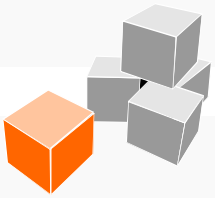
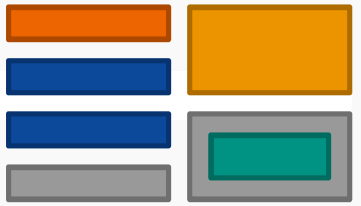
Twente Research and Education on Software Engineering, Universiteit Twente



Software Technology Group
TU Darmstadt | FB Informatik



www.alia4j.org

Advanced-dispatching Language-Implementation Architecture for Java (ALIA4J)

http://www.alia4j.org

Christoph Bockisch, c.m.bockisch@cs.utwente.nl
Andreas Sewe, sewe@st.informatik.tu-darmstadt.de

CASED

Project Goal

- Much research in programming languages
  - New ways of composing and refining modules
  - Increase re-usability and other qualities
- New languages usually …
  - … extend existing, established language
  - … share concepts with other (research) languages
- ALIA4J: architecture for modular language implementation
  - Java as extended language
  - Lets designers re-use implementation of overlapping concepts ⇨ **reduce effort for designers**
  - Can even re-use optimization ⇨ **increase acceptance of new languages**
  - **Concise semantics**

Facts About ALIA4J

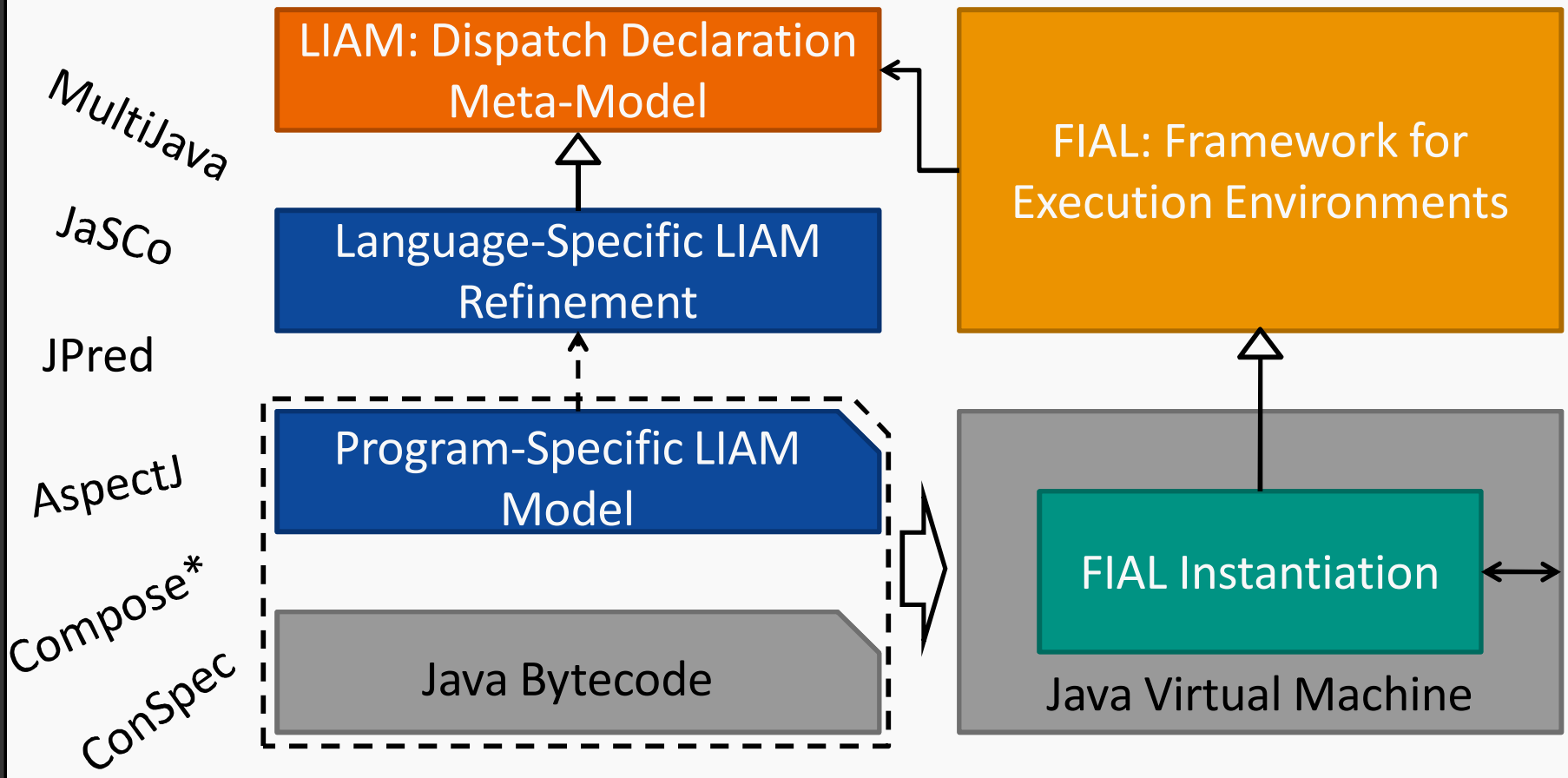
- Jointly lead by University of Twente, the Netherlands (Christoph Bockisch) and University of Darmstadt, Germany (Andreas Sewe)
- Prototypes reach back to 2003, in its current shape since ~2007
- Active developers
  - Me
  - Two PhD students currently working in ALIA4J (Darmstadt)
  - Seven master students currently working in ALIA4J (4 Twente, 3 Darmstadt)
- 14 completed student projects
- Used in APC course
- Publications
  - Two completed PhD theses
  - 13 conference or workshop publications
  - Multiple technical reports
- Partners
  - ALIA4J is reference execution model in AOSD Europe NoE
  - Parts of ALIA4J used as runtime layer in CASED
- Build process using maven
- More than 3700 integration tests
- Eclipse Public License

Facts About ALIA4J

- Jointly lead by University of Twente, the Netherlands (Christoph Bockisch) and University of Darmstadt, Germany (Andreas Sewe)
- Prototypes reach back to 2003, in its current shape since ~2007
- Active developers
  - Me
  - Two PhD students cu[illegible]y worki[illegible]nstadt)
  - Seven master students [illegible], 3 Darmstadt)
- 14 completed student pro[illegible]
- Used in APC cou[illegible]
- Publications
  - Two completed PhD th[illegible]
  - 13 conference or wo[illegible]
  - Multiple technic[illegible]
- Partners
  - ALIA4J is reference executi[illegible]nod[illegible]OSD[illegible]pe NoE
  - Parts of ALIA4J used as runtime lay[illegible]CASED
- Build process using maven
- More than 3700 integration tests
- Eclipse Public License

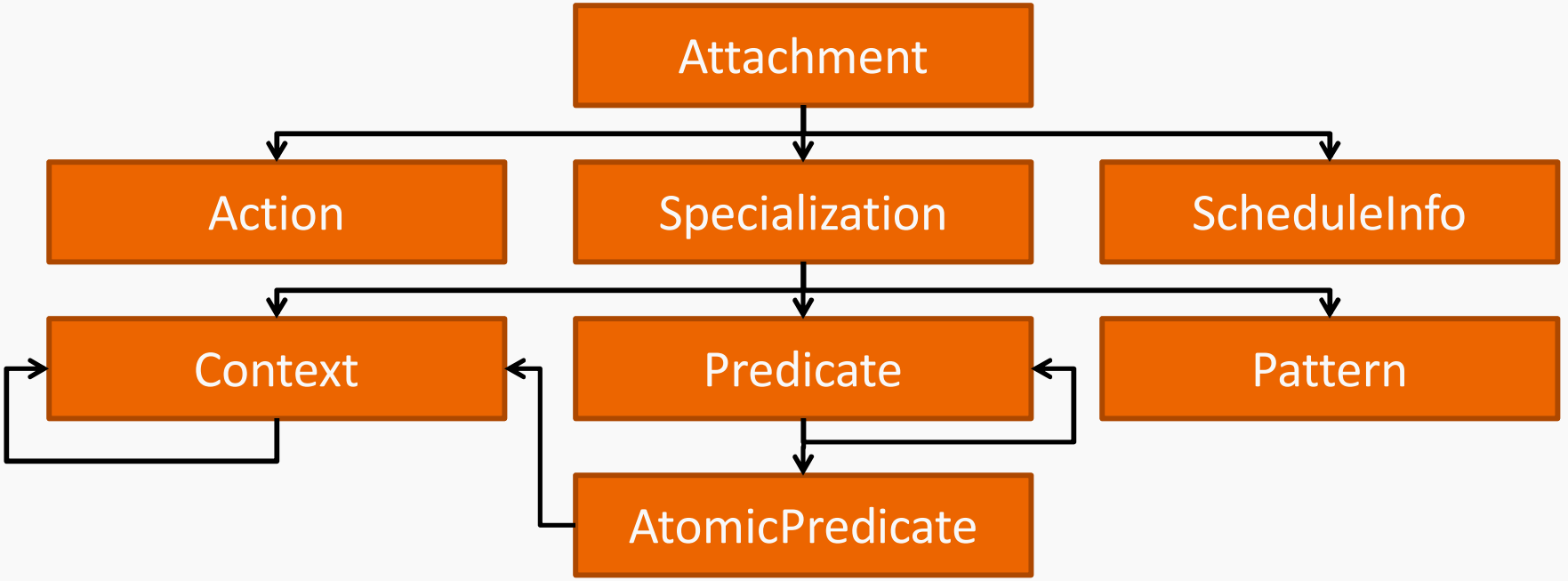
First official, reasonable release!

