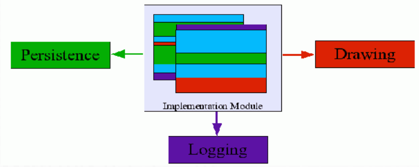
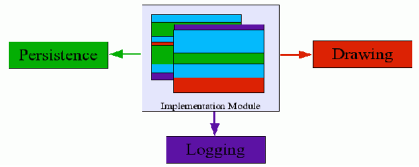
Aspects

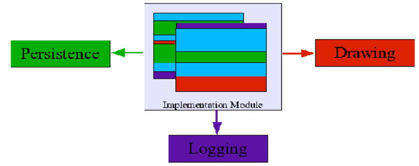


- Problem of Crosscutting Concerns
- Solution with Aspects
- Aspect Oriented languages and tools
- Development and Production Aspects for HEP

What's Wrong ?



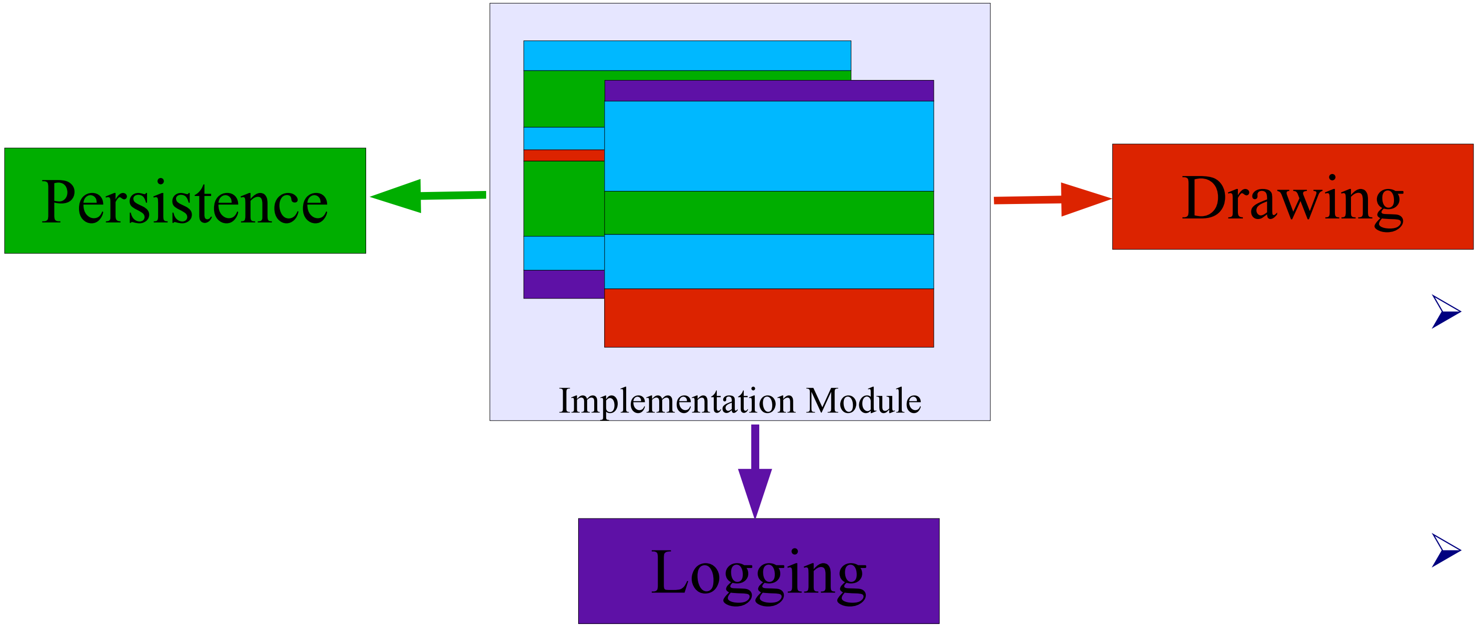
- In OOP, Object is the only fundamental abstraction. In real life, however, other abstractions are needed, e.g.:
  - Before-after
  - Cause-effect
  - State
- In OOP, Hierarchies (is_a) and Collections (has_a) are the only relations. In real life, however, other relations are needed, e.g.:
  - Master-slave
  - NxM
  - Component-container
  - Interval
  - Element-metadata
- OOP solves this by work-arounds (Patterns, Hooks, Wrappers,...).
- Can Aspects be the first step of a more organic solution ?



Crosscutting Concerns

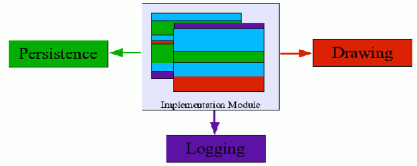
```
class Track implements Drawable, PersistenceCapable {
  reconstruct() {
    log.info("Starting reconstruction ...");
    ...
    }
  draw() {...}
  write(Writer outstream) {...}
  static read(Reader instream) {...}
  static Logger log = ...;
  }
```

```
class Hit implements Drawable, PersistenceCapable {
  ...
  }
```



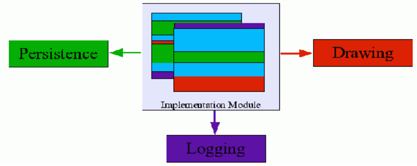
- Besides its own Mission, classes have to fulfill other (unrelated) tasks:
  - Logging/Tracing
  - Authentication
  - Persistency
  - Exception handling
  - Contract Enforcing
  - Distribution
  - Self-testing
  - ...
- Those tasks are spread over classes from different domains.
- OOP doesn't give tools to modularize them.

Problem



- Crosscutting Concerns have serious impact on source code:
  - Code Tangling
  - Code Scattering:
    - Duplicated Code
    - Complementary Code
- With consequences on software quality:
  - Poor Traceability
  - Low Reuse
  - Hard Evolution
- Traditional OOP (abstract interfaces,...) can't modularize Crosscutting Concerns:
  - Using interfaces, implementation should be defined for each class.
  - Interface can't define which classes it should act on.
  - Hooks (Publish/Subscribe, Visitor,...) must be placed before affected class.
  - Wrappers can be circumvented.

