



JAML

An Extensible Framework for Developing Aspect-Specific Languages

Crista V Lopes and Trung Ngo

University of California, Irvine

General Purpose vs. Domain Specificity

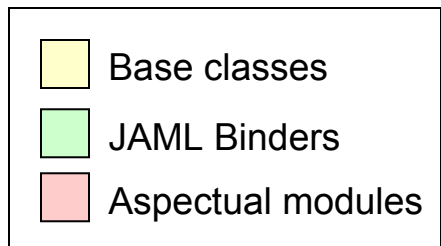
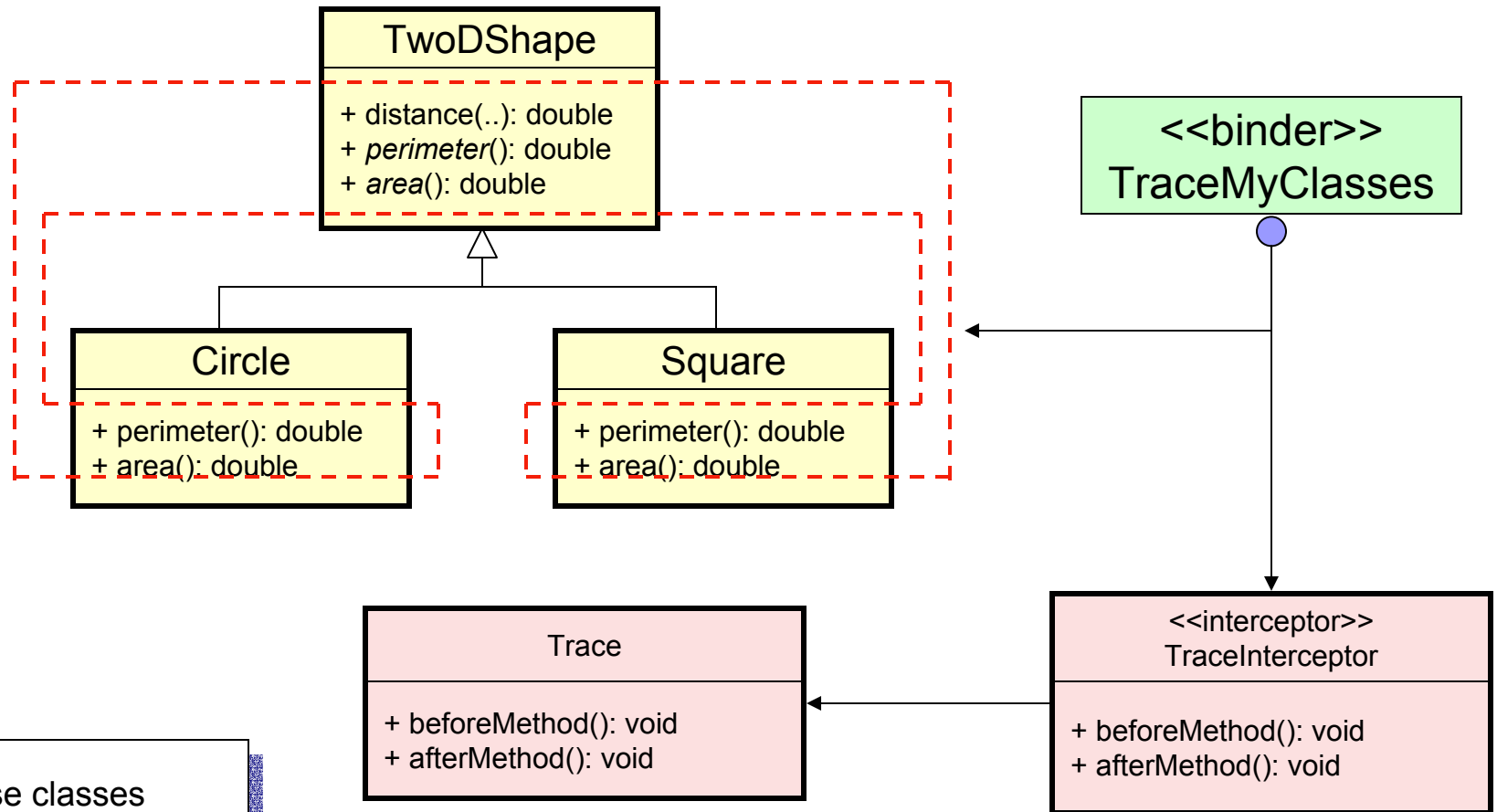
- Domain-specific languages
 - Nice in terms of the expressive power of programming constructs
 - More *natural* for domain experts
 - Example: LEX, YACC, SQL, HTML,...
- Problems of DSL?
 - Tend to be too big/complex
 - Development of DSL is usually expensive, time-consuming, and not reproducible
 - particularly bad for arbitrary concerns