Advanced-Dispatching Language-Implementation Architecture (ALIA)

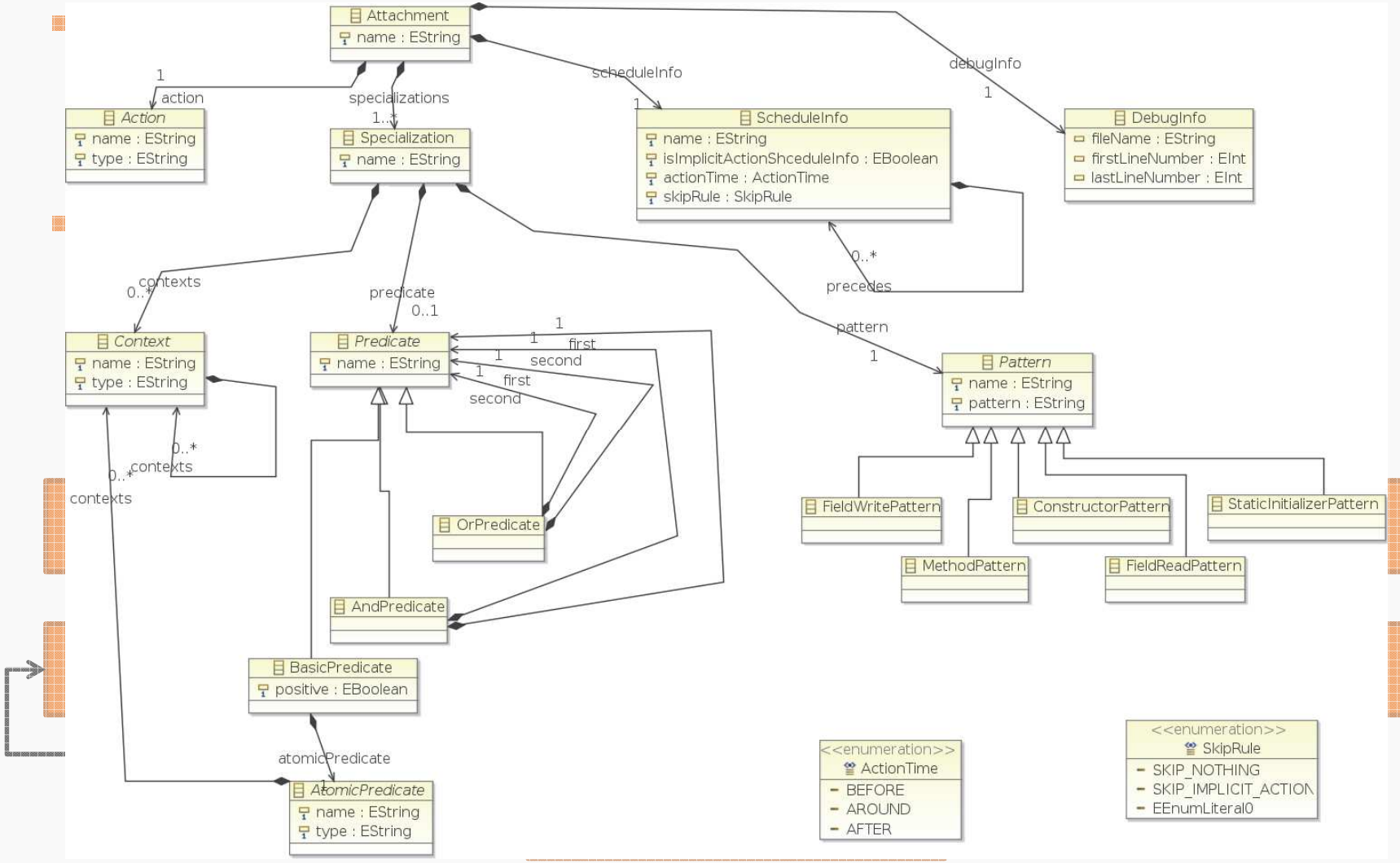


Dispatch Declaration Meta-Model

- Language support:
  - Implement or re-use meta-model refinements
  - Create model from (source) code
- Fine-grained meta-model: One concrete concept generally mapped to multiple meta-model entities



Dispatch Declaration Meta-Model



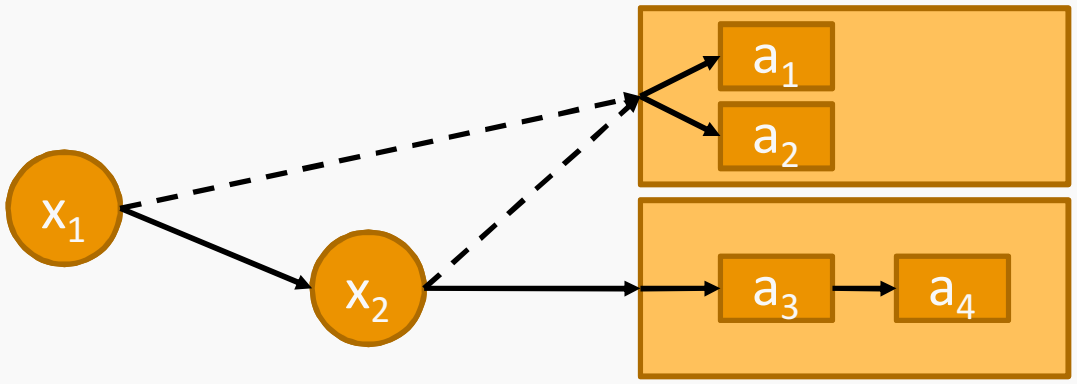
Dispatch Declaration Meta-Model

Meta-model entity	Example
Attachment	Pointcut-advice, superimposed filter, predicate method.
Specialization	Similar to pointcut, method signature in predicate dispatching.
Pattern	Property-based pointcut, event matching, e.g., “call”, “get”. In predicate dispatching pattern matches exactly one signature.
Predicate	Dynamic part of pointcut, predicate.
AtomicPredicate	Dynamic and scoping pointcut designator, e.g., “cflow”, “within”, “this”. Condition in Composition Filters.
Context	Context binding in pointcut, e.g., “this”. Can be used for instantiation strategy, e.g., “issingleton()”, “perThis()”.
Action	Advice, filter action, predicate method body.
ScheduleInfo	Advice precedence, overriding between predicate methods.