AOP

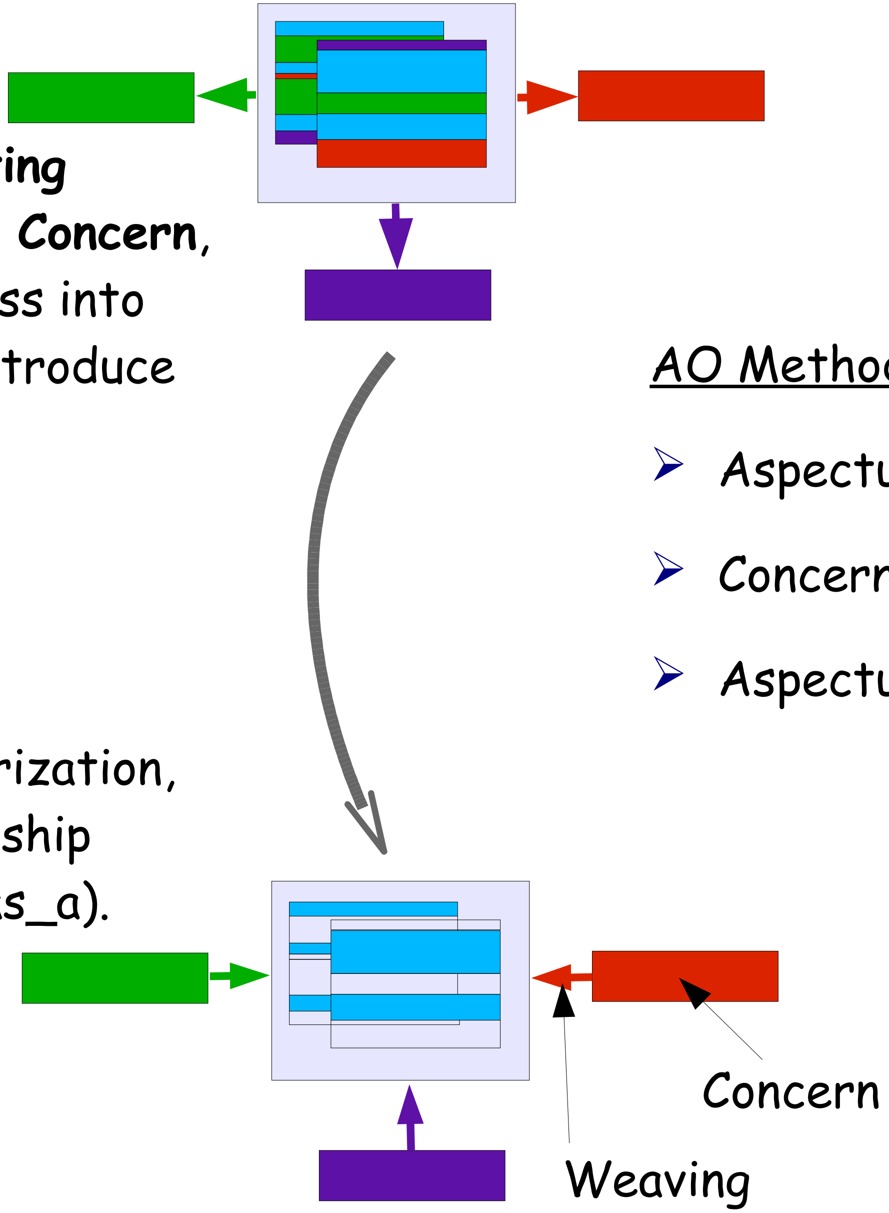


Lets separate **Crosscutting Concerns** from the **Core Concern**, move them from the Class into other entities, and re-introduce them later.

Lets call them **Aspects**.

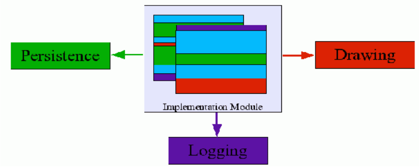
We have introduced:

- ➢ new level of Modularization,
- ➢ new kind of Relationship (besides is_a and has_a).



AO Methodology:

- ➢ Aspectual Decomposition
- ➢ Concern Implementation
- ➢ Aspectual Recomposition



Aspect

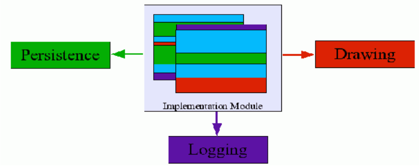
- **Join Point** (identifiable point, formally described by PointCut):
  - Method – call, execution
  - Constructor – call, execution
  - Field Access – read, write
  - Exception
  - Initialization – class, object, object pre-initialisation
  - Advice Execution
- **Advice** (code to be executed at Joint Point):
  - Before
  - After – returning, throwing, always
  - Around
- **Introduction** (modification of code)
- **Compile-time Declaration**
  - Warning
  - Error