JAML – Java Aspect Markup Language

- XML-based
 - Separation of binding instructions and aspectual behavior
- Use the ~same~ semantics as AspectJ
- Allows programmers to define new aspect languages via plugins
 - Similar to writing Ant tasks

Example: Tracing aspect



Aspect-Specific Languages (ASLs)

- JAML supports the development of new ASLs
 - Example: ASLs for *tracing*, *GoF design patterns*, *Security*, *etc ...*

ASL = Domain-Specific Language developed on top of a General Purpose Aspect Language



The pre-processing approach

- Plain JAML – an “assembly” language
- Pre-process:
 - Convert aspect-specific constructs to primitive crosscutting constructs
 - Type check, to some extent

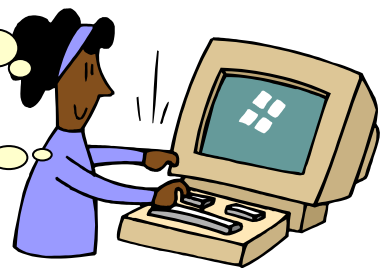
Software Development

Trace classes, methods?

Log files?

Tracing ASL =
2 pointcuts & 4 advices
...

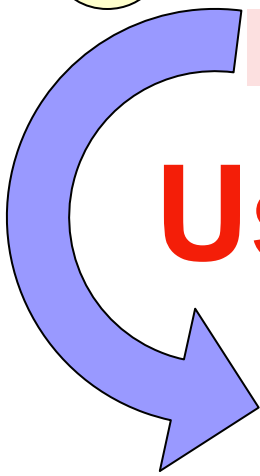
Trace level?



Corporate Programmers

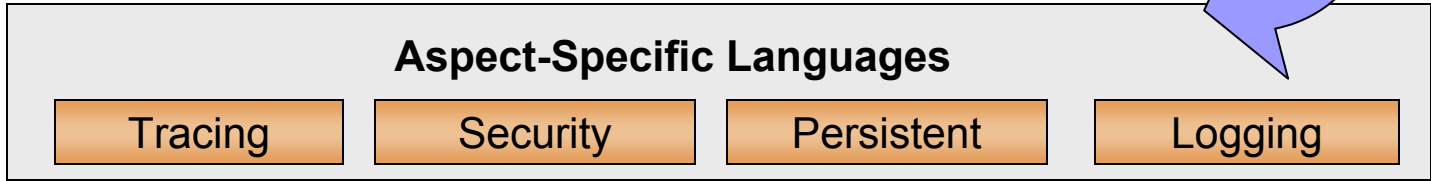


AOP experts



Use

Develop



Plan JAML (General-purpose)

Example: Tracing ASL

Requirement
Trace all methods
of Square and Circle
classes
and log the results
to "trace.log"