Dispatch Execution Model

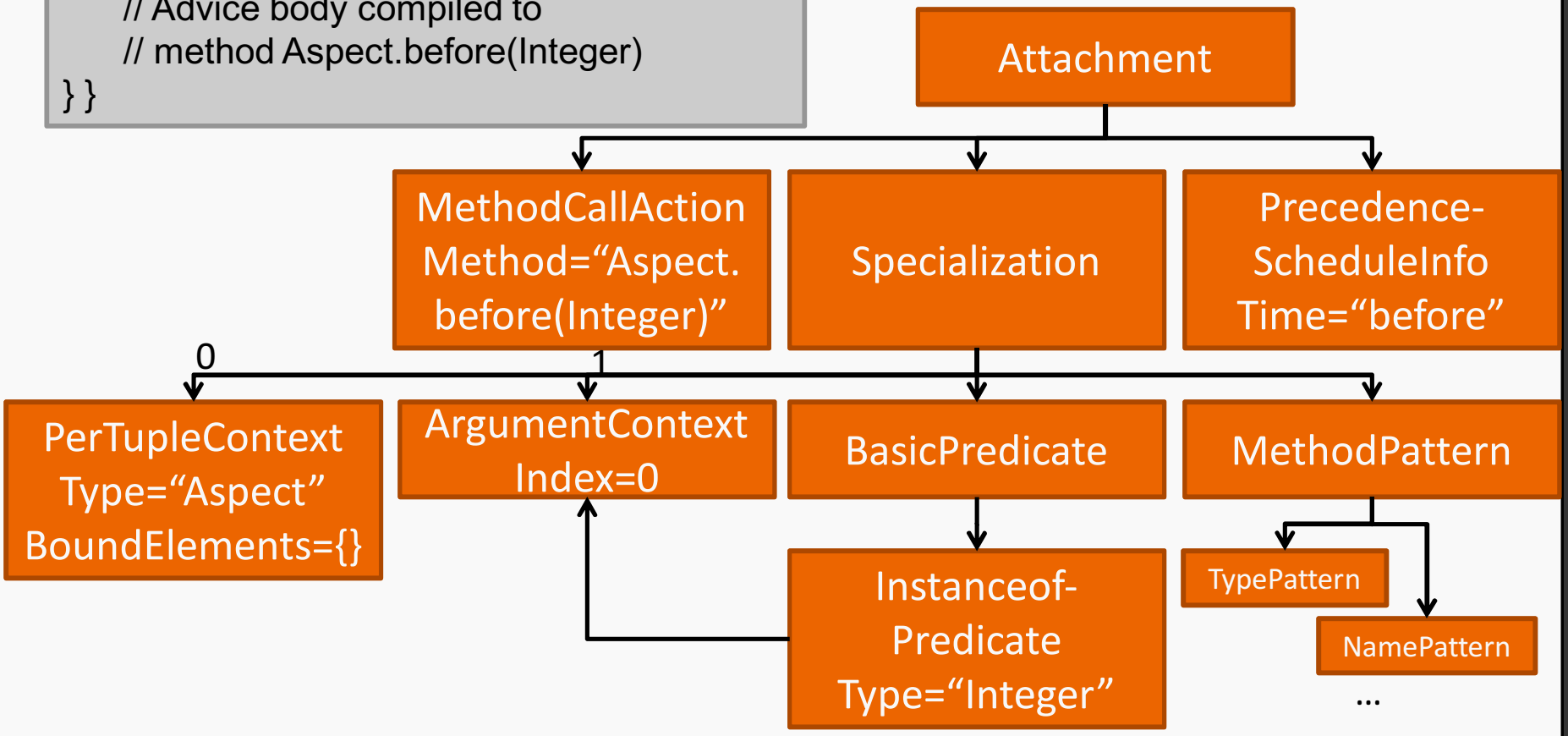
- FIAL partially evaluates dispatch declarations
  - Dispatch function as binary decision diagram
    - Optimize evaluation strategy
  - Actions to execute as tree
    - Supports nested actions like AspectJ's around
- Execution Model refers to dispatch declaration model entities

Example:

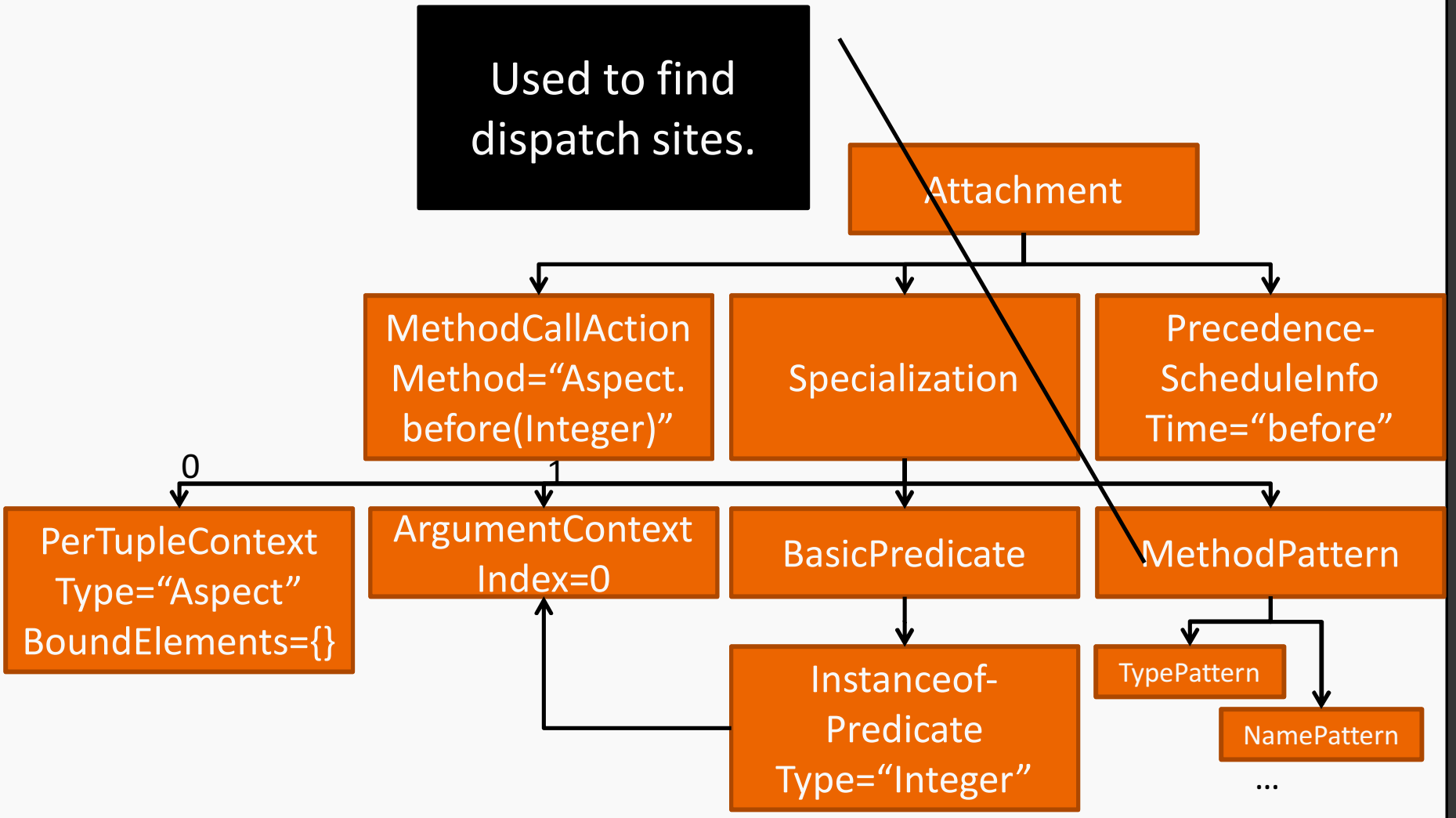


Example

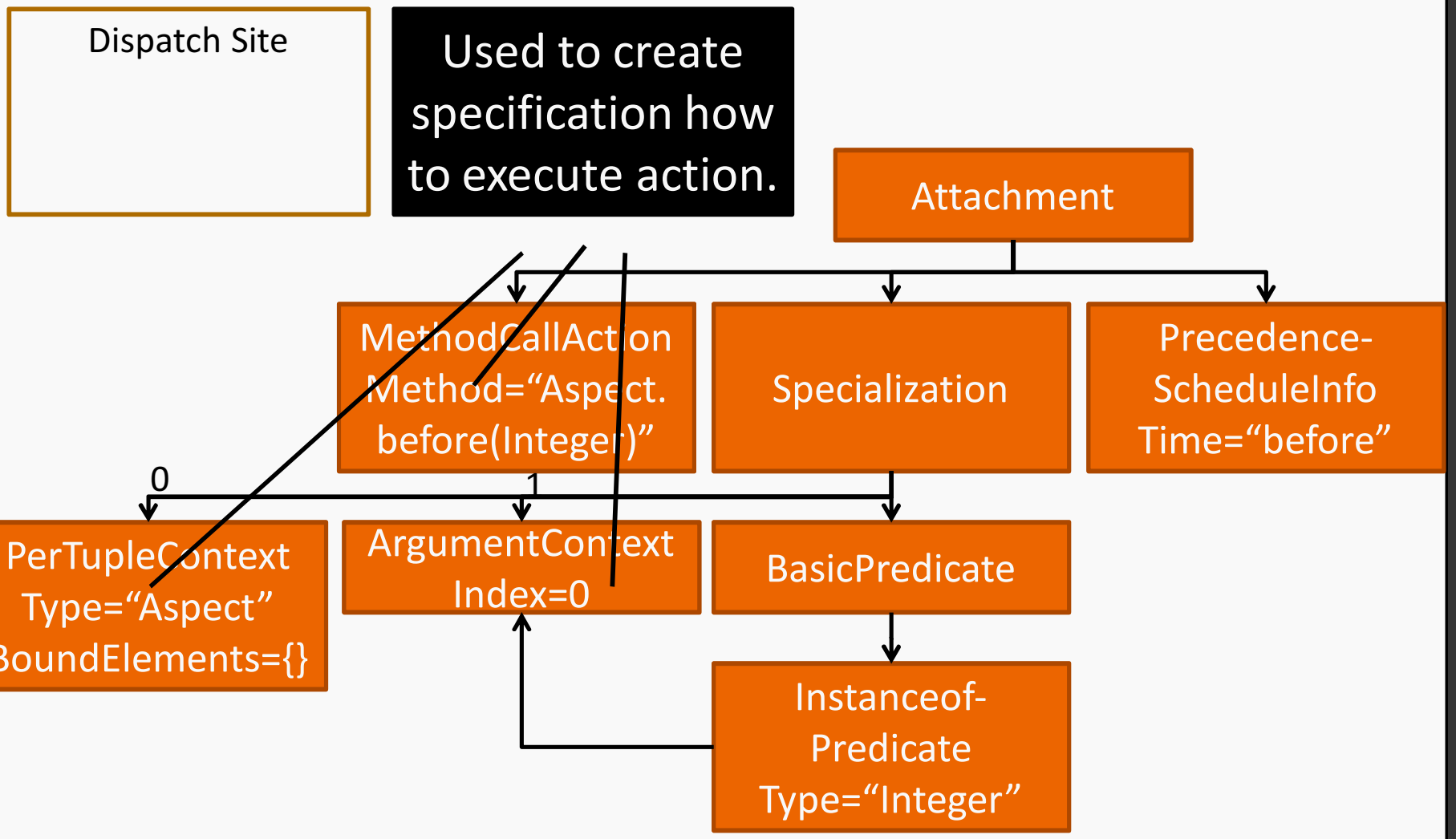
```
public aspect Aspect issingleton() {
  before(Integer i) :
    call(void *.m(Number)) && args(i) {
      // Advice body compiled to
      // method Aspect.before(Integer)
} }
```