<u>Aspect can</u>

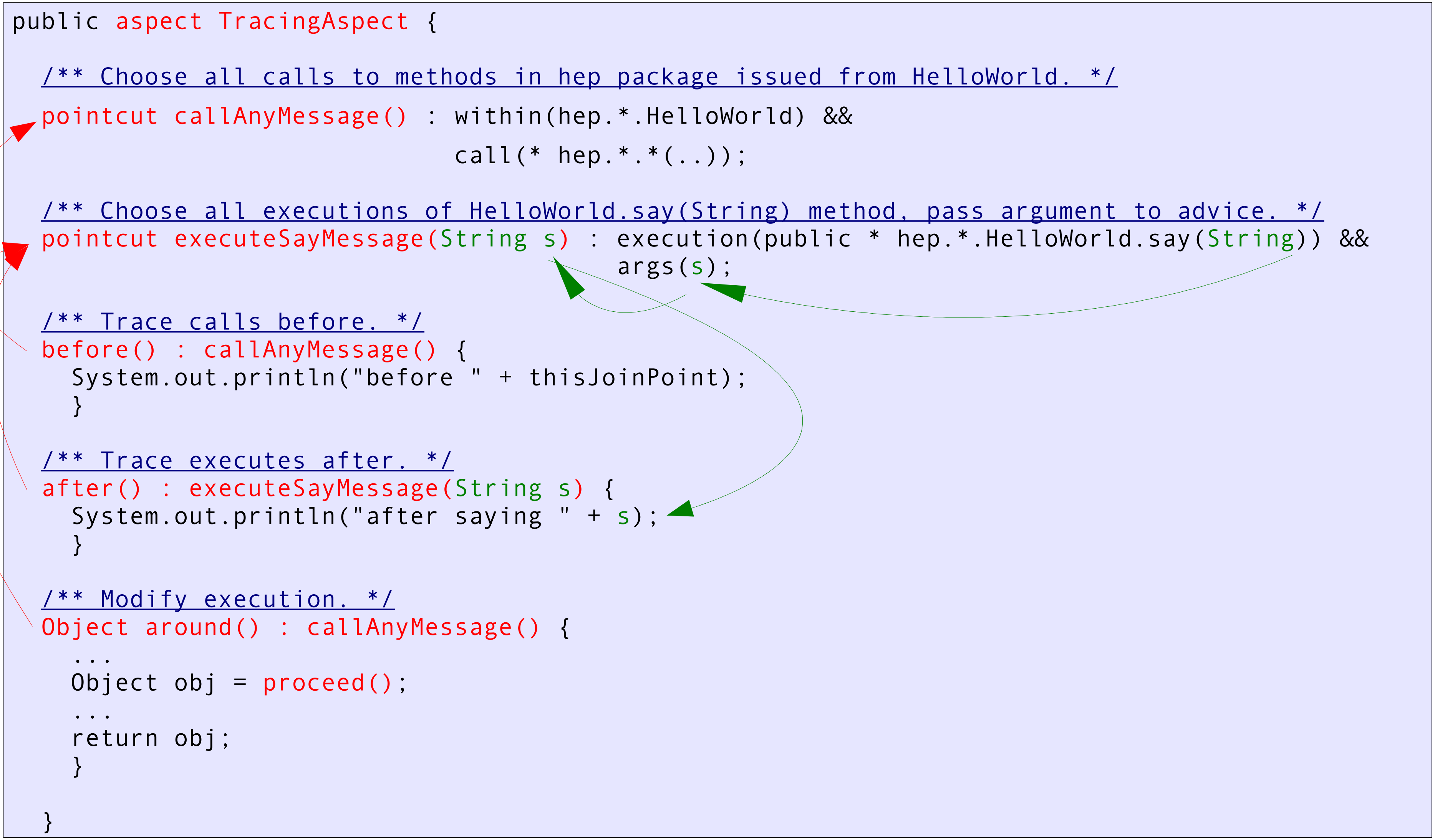
- extend class
- implement interface
- extend another aspect
- contain methods and data

<u>Analogy with OOP:</u>

- Aspect = Class
- Pointcut = Method Declaration
- Advice = Method Implementation



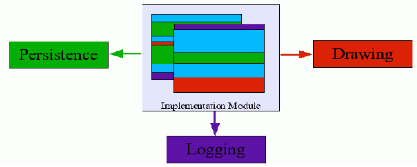
AspectJ Example



```
public aspect TracingAspect {

  /** Choose all calls to methods in hep package issued from HelloWorld. */
  pointcut callAnyMessage() : within(hep.*.HelloWorld) &&
                              call(* hep.*.*(..));

  /** Choose all executions of HelloWorld.say(String) method, pass argument to advice. */
  pointcut executeSayMessage(String s) : execution(public * hep.*.HelloWorld.say(String)) &&
                                         args(s);

  /** Trace calls before. */
  before() : callAnyMessage() {
    System.out.println("before " + thisJoinPoint);
    }

  /** Trace executes after. */
  after() : executeSayMessage(String s) {
    System.out.println("after saying " + s);
    }

  /** Modify execution. */
  Object around() : callAnyMessage() {
    ...
    Object obj = proceed();
    ...
    return obj;
    }

  }
```

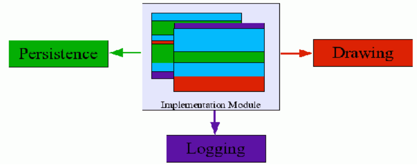
- AspectJ uses extensions to Java – Aspects have to be compiled by special tools.
- Weaving rules are defined inside Aspects.

AspectJ Example (cont.)



```
/** Modify class hierarchy, declare Track PersistenceCapable. */
declare parents : Track implements PersistenceCapable;

/** Add Vertex to Track. */
private Vertex Track._vertex;

/** Issue error, if user wants to create Track directly instead of by TrackFactory subclass. */
declare Error : call(Track.new(..)) &&
                ! within(TrackFactory+): “Only TrackFactory can create Tracks !”;
```

AspectWerkz Syntax



```
package TestAOP;

import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspect {

  public void beforeGreeting(JoinPoint joinPoint) {
    System.out.println("before greeting...");
    }

  public void afterGreeting(JoinPoint joinPoint) {
    System.out.println("after greeting...");
    }

  @Around(“greetMethod”)
  public Object aroundGreeting(JoinPoint joinPoint) {
    Object greeting = joinPoint.proceed();
    return "<yell>" + greeting + "</yell>";
    }

  }
```

Aspect

Java 1.5 Annotation (optional)

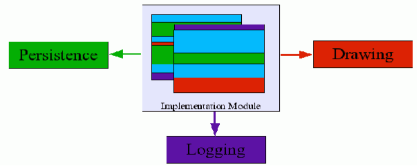
Differences to AspectJ:

- Aspect is normal Java class so it can be compiled by standard compiler and distributed as standard jar library.
- Weaving Rules can be external (in XML) so it can be applied independently, later.
- Weaving Rules can be expressed using Java 1.5 Annotations.

```
<aspectwerkz>
    <system id="AspectWerkzExample">
        <package name="TestAOP">
            <aspect class="MyAspect">
                <pointcut name="greetMethod" expression="execution(* *.greet(..))"/>
                <advice name="beforeGreeting" type="before" bind-to="greetMethod"/>
                <advice name="afterGreeting"  type="after"  bind-to="greetMethod"/>
                <advice name="aroundGreeting" type="around" bind-to="greetMethod"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Weaving Rules

Syntax and Languages

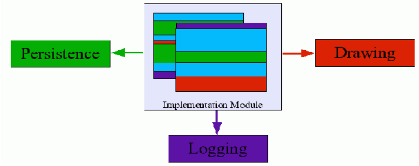


- **Constructs:**
  - Pointcut
  - Advice
  - Weaving instructions
- **Language:**
  - Target language
  - Extension of Target language
  - XML
  - (Embedded) Annotations
  - Special language
  - Framework/GUI
- **Composition:**
  - All in the same unit
  - Different units for different Constructs

Level of support varies

Java generally well supported

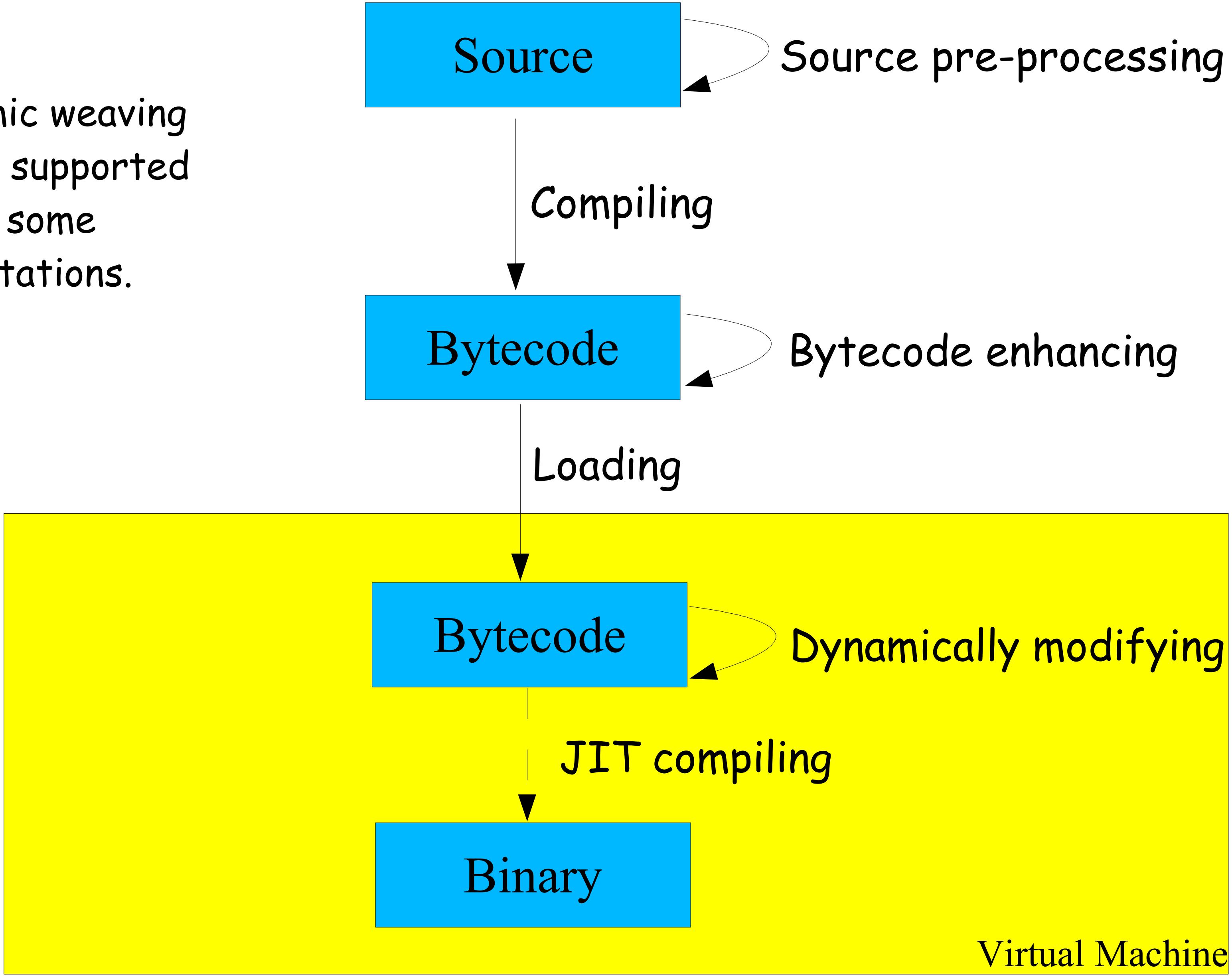
- **Java (195k GoogleMarks):**
  - **AspectJ** (125k)
  - AspectWerkz (40k)
  - Java Aspect Components (20k)
  - JBoss AOP (10k)
- **C/C++ (4k):**
  - AspectC (2k)
  - AspectC++ (2k)
- **Others:**
  - Python – Pythius, Pythonic (0.5k)
  - Perl - Aspect
  - Ruby – AspectR (3k)
  - C# - AspectC# (2k)
  - Lisp – itself



Weaving

Introducing Aspects into code.

More dynamic weaving methods are supported only by some implementations.



Source

Source pre-processing

Compiling

Bytecode

Bytecode enhancing

Loading

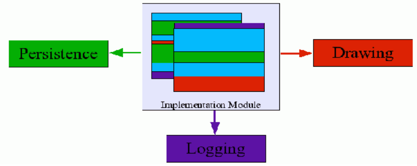
Bytecode

Dynamically modifying

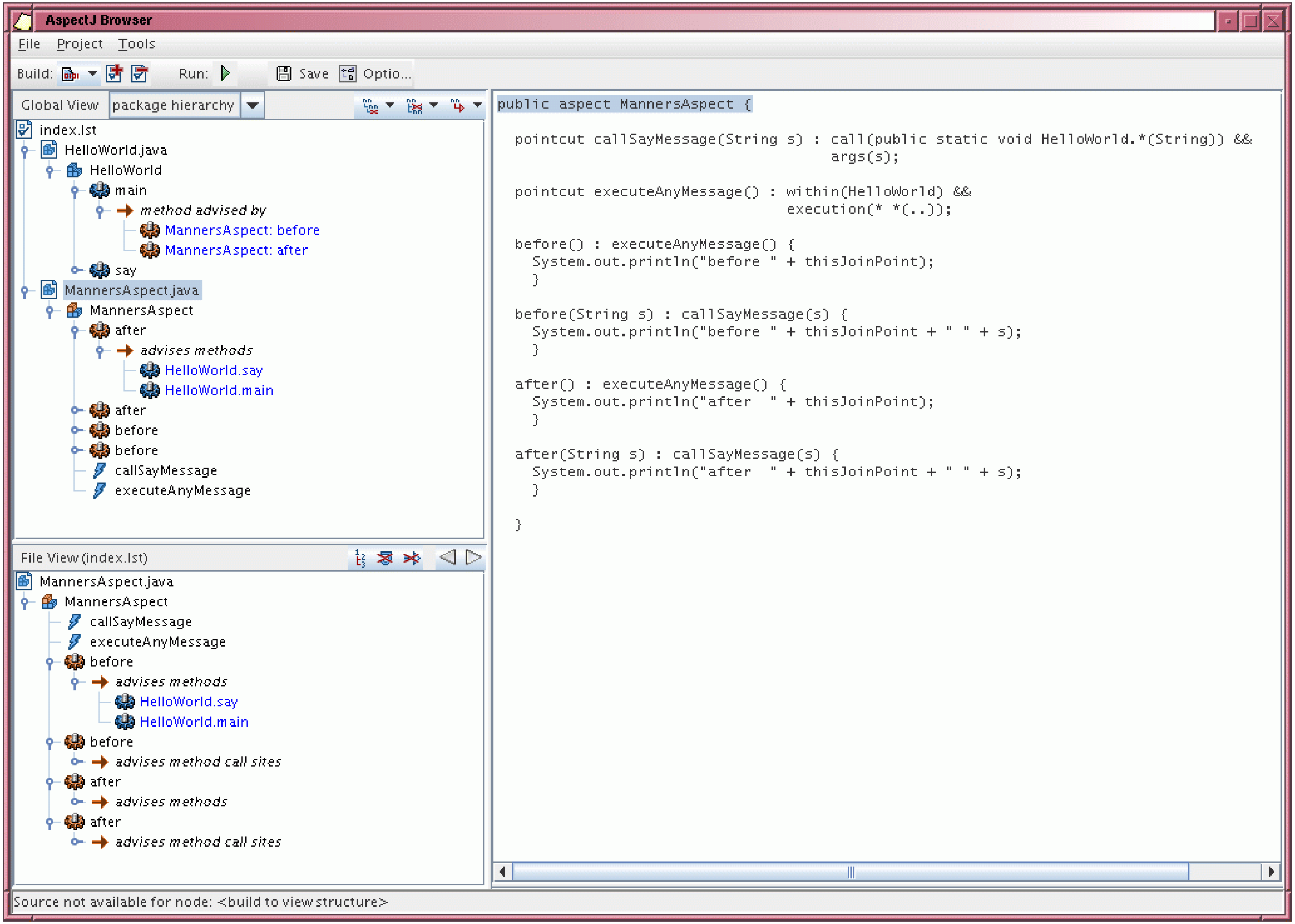
JIT compiling

Binary

Virtual Machine



GUI



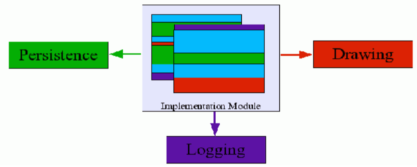
```
index.lst
  HelloWorld.java
    HelloWorld
      main
        method advised by
          MannersAspect: before
          MannersAspect: after
      say
  MannersAspect.java
    MannersAspect
      after
        advises methods
          HelloWorld.say
          HelloWorld.main
      after
      before
      before
      callSayMessage
      executeAnyMessage
```