Transform

'TraceMyClasses' binder

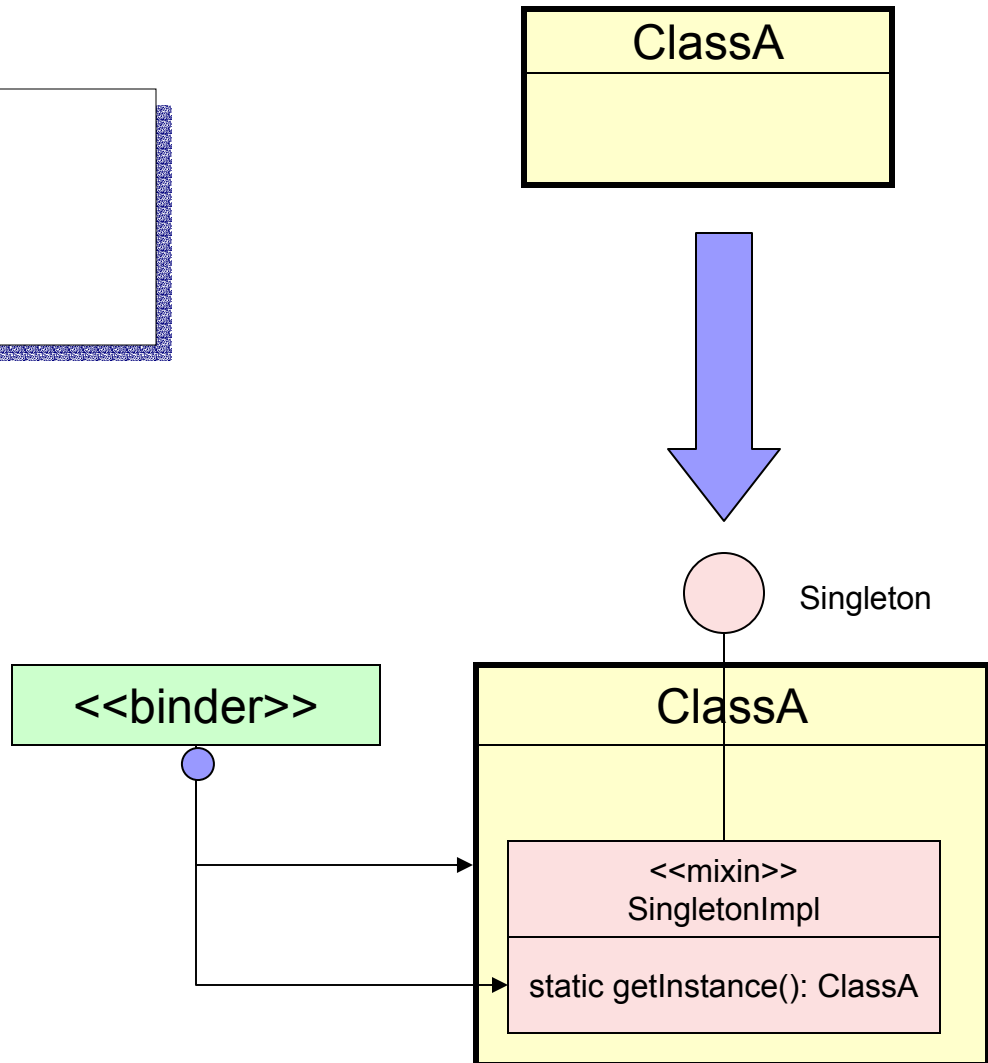
```
<binder id="TraceMyClasses">
  <pointcut id="myMethods">
    <and>
      <pointcut type="method-execution"
        pattern="* Square.*(..)"/>
      <pointcut type="method-execution"
        pattern="* Circle.*(..)"/>
    </and>
  </pointcut>

  <interceptor class="TraceInterceptor">
    <parameters>
      <param id="logFile" value="trace.log"/>
    </parameters>

    <advice type="before" pointcut-refid="myMethods"
      interceptor-method="void beforeMyMethods()"/>
    <advice type="after" pointcut-refid="myMethods"
      interceptor-method="void afterMyMethods()"/>
  </interceptor>
</binder>
```

Example: ASL for GoF design patterns (Singleton)

```
<gof:singletonPattern>  
  <class name="ClassA"  
    excludeSubclasses="yes"/>  
</gof:singletonPattern>
```