Example

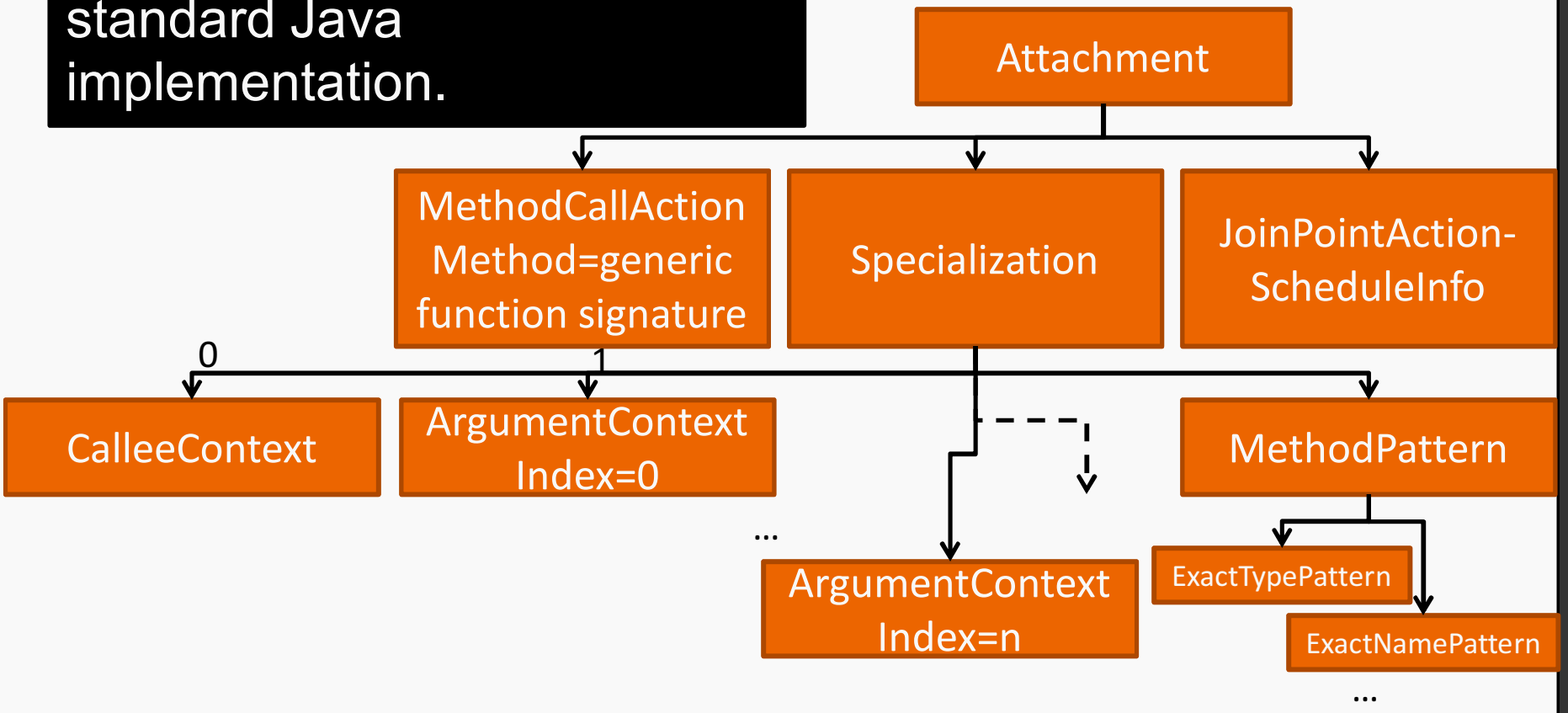

Used to find dispatch sites.
Attachment
MethodCallAction Method="Aspect. before(Integer)"
Specialization
Precedence-ScheduleInfo Time="before"
0
1
PerTupleContext Type="Aspect" BoundElements={}
ArgumentContext Index=0
BasicPredicate
MethodPattern
Instanceof-Predicate Type="Integer"
TypePattern
NamePattern
...

Example

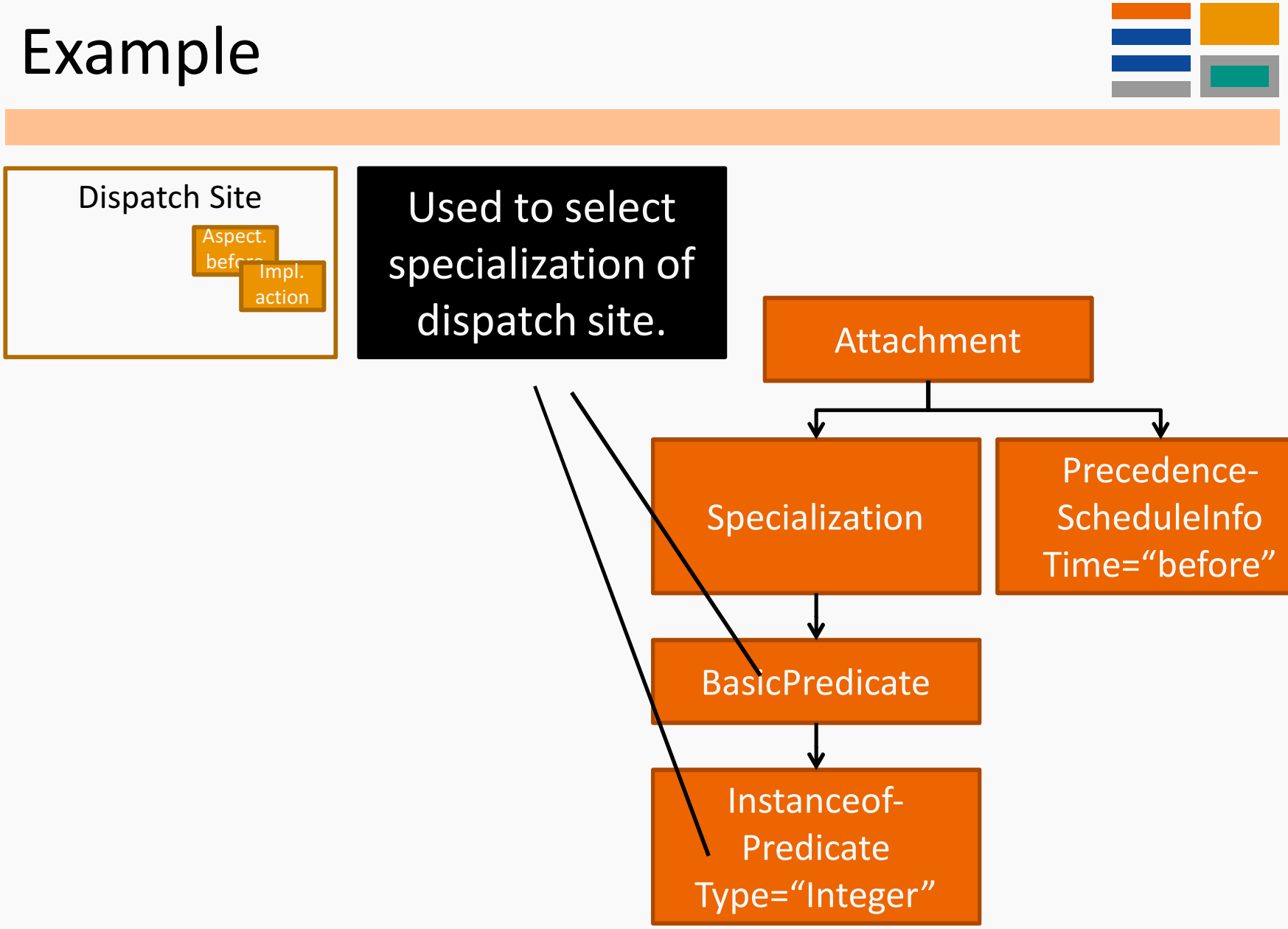

Dispatch Site
Used to create specification how to execute action.
Attachment
MethodCallAction Method="Aspect. before(Integer)"
Specialization
Precedence-ScheduleInfo Time="before"
0
1
PerTupleContext Type="Aspect" BoundElements={}
ArgumentContext Index=0
BasicPredicate
Instanceof-Predicate Type="Integer"

