's signature
  - **open**: expands signature, propagates friends
- Order also defines precedence

Open Modules in AspectJ

- Joinpoint ownership
  - The visibility specification of a class only affects the joinpoints "owned" by that class

Joinpoint	Owner
call and set/get	Class where the method/field was declared
All else	Where the joinpoint (shadow) occurs

Open Modules in AspectJ

```
module FigureModule {
    class Figure;
    friend DebugAspect;
    advertise : call(Figure Figure.translate(int, int));
    expose to tracingaspects.* : call(* *(..));
    constrain FigureUtils;
}
```

Compiling produces this warning:

```
Figure.java:7: Warning -- An advice in aspect ReplayAspect
would normally apply here, but does not match any of  the
signatures of module FigureModule
                                elem.translate(x,y);
                                     ^-------------^
```

Normal Form

- Normal form fully defines module's effect
- To get a normal form:
  - Disjoin class members, collect friend aspects into a single list
  - Convert **advertise**():<pc> signatures to **expose**(): <pc> && ! **within**(<classes>)
  - Convert **to** clauses to **thisAspect**(<aspectpattern>)
    - **thisAspect**(A) is true if the aspect being woven matches A
  - Collect all signatures into a single disjunction (||)

Normal Form

- Module

```
module FigureModule {
    class Figure;
    class Point;
    advertise : call(* translate(int, int));
    expose to tracingaspects.* : call(* *(..));
    friend DebugAspect;
    friend Logger;
}
```

- Normal Form

```
module FigureModule {
    class Figure || Point;
    expose : (call(* translate(int, int)) && !within(Figure || Point))
      || (call(* *(..)) && thisAspect(tracingaspects.*));
    friend DebugAspect, Logger;
}
```

Precedence

- Order of friend aspects determine precedence

```
module M1 {
    class C1;
    friend A1, A2;
}
```

$\longrightarrow$

```
class C1 {
    //contents
}
declare precedence : A1, A2;
```

- Coexists with declare precedence statements
  - Adds equivalent declare precedence statements
- The syntax now is more consistent with the declare precedence syntax
  - The order in the friend list is now the same in declare precedence (not so in the paper)

Module Inclusion

- A module can include other modules
  - Affects *included* module
  - Hierarchical structure for class visibility
  - Also defines precedence
- Two forms of inclusion
  - **Constrained**
    - Restricts signature of included module
    - Does not propagate friend status to included module
  - **Open**
    - Expands the signature of the included module
    - Propagates friend status to included module

Constrained Module Inclusion

Module

```
module M1 {
    class C1;
    friend A1, A2;
    expose: A1.pointcut1();
}

module M2 {
    class C2;
    friend A3;
    constrain M1;
    friend A4;
    expose A4.pointcut2();
}
```

Normal Form

```
module M1 {
    class C1;
    friend  A1, A2;
    expose:
        (A1.pointcut1() && A4.pointcut2())
      ||
        (A1.pointcut1() && thisAspect(A3 || A4)
}

module M2 {
    class C2;
    friend A3, A4;
    expose A4.pointcut2();
}
```

Open Module Composition

Module

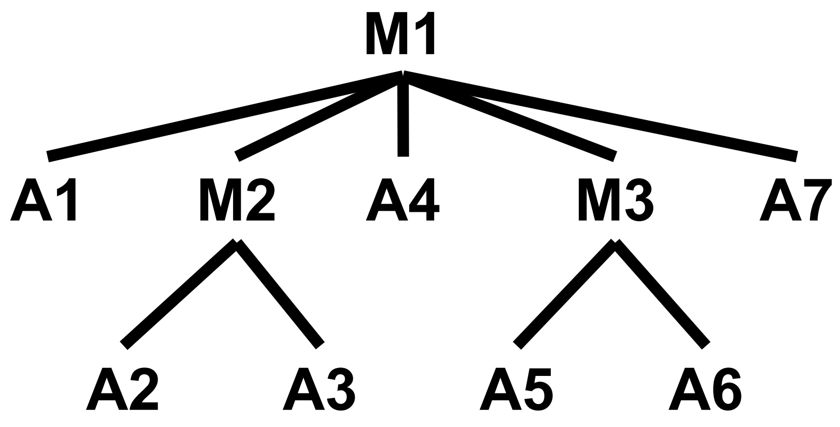
```
module M1 {
   class C1;
   friend A1, A2;
   expose: A1.pointcut1();
}

module M2 {
   class C2;
   friend A3;
   open M1;
   friend A4;
   expose A4.pointcut2();
}
```