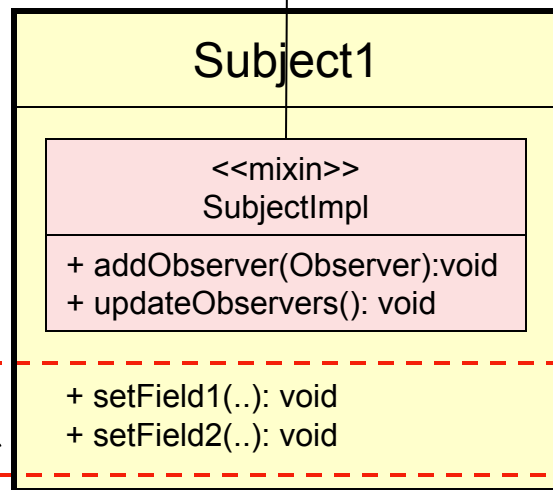
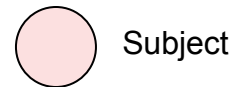
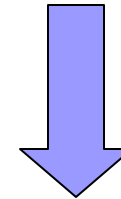
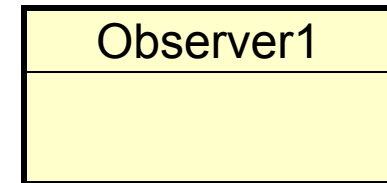
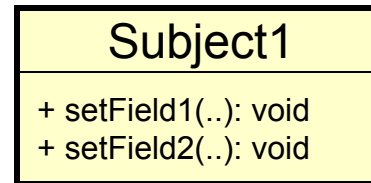


Example: ASL for GoF design patterns (Observer)