Side Note

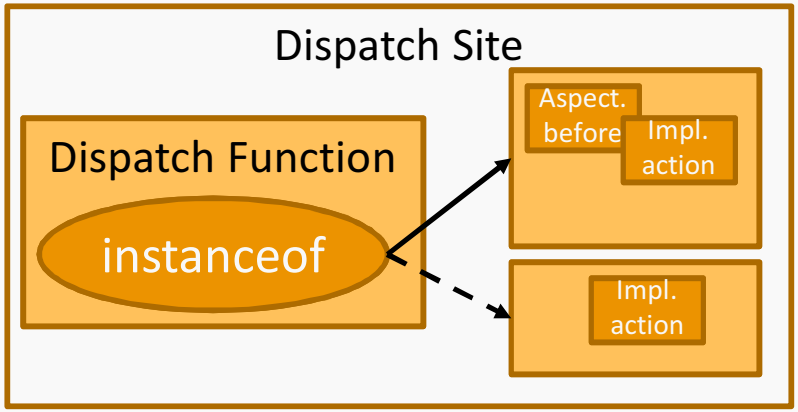
Every method implicitly has an attachment to call the standard Java implementation.



Example


Dispatch Site
Aspect.
Impl.
action
Used to select specialization of dispatch site.
Attachment
Specialization
Precedence-ScheduleInfo Time="before"
BasicPredicate
Instanceof-Predicate Type="Integer"

Example

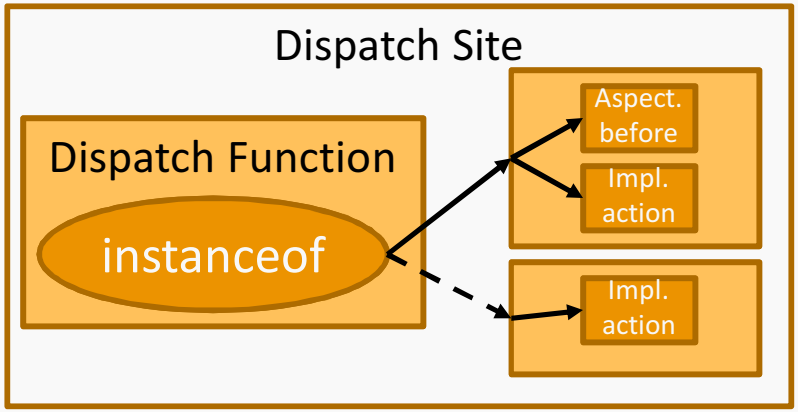


Attachment

Precedence-ScheduleInfo
Time="before"

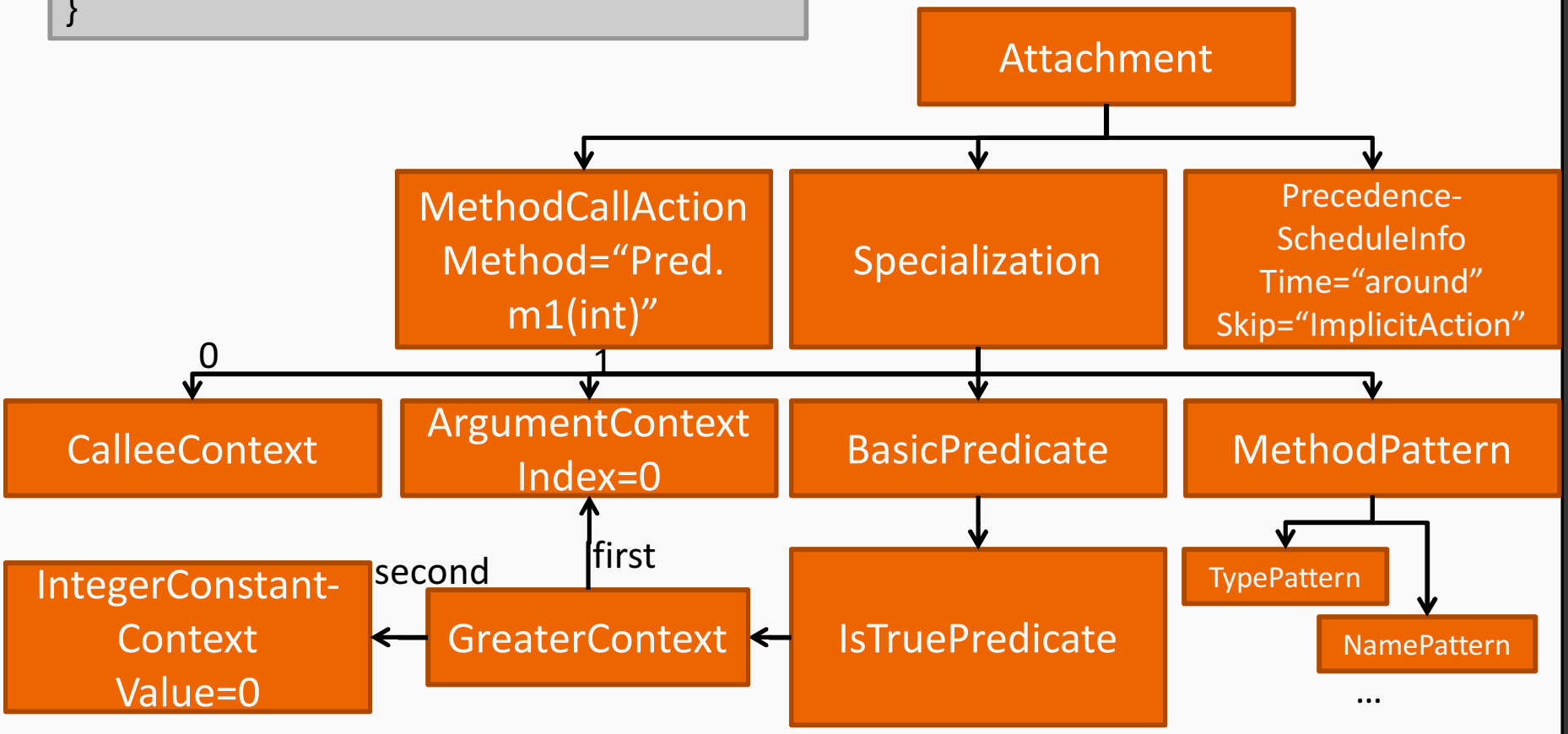
Used to resolve constraints among actions.

Example


Dispatch Site
Dispatch Function
instanceof
Aspect.
before
Impl.
action
Impl.
action

Example

```
void m (int i) where i > 0 {
    // method body compiled to
    // method Pred.m1(int)
}
```



ALIA4J Patterns

- Five kinds of patterns
  - MethodPattern, FieldReadPattern, FieldWritePattern, ConstructorPattern, StaticInitializerPattern
  - Must implement a method match(Signature), where Signature classes mirror Pattern classes
- Default implementations use (common) subpatterns
  - ModifiersPattern, TypePattern, ClassTypePattern, NamePattern, ParametersPattern, ExceptionsPattern
- Subpatterns have default implementations
  - ANY (constant in subpattern class)
  - Two forms for matching against rule: Wildcard… and Exact…
  - Top subpattern classes define boolean operators for subpatterns (and, or, not)

ALIA4J predicates

- Three implementations
  - AndPredicate, OrPredicate
    - Configured with two Predicate objects (subpredicates)
  - BasicPredicate
    - Configured with an AtomicPredicate and
    - A boolean specifying whether the AtomicPredicate must be satisfied or not
- Can express predicates in negated normal form

ALIA4J Factories

- Create entities using factories
  - ActionFactory, ContextFactory, AtomicPredicateFactory
  - (Abstract) factories define static methods that delegate to abstract factory methods
- Patterns, Attachments, Specializations, Predicates can be created using their constructors
  - Attachment, Specialization are final, i.e., always use default implementation
- ScheduleInfo: use factory method in ScheduleInfo
- DebugInfo:
  - Also part of an Attachment
  - Map to definition in source code