```
MannersAspect.java
  MannersAspect
    callSayMessage
    executeAnyMessage
    before
      advises methods
        HelloWorld.say
        HelloWorld.main
    before
      advises method call sites
    after
      advises methods
    after
      advises method call sites
```

```
public aspect MannersAspect {

  pointcut callSayMessage(String s) : call(public static void HelloWorld.*(String)) &&
                                      args(s);

  pointcut executeAnyMessage() : within(HelloWorld) &&
                                 execution(* *(..));

  before() : executeAnyMessage() {
    System.out.println("before " + thisJoinPoint);
    }

  before(String s) : callSayMessage(s) {
    System.out.println("before " + thisJoinPoint + " " + s);
    }

  after() : executeAnyMessage() {
    System.out.println("after  " + thisJoinPoint);
    }

  after(String s) : callSayMessage(s) {
    System.out.println("after  " + thisJoinPoint + " " + s);
    }

  }
```

Source not available for node: <build to view structure>

Incremental compilation is supported.

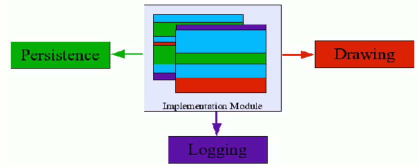


Applicability

- Logging/Tracing
- Exception Handling
- Monitoring/Profiling
- Unit testing
- Const/Final
- Cache Management
- Connection Pool
- Contracts Enforcing
- Security/Authentication/Authorization
- Distribution
- Grid
- Coding Conventions Checking
- Web Service
- Graphics
- Multiple Inheritance
- Mixin
- Persistence
- Fine-grained Access
- Patterns
  (Patterns correct problems in OO languages. In AOP, some Patterns disappear.):
  - Factory
  - Observer (disappears)
  - Visitor (disappears)
  - MVC
  - Entity-Model-Representation

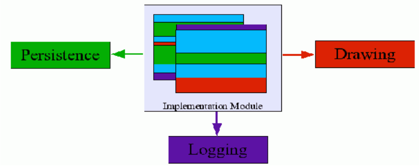
Application Examples

(Development Aspects)

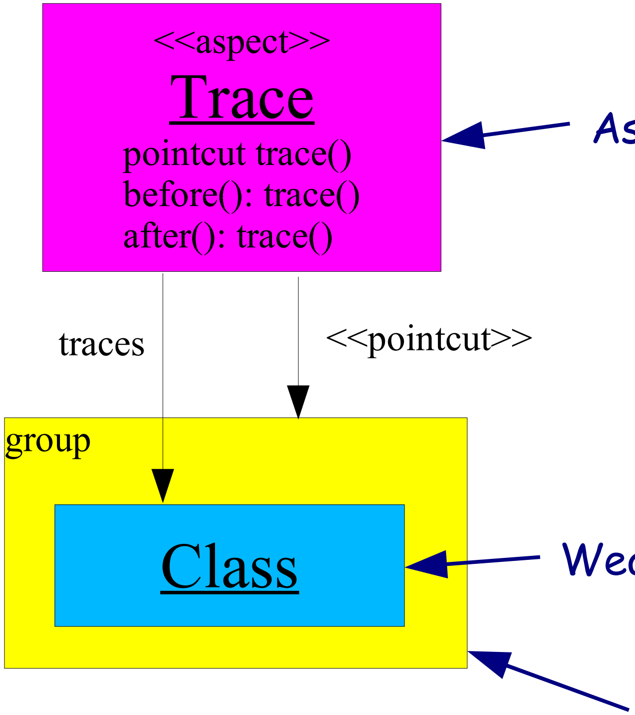


Complete Examples

- **Tracer:**
  - Traces program control flow
- **ContractManager:**
  - Enforces preconditions, postconditions and invariants



Tracer



Aspect

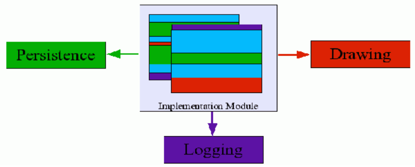
Weaved Class

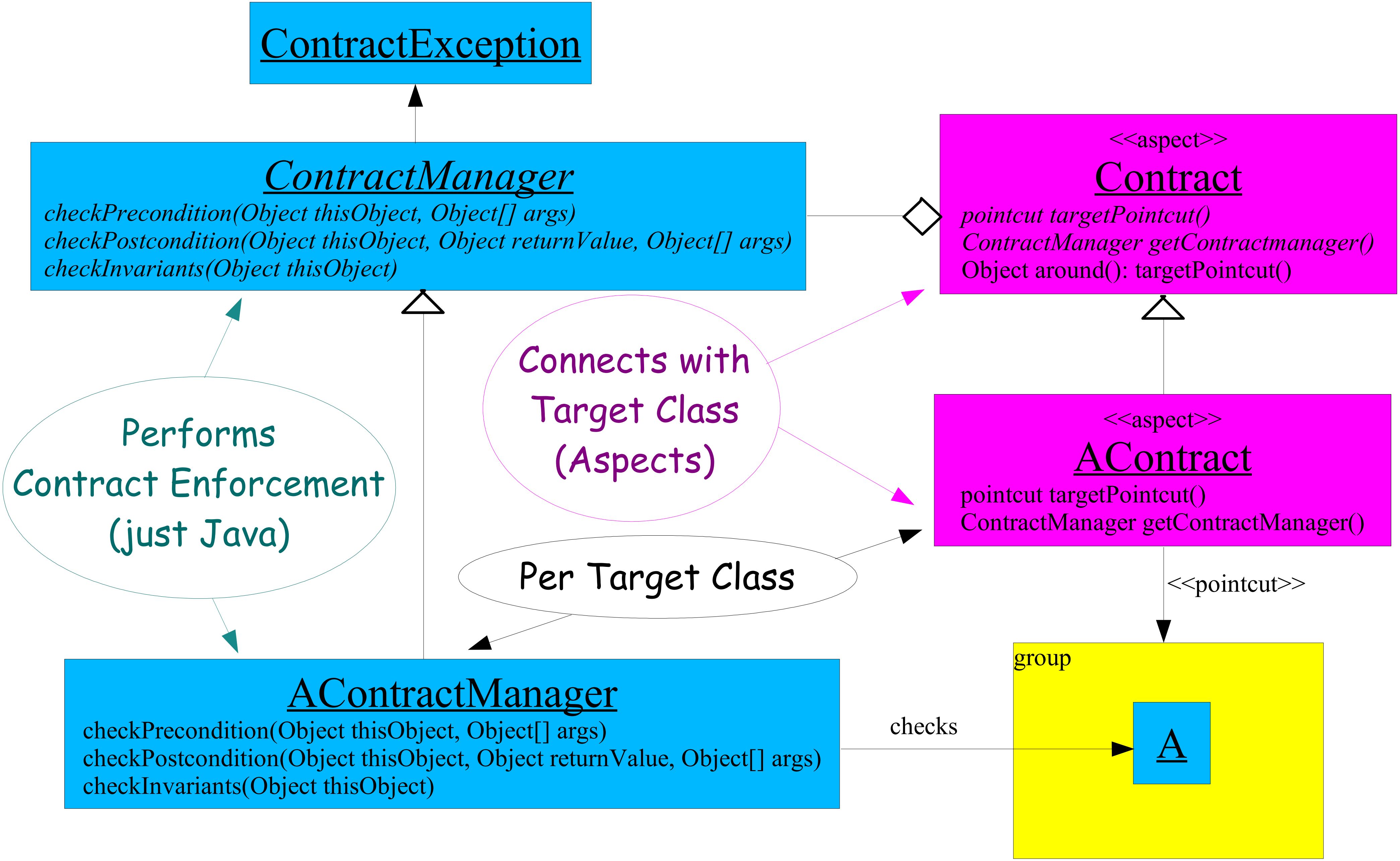
Group of Classes selected by Pointcut