Normal Form

```
module M1 {
   class C1;
   friend A3, A1, A2, A4;
   expose:
      (A1.pointcut1() || A4.pointcut2())
}

module M2 {
   class C2;
   friend A3, A4;
   expose A4.pointcut2();
}
```

Inclusion and Precedence

- In general inclusion forms a tree:



- Is equivalent to a single declare precedence

declare precedence: A1, A2, A3, A4, A5, A6, A7;

- Produces a total order on aspects

Restrictions on Modules

- A class can only occur in one module
  - Prevents overriding a class' visibility specification
- Inclusion should not form a cycle
  - Checked at compile time, throws an error
- A module can only be included at most once
  - As with classes, prevents visibility overriding
  - Contributes to a total precedence order on aspects

Restrictions on Modules

- An aspect can only be declared a friend in one module
  - Is propagated only in the normal form
  - Need to find the “right place” to put the aspect
  - Produces a total order on the precedence of aspects
- Module precedence consistent with declare precedence
  - Causes a compile time precedence error otherwise

Design Issues and Decisions

- Level of abstraction (module)
  - Above classes and packages
  - A class only appears in one module
- Support for AspectJ pointcuts
  - Static primitives (execution, within)
  - Dynamic primitives (cflow, args, if)
  - Named pointcuts (promotes modularity)
- Visibility signatures
  - advertise/expose : external/all joinpoints
  - friend aspects: full access (debugging aspects)

Design Issues and Decisions

- Inclusion
  - Loosely modeled after class inheritance
  - Two different types:
    - constrained: restricts exposed joinpoints
    - open: extend exposed joinpoints
  - A module can only be included *once*
    - Similar to single inheritance
- Precedence
  - Order in friend list defines aspect precedence

Design Issues and Decisions

- Namespace
  - Separate from Java and AspectJ
  - Modules must be on a different file (not in Java/AspectJ code)
  - Allows introduction/removal of modules without forcing invasive changes to existing code

Implementation

- Open modules for AspectJ was implemented in version 1.1.0 aspectbench compiler (abc)

www.aspectbench.org

- abc has proved to be a flexible enough framework for open modules
- Implementation did highlight some extensibility issues
  - Matcher extension

Related Work

- Open Modules (Aldrich)
- Pointcut interfaces (Gudmundson, Kiczales)
- Aspect-aware interfaces (Mezini, Kiczales)
- Spectators and Assistants (Clifton, Leavens)
- Pure Aspects (Dantas, Walker)

Future Work

- Formal model
- Restricted expose to specific aspect types
  - expose to pure <aspects> : <pointcut>
  - expose to @logger <aspects> : <pointcut>
- Restricting inter-type declarations
- Possible new features at the module level
  - Aspect composition (beyond precedence)
  - Aspect instantiation/overriding

Thank You

Any questions?

Appendix

Open Modules in AspectJ

```
[root] module <module_name> {
    class <classname_pattern>;
    friend <aspect_name>;
    [private] advertise [to <aspect_pattern>] : <pointcut>;
    [private] expose: [to <aspect_pattern>] : <pointcut>;
    open <module_name>;
    constrain <module_name>;
}
```

Three main components:

- **class** members and **friend** aspects
- visibility signature: **advertise** and **expose**
- Included modules: **open** and **constrain** modules

Open Modules in AspectJ

- Class members

```
class <classname_pattern>;
```

  - The set of classes to which the visibility signature applies
- Friend aspects

```
friend <aspect_list>;
```

  - A list of aspects that are allowed full access to the class members
  - The order of the list also defines the precedence of the aspects

Open Modules in AspectJ

- Visibility signature

  [**private**] <**expose**|**advertise**> [**to** aspect_pattern]: <pointcut>

  - Specifies the visible joinpoints of the class members
  - **expose**: exposes all joinpoints matched by the pointcut
  - **advertise**: exposes only external joinpoints matched by the pointcut (has an implicit !**within**(<class members>))
  - **to** clause: exposes the pointcut to a specific set of aspects
  - **private** modifier: signature only applies to immediate class members (not included modules)

Open Modules in AspectJ

- Included Modules

```
open <module_name_list>;
constrain <module_name_list>;
```

  - Specifies the set of included modules
  - Module inclusion modifies the signature of the classes in the included module
  - **open**: disjoins (||) the visibility of the including module with the included module
  - **constrain**: conjoins (&&) the visibility of the including module with the included module
  - Inclusion also affects precedence and the effect of friend aspects (more later...)