Execution Environments

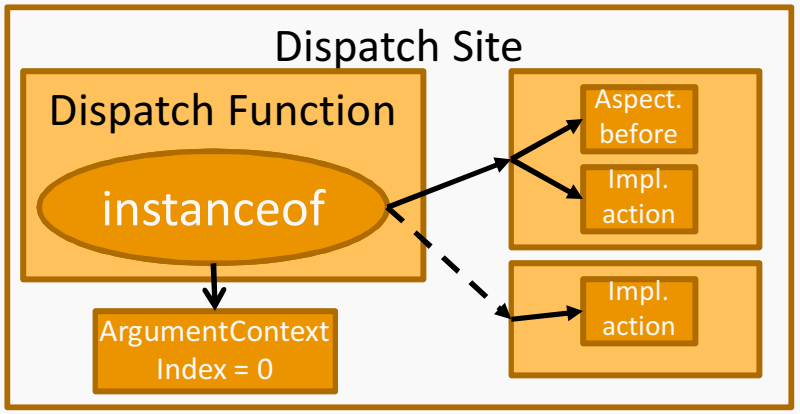
- Generic work flows implemented in FIAL
- Used by all concrete execution environments
  - Dynamic deployment
  - Re-deploy at dynamic class loading
  - Execution semantics of dispatch model
    - Allowing optimization strategies
  - Partial evaluation of attachments
  - “Importer” hook for preparing attachments
- Different concrete execution environments
  - Using interpretation (NOIRIn)
  - Using bytecode generation (SiRIn)
  - Integrated into just-in-time compiler of virtual machine (Steamloom^{ALIA})

Workflow: Execution of Dispatch Model

- Each entity can implement a java method ("compute" method) realizing its semantics
- At runtime
  - When dispatch site is reached
  - Look up dispatch model
  - Evaluate dispatch function
    - Successively call compute function of AtomicPredicates
  - Order applicable actions according to their schedule information
  - Execute applicable actions
    - Calling compute methods of Contexts, Actions

Example

```
public class Main {
  public static void main(String[] args) {
    new Main().m(new Integer(1));
  }
  public void m(Number n) {
    System.out.println("Main.m( " + n + " )");
} }
```

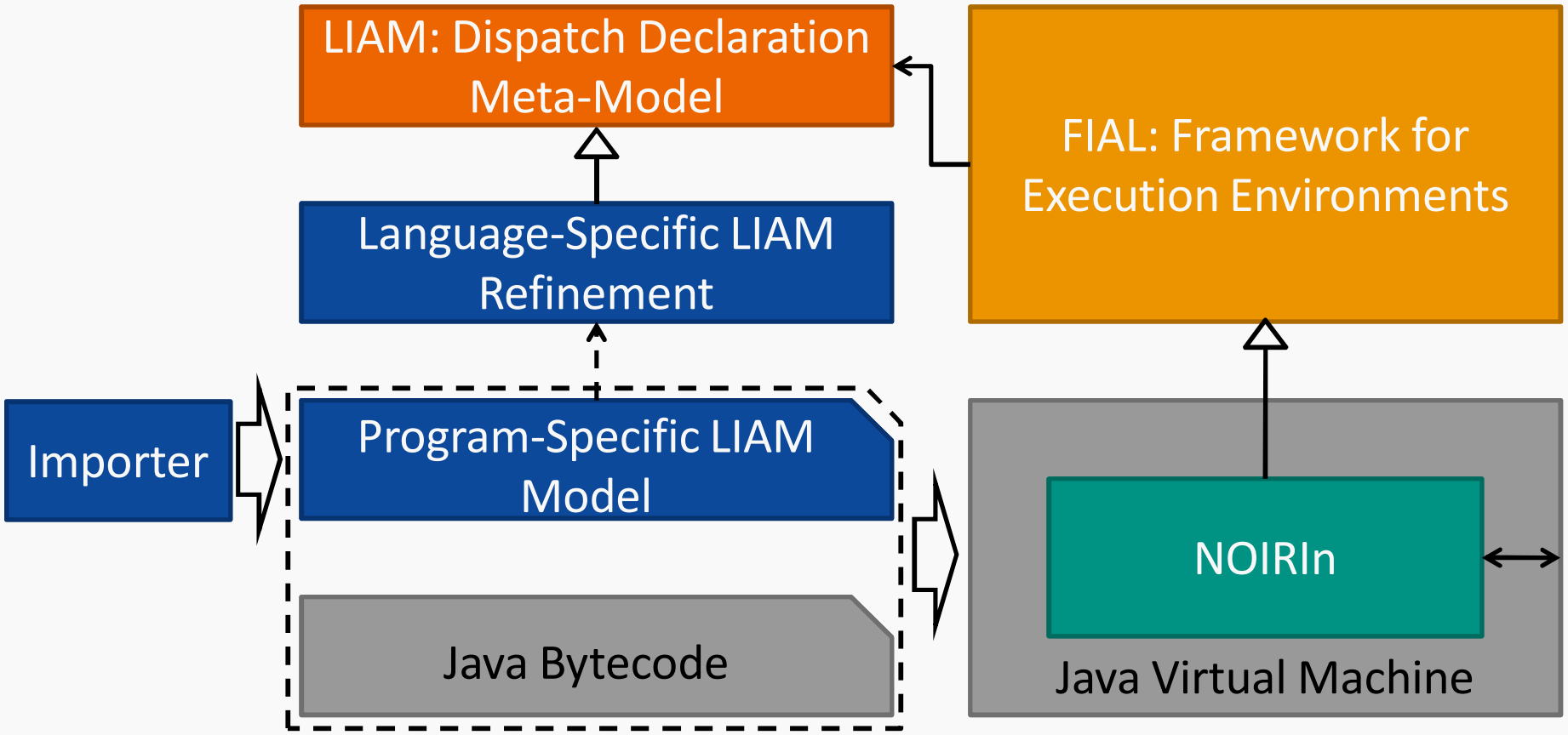


1. Find corresponding dispatch model
2. Get the ArgumentContext object and invoke "getObjectValue()"
3. Get the InstanceofPredicate object and invoke "isSatisfied(Object)", passing the result of step 2
4. If step 3 returns **`true`**, execute actions in top box, otherwise, those in bottom box
5. For each selected action, do similar steps, evaluating the exposed context values and executing the action

Optimized Code-Generation

- SiRIn additionally allows to implement bytecode-generation strategy per entity
  - Generated bytecode used instead of calling compute method
- Steamloom^{ALIA} additionally allows to implement native-code-generation strategy per entity
- Factories used to create LIAM entities
  - Allows execution environments to create specific implementations …
  - Implementing optimized code-generation strategies
  - Transparent for the user

Implementing Your Language



Importer Component

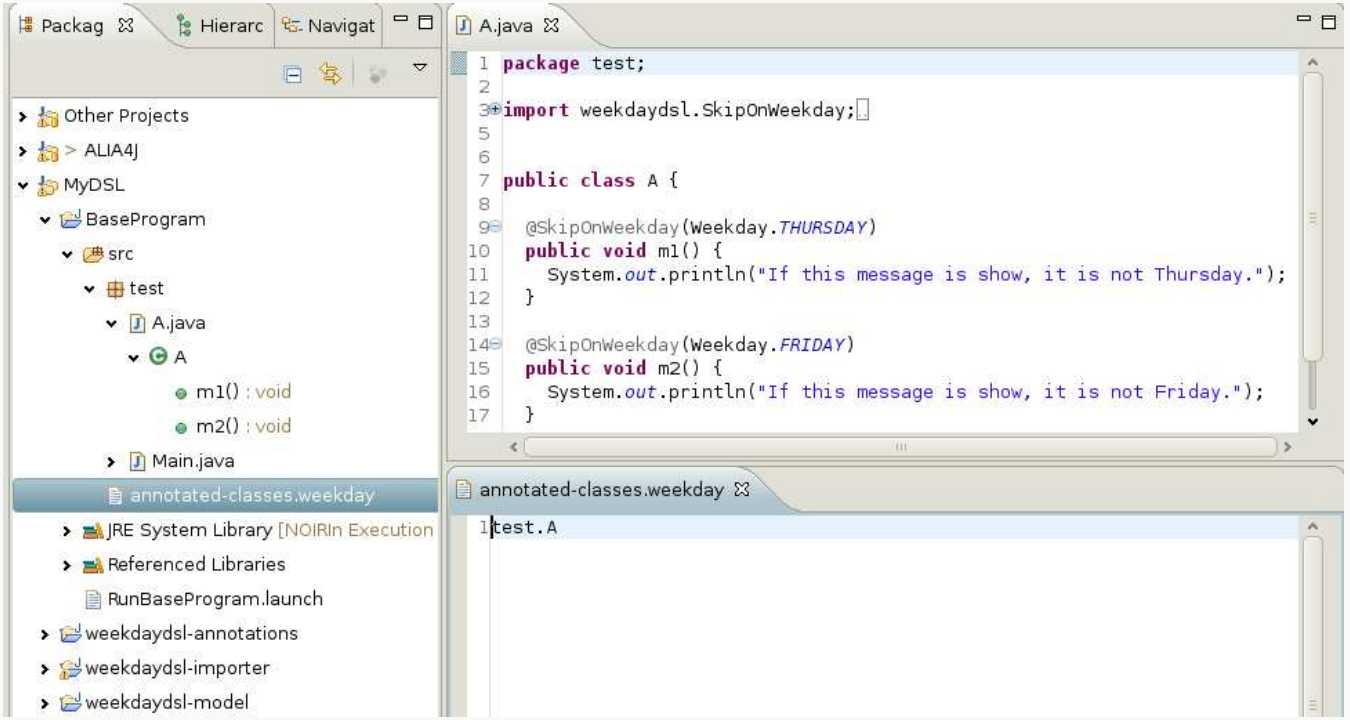
- FIAL declares the interface Importer
  - Importer is run
    - After ALIA4J is initialized
    - Right before application is started
  - Gets the application class loader
  - Can be used to set up/deploy attachments
  - Possibly read DSL from file and convert to Attachments using API calls
- Language designers can implement Importer interface
- Specify importer to be used when launching ALIA4J execution environment
- Importer is the way to integrate your language with ALIA4J