```
package net.hep.aspects.AspectJ;

public aspect Trace {

  /** Trace all executions, except itself. */
  pointcut trace() : execution(* *.*(..)) &&
                    ! within(net.hep.aspects.AspectJ.*);

  /** Widen depth, adjust prefix, write out where we are. */
  before() : trace() {
    callDepth += 2;
    prefix = BLANKS.substring(0, callDepth);
    Object[] args = thisJoinPoint.getArgs();
    System.out.println(prefix + thisJoinPoint.getSignature());
    for (int i = 0; i < args.length; i++) {
      System.out.println(prefix + ">   " + args[i]);
      }
    }

  /** Shorten depth. */
  after() : trace() {
    callDepth -= 2;
    }

  private static int callDepth = -1;

  private static String BLANKS = "                          ";

  private static String prefix;

  }
```

If you already know AOP,
you have seen this example
many times. Sorry.

Contract Enforcement





ContractException

ContractManager

checkPrecondition(Object thisObject, Object[] args)
checkPostcondition(Object thisObject, Object returnValue, Object[] args)
checkInvariants(Object thisObject)

<<aspect>>
Contract

pointcut targetPointcut()
ContractManager getContractmanager()
Object around(): targetPointcut()

Connects with Target Class (Aspects)

Performs Contract Enforcement (just Java)

<<aspect>>
AContract

pointcut targetPointcut()
ContractManager getContractManager()

Per Target Class

<<pointcut>>

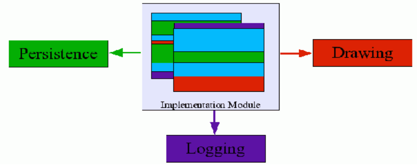
AContractManager

checkPrecondition(Object thisObject, Object[] args)
checkPostcondition(Object thisObject, Object returnValue, Object[] args)
checkInvariants(Object thisObject)

checks

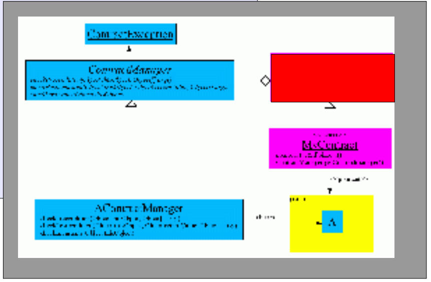
group

A

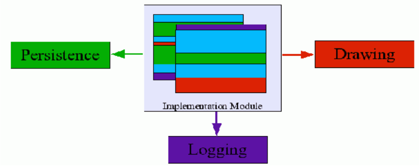


Contract

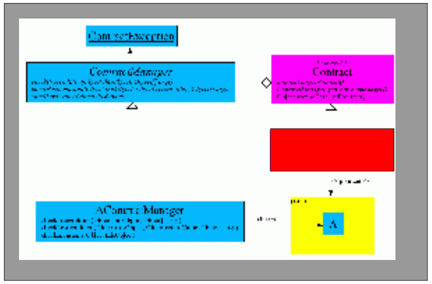
```
/** Contract Manager interface checking preconditions, postconditions and
  * invaiants.
public abstract aspect Contract {

  /** Define the pointcut to apply the contract checking. */
  public abstract pointcut targetPointcut();

  /** Define the ContractManager interface implementor to be used. */
  public abstract ContractManager getContractManager();

  /** Perform the logic necessary to perform contract checking. */
  Object around(): targetPointcut() {
    ContractManager cManager = getContractManager();
    Object obj = null;
    if (cManager != null) {
      System.out.println("... Checking contract using: " + cManager.getClass().getName());
      System.out.println("... Performing initial invariants check");
      cManager.checkInvariants(thisJoinPoint.getTarget());
      System.out.println("... Performing pre-conditions check");
      cManager.checkPreConditions(thisJoinPoint.getTarget(), thisJoinPoint.getArgs());
      obj = proceed();
      System.out.println("... Performing post conditions check");
      cManager.checkPostConditions(thisJoinPoint.getTarget(), obj, thisJoinPoint.getArgs());
      System.out.println("... Performing final invariants check");
      cManager.checkInvariants(thisJoinPoint.getTarget());
      }
    else {
      System.out.println("... No ContractManager found");
      obj = proceed();
      }
     return obj;
     }

  }
```

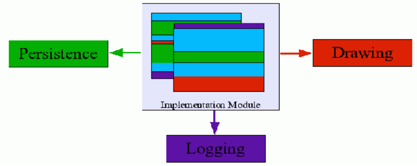


AContract



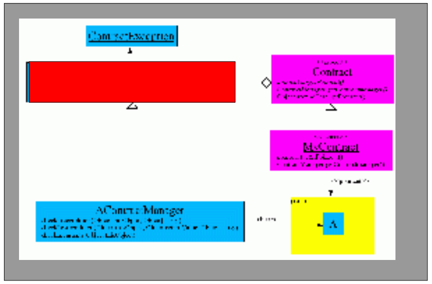
```
/** AContract extends abstract AContract aspect for use
  * to check A class. */
public aspect AContract extends Contract {

  /** Check A.greet(..) method. */
  public pointcut targetPointcut(): call(String A.greet(..));

  public ContractManager getContractManager() {
    return new AContractManager();
    }

  }
```

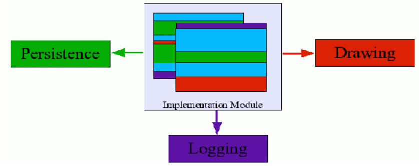




ContractManager

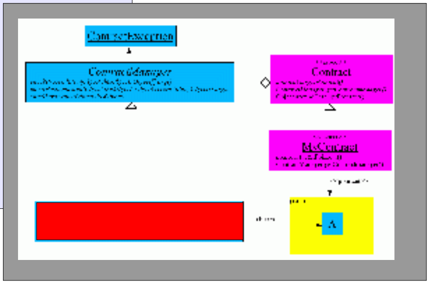
```
/** Contract Manager interface checking preconditions, postconditions and
  * invariants. */
public interface ContractManager {

  /** Check the preconditions. */
  public void checkPreConditions(Object thisObject,
                                 Object[] args) throws ContractException;