Private Signature Modifier

- Allows a signature to be added without affecting included modules

Module

```
module M1 {
   class C1;
   friend A1;
   open M2;
   expose: A1.pointcut1();
   private expose: A1.pointcut2();
}

module M2 {
   class C2;
   friend A2;
   expose A2.pointcut3();
}
```

Normal Form

```
module M1 {
   class C1;
   friend A1;
   expose:
      (A1.pointcut1() || A1.pointcut2());
}

module M2 {
   class C2;
   friend A1, A2;
   expose A1.pointcut1() || A2.pointcut3();
}
```

Root Module Modifier

- Some modules should not be included in others
  - Master module enforcing global constraints
  - Prevents new modules from overriding the constraints
- A root module can not be included in other modules
- An example:

```
root module MasterModule {
    constrain M1, M2, M3;
    expose: !call(* new(..));
}
```

Inclusion and Precedence

- The order of inclusion specifies the order of aspects in the included modules

Module

```
module M1 {
   class C1;
   friend A1;  open M2;
   friend A4;  open M3;
   friend A7;
}
module M2 {
   class C2;
   friend A2, A3;
}
module M3{
   class C3;
   friend A5,A6;
}
```

Normal Form

```
module M1 {
   class C1;
   friend A1, A4, A7;
}

module M2 {
   class C2;
   friend A1,A2, A3,A4,A7;
}

module M3 {
   class C3;
   friend A1,A4,A5,A6,A7;
}
```

Inclusion and Precedence

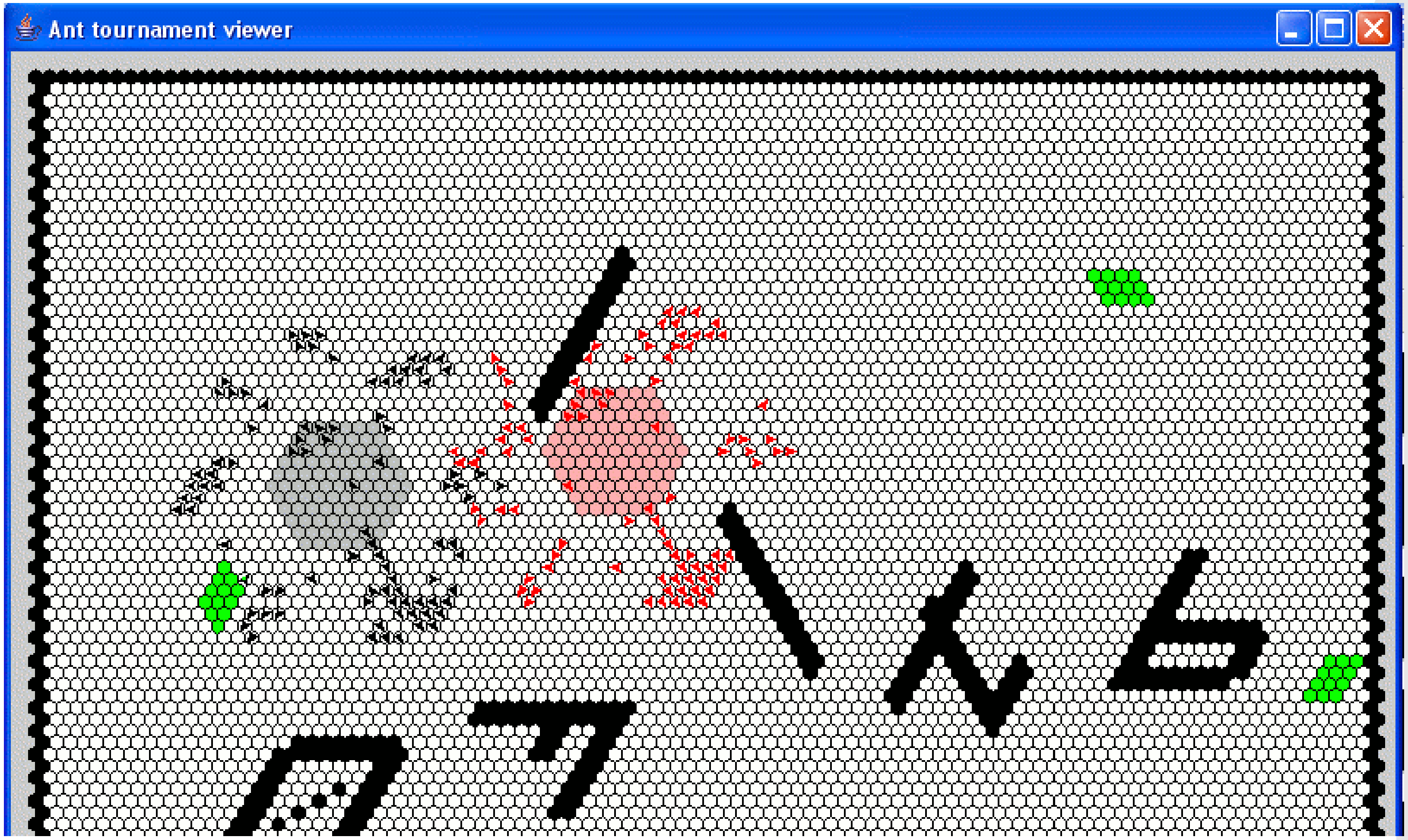
- The precedence order defined in module inclusion is consistent with a total order of the aspects

```
module M1 {
    class C1;
    friend A1, A4, A7;
}

module M2 {
    class C2;
    friend A1,A2, A3,A4,A7;
}

module M3 {
    class C3;
    class A1,A4,A5,A6,A7;
}
```