Example Language

- Some methods should be ignored on certain week days
- DSL has two parts:
  - A file in a specific syntax specifying which classes contain such methods
    - One fully qualified class name per line
  - Annotations in the specified classed that state at which days the method is to be omitted.
    - @SkipOnWeekday(Weekday)
    - Enumeration Weekday

Implementing the Importer

- Assume the definition file resides in the default package of the program and is called “annotated-classes.weekday”



Implementing the Importer

- The importer gets the application class loader
- Use this to read the file

```
public WeekdayDSLImporter(ClassLoader classLoader) {
  this.classLoader = classLoader;
}

@Override
public void performImport() {
  // ...
  InputStreamannotatedClassesDefinitionStream =
  classLoader.getResourceAsStream(
  "annotated-classes.weekday");
 // ...
```

Implementing the Importer

- Read in the specified classes
  - Scan for annotated methods
  - For each create/deploy an attachment
- Do not violate Java's lazy class loading:
  - Can use class loader to read class files as resource
    - Parse with bytecode toolkit
  - Do not use class loader to load class and search annotations using reflection

Creating the attachments

- Create a pattern for the method with the annotation
- Dynamic checks
  - Actual target method must be defined in correct type
  - Weekday must be as specified
- When checks succeed: skip implicit action

Creating the attachments

```
NamePattern namePattern =
  new ExactNamePattern(method.getName());
ParametersPattern parametersPattern =
  new ExactParametersPattern(TypeHierarchyProvider.
    createTypeDescriptors(method.getParameterTypes()));

// don't care about modifiers, declaring type, return type and exceptions
// we want to specify one method exactly and therefore
// it is sufficient to match for the declaring class, name and parameter types
MethodPattern methodPattern =
  new MethodPattern(ModifiersPattern.ANY,
    TypePattern.ANY, ClassTypePattern.ANY, namePattern,
    parametersPattern, ExceptionsPattern.ANY);

AtomicPredicate typePredicate = AtomicPredicateFactory.
  findOrCreateExactTypePredicate(
    ContextFactory.findOrCreateCalleeContext(),
    TypeHierarchyProvider.
    findOrCreateTypeDescriptor(method.getDeclaringClass()));
```

Creating the attachments

```
AtomicPredicate weekdayPredicate; // = ...
Predicate<AtomicPredicate> predicate =
    new AndPredicate<AtomicPredicate>(new
    BasicPredicate<AtomicPredicate>(typePredicate, true),
        new BasicPredicate<AtomicPredicate>(weekdayPredicate, true));
Specialization specialization = new Specialization(methodPattern, predicate,
     Collections.<Context>emptyList());

Action action =
    ActionFactory.findOrCreateNoOpAction();

ScheduleInfo scheduleInfo =
    ScheduleInfo.createScheduleInfo(ActionTime.DONT_CARE,
    SkipRule.SKIP_IMPLICIT_ACTION,
    Collections.<ScheduleInfo.PrecedenceScheduleInfo>emptySet());
Attachment attachment =
    new Attachment(Collections.singleton(specialization), action,
        scheduleInfo, DebugInfo.UNKNOWN_INFO);
```

Say “false” if the predicate is negated.

Deploying attachments

- When the importer has created all attachments, it has to deploy them

```
List<Attachment> attachments = new ArrayList<Attachment>();
Attachment[] attachmentsArray = new Attachment[attachments.size()];
attachmentsArray = attachments.toArray(attachmentsArray);
System. deploy(attachmentsArray);
```

Creating the predicate checking for the weekday

- Maybe you have to implement your own concepts

```
Context expectedObject =
  ContextFactory.findOrCreateObjectConstantContext(value);
Context actualObject =
  WeekdayDSLContextFactory.findOrCreateWeekdayContext();
AtomicPredicate weekdayPredicate =
  AtomicPredicateFactory.findOrCreateEqualsPredicate(
      expectedObject, actualObject);
```

Implementing your own concepts

- “Best practice”
  - Create a project mydsl-model
  - Add packages mydsl.context, mydsl.predicate, etc. when you implement contexts, atomic predicates, etc.
  - Implement an (abstract) factory for each kind of concept
  - Add static findOrCreateMyConcept methods which internally call createMyConcept on the factory’s singleton
  - The findOrCreate methods are supposed to canonize the entities

Implementing your own concepts

- Implement subclass of Context, AtomicPredicate, Target or Action
  - Pass required context values to super constructor
  - Implement “compute” method
    - Must accept values corresponding to required contexts
    - Follow naming convention:
      - AtomicPredicate: isSatisfied
      - Action: perform
      - Context: get<Type>Value
        - Where <Type> reflects the result type (Boolean, Char, Byte, Short, Int, Long, Float, Double, Object)
        - Also implement method getDeclaredResultType

Be careful!

- All (concept) classes must be public!
- All “compute” methods must be public!
- “compute” methods for all possible combinations of required context values must be implemented
  - Have a look at the required Contexts and what they may return as “getDeclaredResultType”
- If you make a mistake, the JVM may crash
  - It will tell you the location of a file containing a dump of the crash
  - If you cannot solve it yourself, post dump and all your code on forum

Implementing the WeekdayContext

```
public class WeekdayContext extends Context {
  protected WeekdayContext() {
    super(Collections.<Context>emptyList());
  }
@Override
  public Maybe<? extends Object> computeValueStatically(List<? extends Signed<?>>
callStack) {
    return new Maybe<Object>();
  }
@Override
  public SimpleType getDeclaredResultType(Signed<?> call) {
    return SimpleType.REFERENCE;
  }
public Object getObjectValue() {
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(new Date());
    switch (calendar.get(Calendar.DAY_OF_WEEK)) {
    case Calendar.MONDAY:
      return Weekday.MONDAY;
```