  /** Check the postconditions. */
  public void checkPostConditions(Object thisObject,
                                  Object returnValue,
                                  Object[] args) throws ContractException;

  /** Check the invariants */
  public void checkInvariants(Object thisObject) throws ContractException;

  }
```



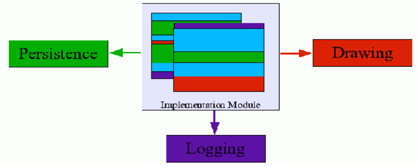


AContractManager

```
/** AContract Manager implements ContractManager for use
  * to check A class. */
public class AContractManager implements ContractManager {

  /** Check whether argument is not null. */
  public void checkPreConditions(Object thisObject,
                               Object[] args) throws ContractException {
    Object arg = args[0];
    if (arg == null) {
      throw new ContractException("\n*** Precondition Violated: " +
                                  "Argument shouldn't be null !");
      }
    }

  /** Check whether return value is not null. */
  public void checkPostConditions(Object thisObject,
                                Object value,
                                Object[] args) throws ContractException {
    if (value == null) {
      throw new ContractException("\n*** Postcodition Violated: " +
                                  "Return value shouldn't be null !");
      }
    }

  /** Check whether n is not negative. */
  public void checkInvariants(Object thisObject) throws ContractException {
    if (thisObject instanceof A &&
      ((A)thisObject).n < 0) {
      throw new ContractException("\n*** Invariant Violated: " +
                                  "n shouldn't be negative !");
      }
    }

  }
```

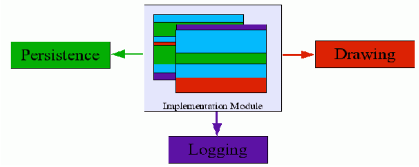


Other Examples



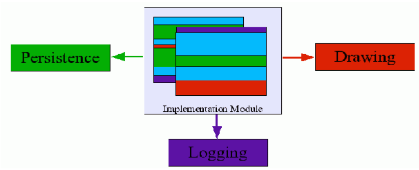
- Graphics:
  - Aspect uses core class and performs all graphical actions for it (prototyped to connect GraXML display (4.x.x) to external framework)
- Fine-Grained Access Control:
  - Aspects checks that only allowed relations are used
- Cache:
  - Around advice stores all results in a cache
  - Cached result is returned if it exists
- Connection Pool: analogical to Cache
- Web Service:
  - Aspect wraps serving class in a Web Service
  - Around advice forwards service request through Web Service

Other Examples (cont.)



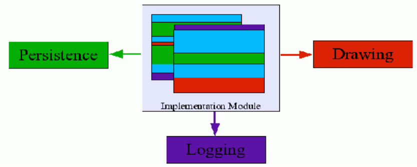
- Persistence:
  - Aspect introduces read/write functions
  - Field access advice performs reading/writing when necessary
  - Aspect makes class (JDO) PersistenceCapable (used in JOnAS Speedo)
  - (JDO) PersistenceCapable Aspect connects to a core class and handles its persistence (prototyped for AIDA FreeHEP)
- Factory:
  - Around advice returns unique Object on all Constructor calls
  - Compile-time declaration checks that objects are not created directly
- Singleton:
  - Around advice on Constructors returns single Objects, if it already exists; creates it otherwise

Devil's Arguments



- Immaturity:
  - Aspect syntax is not standardized, there are several incompatible approaches.
  - AOP Theoretical Foundation is not yet very solid.
  - AOP Methodology is still very primitive. UML syntax for Aspects is not yet standardized
  - Pointcuts rely on naming conventions, they use just (a bit better) regular expressions and pattern matching (with weak grammar).
- Fundamental problems:
  - AOP breaks encapsulation. (Yes, but in a controlled way. Otherwise, equivalent functionality would require more serious break.)
  - AOP improves locality of Concerns, but destroys locality of Control Flow. Control Flow of program with Aspects is difficult to understand. Tools are necessary. (But that is true for Object Oriented Program compared with Procedural Program too.)
  - Aspects can change program behavior without original author being aware (and what about copyright ?). (But this is what we want.)
  - AOP programs can be hard to evolve as they rely on (coding) conventions. Objects depend on Aspects, but Aspects depend on Objects' structure. (This is not much more serious that pre-AOP dependencies in OOP.)

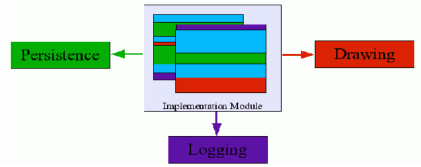
HEPAspects



http://home.cern.ch/hrivnac/Activities/Packages/HEPAspects

- Reusable Aspects (incl. Examples from this talk).
- Growing.
- Contains Ant tasks for AspectJ management.
- Naming convention:
  - <library>.jar – core/naked library
  - <library>.Aspects.jar – Aspects library (i.e. Aspects + supporting classes)
  - <library>.Weaved.jar – weaved library
- HEPAspects contains:
  - HEPAspects.Aspects.jar – Aspects library
  - HEPAspects.jar- testing library
- **HEPAspects/bin/weave.sh <mylibrary>.jar** applies HEPAspects.Aspects.jar and creates **<mylibrary>.Weaved.jar**

Summary



- Object Oriented Programming abstractions are not rich enough to capture actual Use Cases.
  - In particular, **Crosscutting Concerns** can't be expressed.
- Various ways have been created to fix that problem (OO Patterns, etc.).
  - Those solution are too complex and fragile as they are not naitive to existing (OO) languages (Abstraction Leak).
- **Aspect Oriented Programming** offers organic way of modularizing Crosscutting Concerns.
- There are several fully functional AOP systems, the most popular is AspectJ.
- Many HEP Crosscutting Concerns can be easily separated with AOP.
- <u>AOP (in Java) is ready for Development and optional Production Aspects.</u>
- Aspects mentioned this week also in talks about Alice and Atlas frameworks.

AOP (Java) is
- Solid
- Easy to use
- Powerfull (maybe too)

However
- In rapid evolution
- With unclear impact on Architecture