$\longrightarrow$

```
class C1 {/*contents*/}
class C2 {/*contents*/}
class C3 {/*contents*/}
declare precedence :
    A1, A2, A3, A4, A5, A6, A7;
```

An Example: Ants

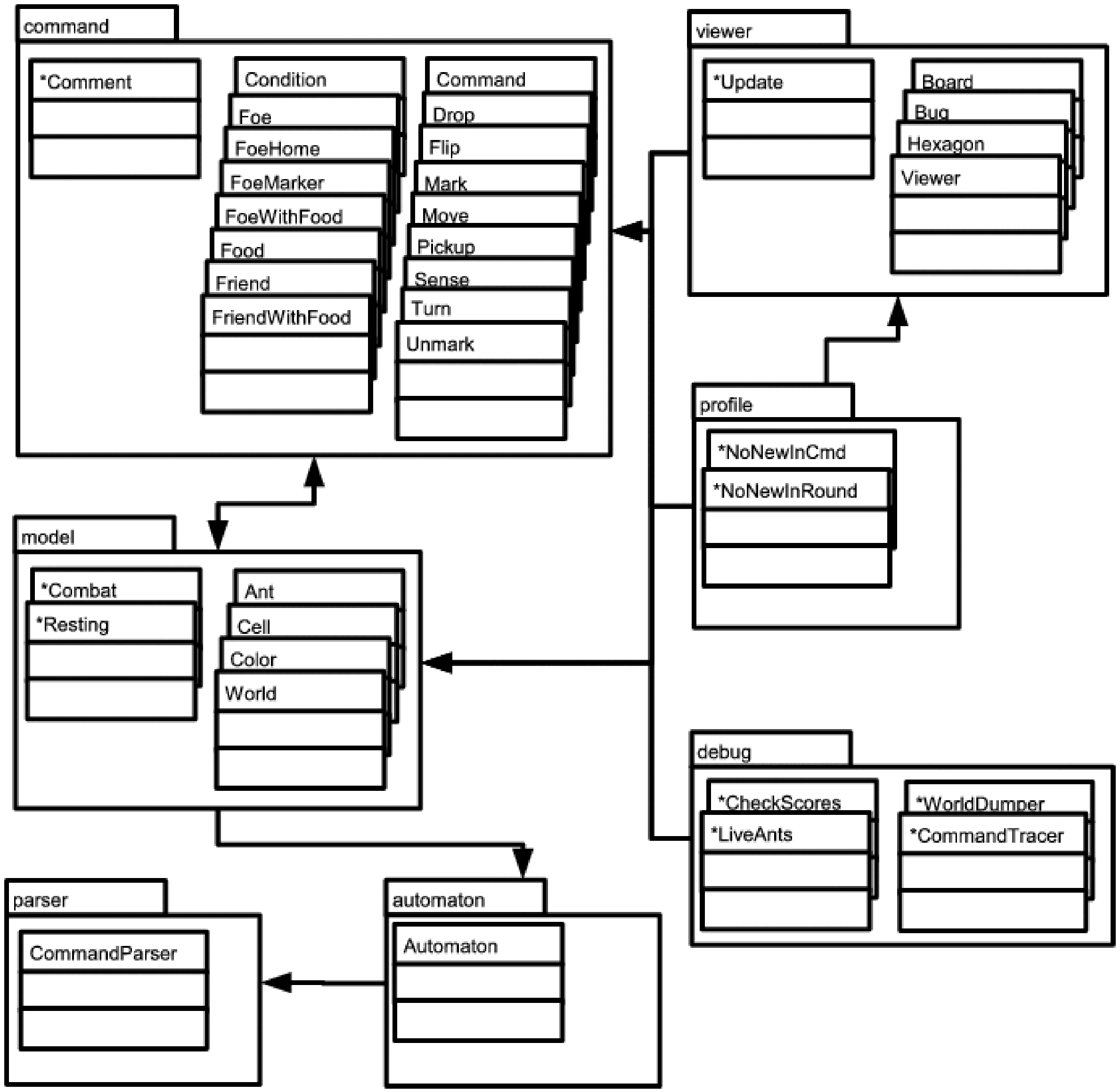


An Example: Ants

Ants

- Ant simulator and visualizer
- 27 classes, 10 aspects in 7 packages
- Core simulator
  - Loads ant spec and runs the simulation
  - Packages: *automaton, command, model, parser*
- GUI
  - Visualizer
- Debugging aspects
- Profiling aspects

An Example: Ants



An Example: Ants

Aspects:

- Package *automaton*
  - aspect *Comment*
- Package *model*
  - aspects Combat, Resting
- Package *debug*
  - aspects *CheckScore, CommandTracer, WorldDumper, LiveAnts*
- Package *profile*
  - aspects *NoNewInRound, NoNewInCmd*

Ants Module Specification

- Open modules can be used to make the class-aspect interfaces explicit
- Expose only the joinpoints that are to be advised by aspects
- **advertise** preferred, followed by **expose to** and finally **expose**
- Debugging and profiling aspects are invasive
  - Explicit **advertise**/**expose** too tedious
  - Access given by **friend** status and **open** inclusion

Ants Module Specification

```
module Model {
    class model.*;
    class automaton.*;
    friend model.Combat, model.Resting;
    advertise : call(* model.World.round());
    expose : call(* model.Ant.kill());
}
module Command {
    class command.*;
    class parser.*;
    friend command.Comment;
    advertise : call(* command.Command.step(..));
}
```

A module can contain classes across multiple packages