No other context required

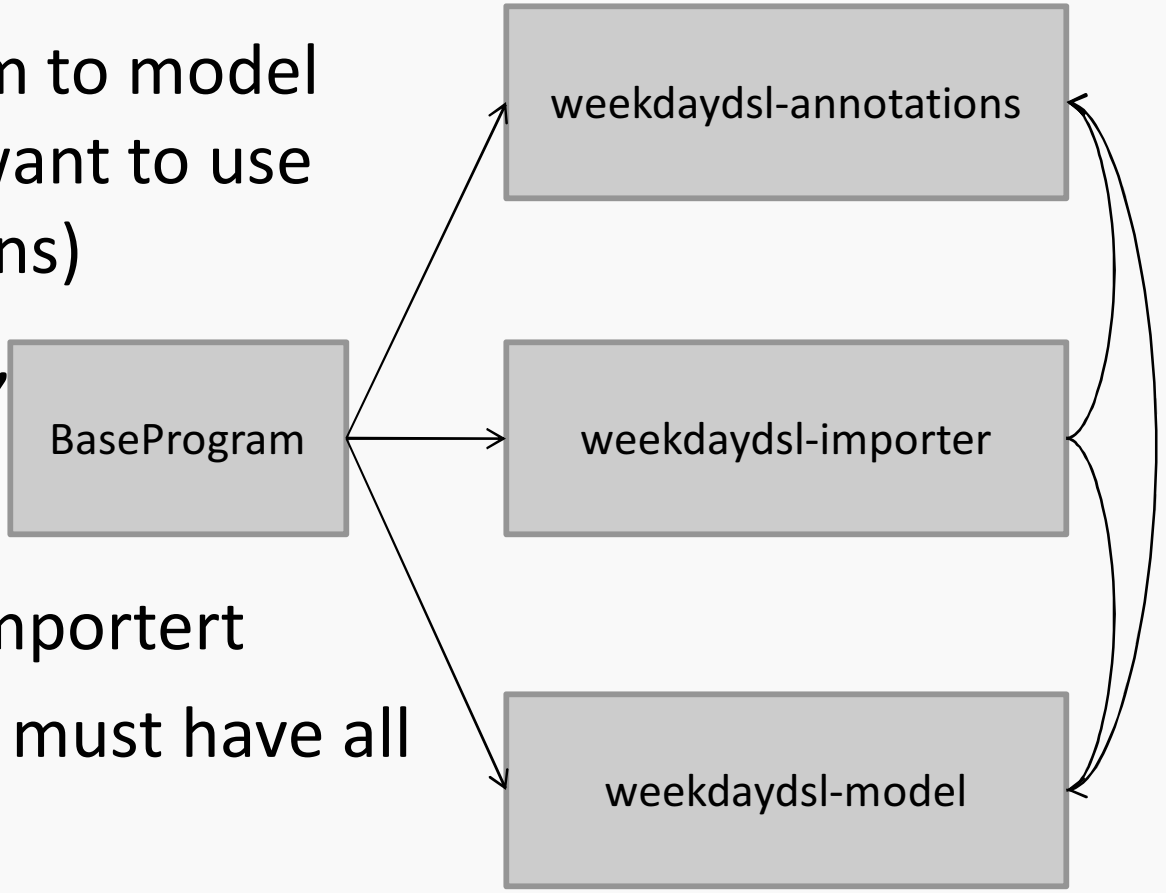
Can context be statically evaluated?

The value type can always be statically approximated. Corresponds to result type of get*Value().

Implement one get*Value method per possible return type.

Dependencies between projects

- Classpath dependencies
  - Add alia4j-liam to model (maybe also want to use alia4j-commons)
  - Add alia4j-fial, -hierarchy, -liam, and -patterns to importert
  - Base program must have all of ALI4J



Optimizations

- Default code generation strategy
  - Try to “computeValueStatically”
  - Success:
    - Inline constant into code
    - For atomic predicates: simplify dispatch function
  - Failure:
    - Generate call to “compute” method
- If entity implements BytecodeSupport (SiRIn and Steamloom$^{ALIA)}$
  - Hand control over bytecode generation to entity
- If entity implements CompilerSupprt (Steamloom^{ALIA})
  - Hand control over machine-code generation to entity
  - Overrides BytecodeSupport
- **Can combine code-generation strategies freely**

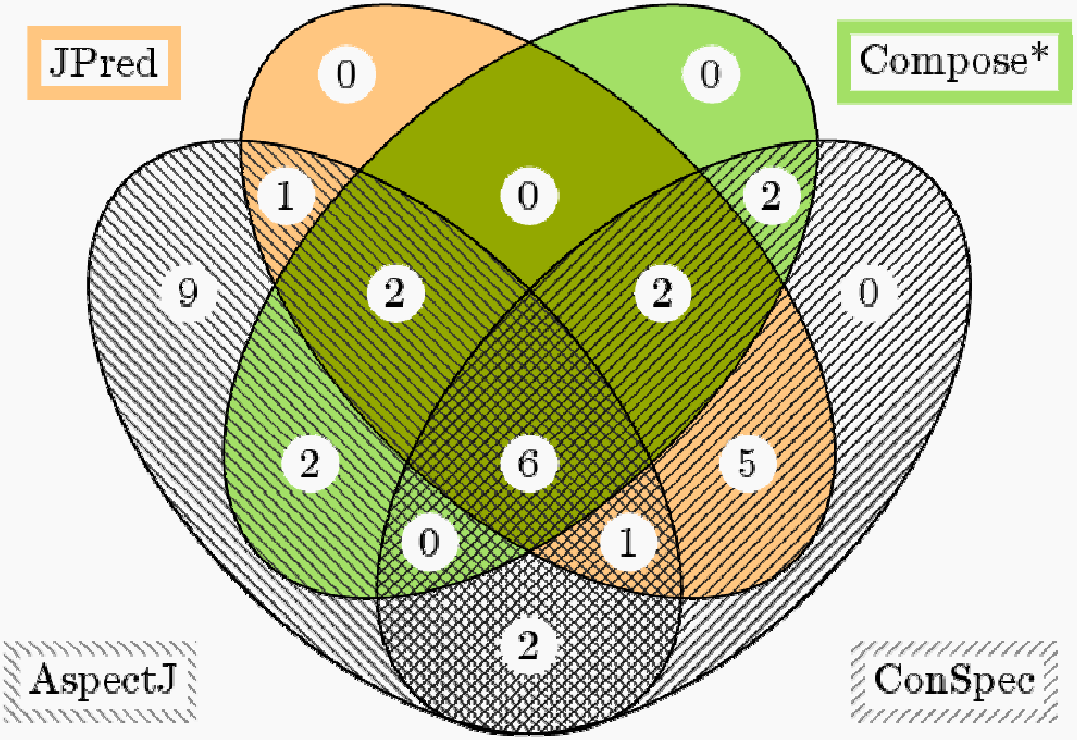
MicroMeasurements

- Measure time for evaluating one entity
- ALIA4J includes MicroMeasurement framework
- Example: Atomic predicate testing whether a context value is in a mapping

Code Generation Strategy	Compiler	
	Baseline	Optimizing
Default	47.54±0.33	23.10±0.17
Bytecode Support	5.11±0.05	3.51±0.06
Compiler-specific IR Support	4.32±0.10	3.49±0.02

Re-use of LIAM entities

- ALIA4J includes implementations of many concepts
  - Most of them implement at least BytecodeSupport
- Case stude: implementd significant parts of Jpred, Compose*, AspectJ, ConSpec with ALIA4J
  - In total 32 concepts are used
  - 23 concepts are used by more than one language



Concise semantics

- Dispatch model together with execution semantics
- What are dispatch sites
  - Method calls
  - Constructor calls
  - Static initializer executions
  - Field read and write access
    - Not for final fields
- When matching against types
  - LIAM defines symbolic type references
  - Including defining class loader
  - Class loaders also contribute to namespace of types!
    - So far only rudimentary support
    - But: forces developer to be aware

Concise semantics

- Do you know your AspectJ?
  - What happens when “object” is used inside “m1”?

```
B object;
m1(object);
```

```
before() : call(* B.m()) {
    System.out.println("before B.m()");
}

before() : execution(* B.m()) {
    System.out.println("before execution B.m()");
}
```

  - Depends on many things:
    - What is the static type of the argument of “m1”?
    - Will “object” be used by means of other static types?
      - Consider class and interface hierarchy.
    - Is the method “m” implemented in “B” or inherited?

Concise Semantics

- Only reliable static information at call site
  - Name of accessed member
  - Type of the field / result type of the method
  - For methods: parameter types
- Use only these in patterns statically evaluated by FIAL
  - Other parts of patterns should go to atomic predicates:
    - Declaring type, thrown exceptions, modifiers
  - Can choose explicitly to evaluate patterns on
    - Resolved, actually executed method
    - Dynamic receiver type
    - Top-most declaration of method in class or interface hierarchy
  - Concept: **match on Events rather than on Methods**
  - Work in progress

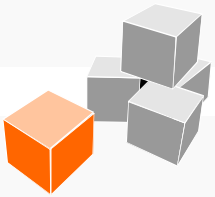
Current and Future Work

- Master projects
  - Andre Loker:
    - Implement generic Context for per-objects instantiation strategies
    - Comprise singleton, perTarget, perThis, associtation aspects, etc.
    - Implement adaptive optimization
  - Remko Bijker:
    - Implement optimization for patter evaluation
    - Apply strategies for optimizing evaluation strategy
    - Survey common structure of signatures and patterns to find and exploit commonalities
  - Haihan Yin:
    - Implement dynamic debugging support for ALIA4J execution environments
    - Language-independent debugging support
    - Currently focus on Eclipse IDE and NOIRIn execution environment

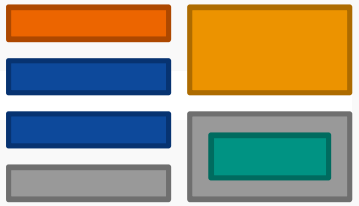
Current and Future Work

- Steven te Brinke:
  - Implement Co-op based on ALIA4J
  - Re-design Co-op language
  - Make it more declarative
  - Map Co-op to ALIA4J concepts, extend FIAL with more advanced constraint resolution
- Future Work
  - Compare ALIA4J approach with SOOT

Twente Research and Education on Software Engineering, Universiteit Twente



Software Technology Group
TU Darmstadt | FB Informatik



www.alia4j.org

http://www.alia4j.org

CASED