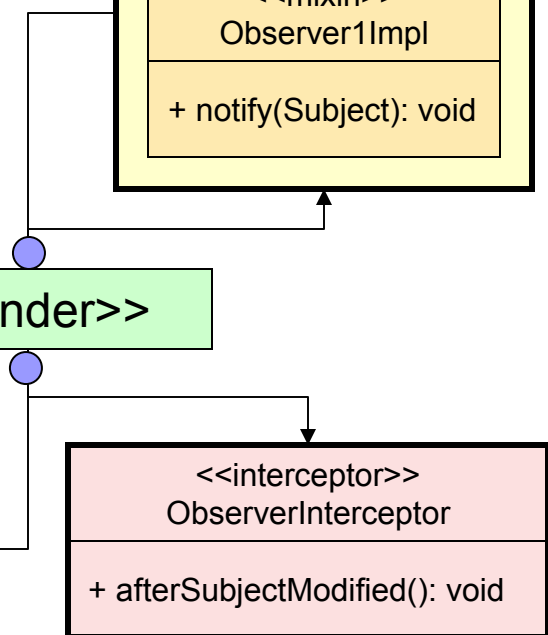
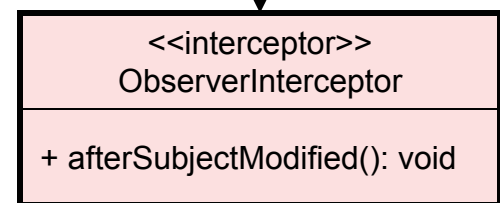
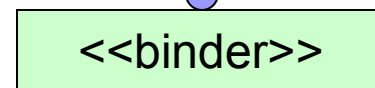
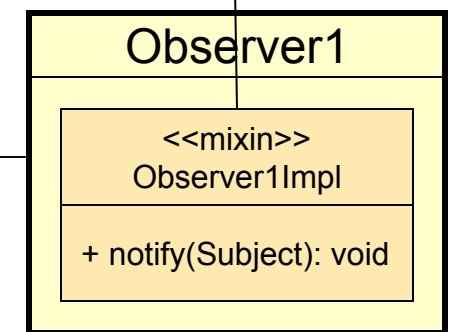
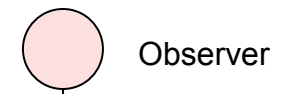
```

<gof:observerPattern>
  <subject classList="Subject1">
    <mutators>
      <method signature="* set*(..)" />
    </mutators>
  </subject>

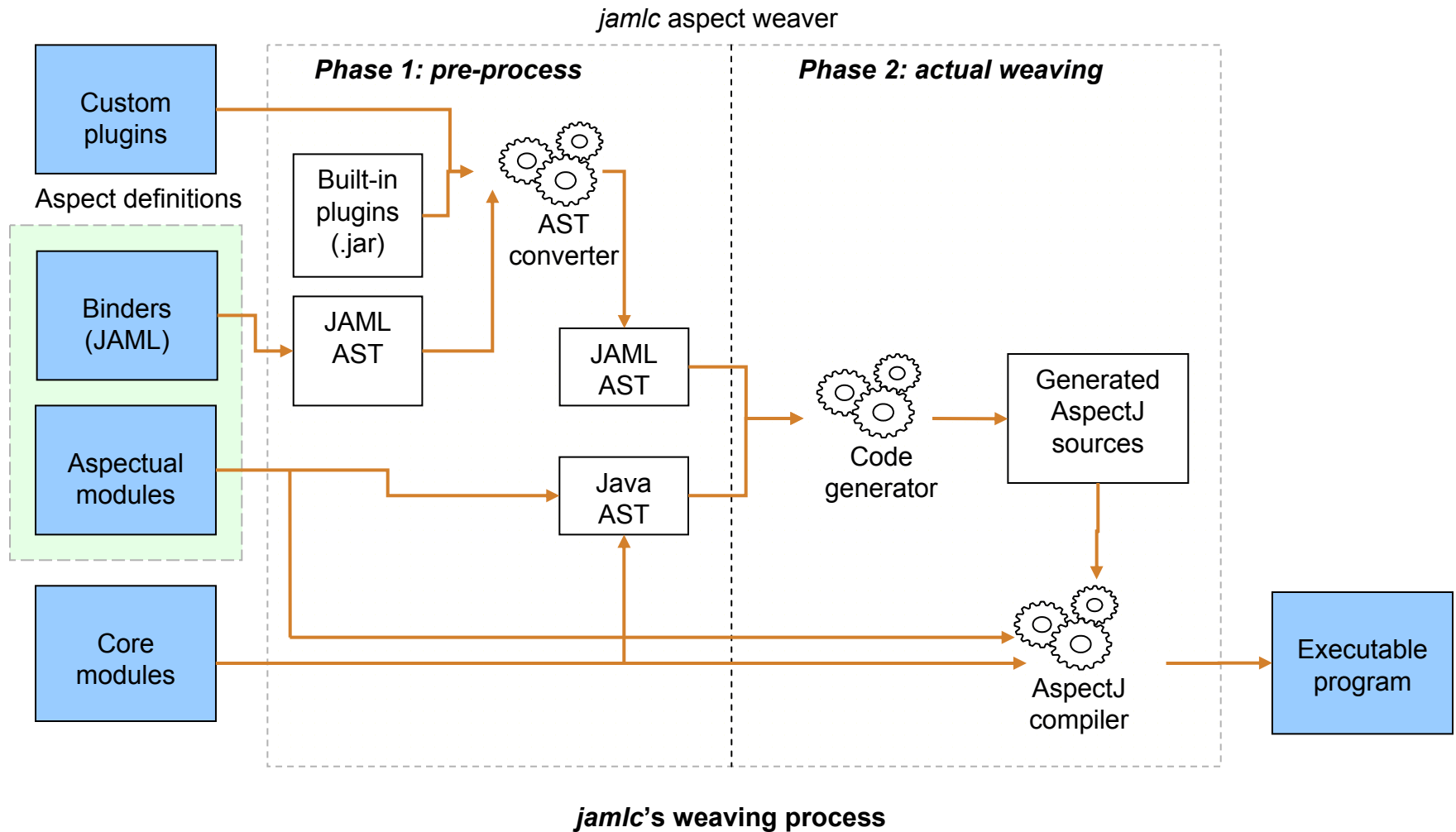
  <observer class="Observer1"
    implementation="Observer1Impl" />
</gof:observerPattern>
  
```



SubjectModified pointcut



Implementation





ASL vs. Abstract Aspect

- Similarity

- Can be used to develop aspect libraries

- Difference

- Abstract Aspect requires deep understanding of the underlying join point model, ASL doesn't

Conclusion

- Programming crosscutting concerns can be done without programmers having to master the underlying AOP model
- Developing aspect-specific languages can be: fast and reproduced from aspect to aspect
- <http://www.ics.uci.edu/~trungcn/jaml/>