Ant.kill exposed as most calls are internal to Model

Ants Module Specification

```
module DebugAndProfile {
    class profile.*;
    class debug.*;
    friend profile.NoNewInCmd, profile.NoNewInRound;
    friend debug.WorldDumper, debug.LiveAnts,
        debug.CommandTracer, debug.CheckScores;
    open Model, Command;
}
module AntSystem {
    class viewer.*;
    friend viewer.Update;
    constrain DebugAndProfile;
    private expose to profile.*: call(*.new(..));
}
module JavaLang {
    class java.lang.*;
    advertise : !call(java.lang.StringBuffer.new(..));
}
```

Debugging and profiling aspects given access to Model and Command by open inclusion

Profiling aspects are given access to constructor calls in viewer classes

Hides calls to StringBuffer constructors, prevents matching String literals

Open Modules and Tool Support

- Just tool support is not enough to modularize aspects
  - Can only show you when advice applies at a specific point
  - Cannot prevent aspects from matching into your code
- This unbounded quantification becomes a problem when using 3rd party aspects and aspect libraries

Open Inclusion and Modularity

- Open inclusion expands the set of joinpoints of a module
  - This possibly overrides the assumptions of the modules creator
- Once a module has been opened up, the creator of the including module takes responsibility for the modularity of all modules included using **open**
  - Would have to adapt if any change in the included modules causes the system to break