Utilizing design information in aspect-oriented programming

István Nagy, Lodewijk Bergmans, Wilke Havinga & Mehmet Aksit

[nagyist,bergmans,havingaw, aksit@cs.utwente.nl]

TRESE group, University of Twente, The Netherlands
Context

- Composition Filters Model
  - Goal: support robust, scalable composition
  - Modularize crosscutting concerns
  - Implemented within Compose*.NET
    - language-independent

- Metadata in OOP languages
  - Custom attributes (.NET)
    - e.g. `[BusinessObject]` class User { .. }
  - Metadata annotations (Java)
    - e.g. `@BusinessObject` public class User { .. }
Motivation

- AOP: pointcuts define locations in the program where the behavior should be enhanced/modified
  - Often specified based on structural/syntactical patterns: execution(* set*(..))
  - Mismatch between design intention and pointcut expression: fragile pointcuts
  - Cause: design information is implicit
    - Lost when mapping design to implementation
Problem statement

- How to represent/access design information (in an AOP approach)
  - Such that pointcuts are more robust

Outline

- Investigate mechanisms for accessing design information (analysis)
- Integration of mechanisms in Compose*
- Conclusion/evaluation
Accessing design information (1): Encoding

Naming patterns

- Design intention can be obtained from identifiers
- Example: a method changes object state
  
  Encoding: public void setName(..)
  Pointcut: execution(public void set*(..));

Problems

- Tight coupling between pointcut and base classes (fragile pointcuts)
- Representing multiple semantic properties
Accessing design information (2): Encoding

Structural patterns

- Design intentions represented by language constructs (e.g. marker interfaces, dummy field,..)

- Example: A class represents a BusinessObject
  Encoding: public interface BusinessObject {}
  public class User implements BusinessObject {...}

- Problem: information permanently attached to units - no late binding
  - class User always a business object?
Accessing design information (3): Attaching

Using annotations

- Design intentions explicitly represented by metadata annotations
- Example: method changes object state

  Encoding: `[Update]` public void setName(..)

- Problems:
  - Late binding (introduction of annotations) not supported by many AOP languages.
  - Annotations are scattered over the program
Accessing design information (4): Inferring

Deriving design properties

- Use automated reasoning to derive design information from common rules
  - Example: When does a method update state?

```
Rule changeState(?class, ?methodName) if
  shadowIn(?class, ?methodName, ?sp),
  assignmentShadow(?sp, ?variable)
```

- Problems
  - Information not always obtainable through (automated) reasoning about syntax/structure
  - Need to specify **domain specific properties**
## Accessing design information: Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Naming Patterns</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Structural Patterns</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Annotations</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Deriving</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

**Motivation -> Problem -> Analysis -> Realization -> Conclusion**
Analysis results

- Annotations + derivation are a good solution to represent design information in AOP languages

- Requirements for implementation:
  - Pointcuts that can refer to annotations
  - Means to introduce/derive annotations (implement late binding, reasoning)
  - Ensures that decoupling of design information from base code is possible
Integration in Compose* (1)

Selection based on annotations

```
[BusinessObject]
public class User {
    String name;
    String email;
    SessionID session;
}
```

```
concern Persistence {
    filtermodule PersistAdvice {..}
    superimposition selectors
        persistClasses = { C | classHasAnnotationWithName (C, 'BusinessObject') };
        filtermodules
            persistClasses <- PersistAdvice;
}
```

Benefit:
- Write pointcuts based on explicit design information
Integration in Compose* (2)

- Superimposition of annotations

Concern `MyAppPersistence` {
  superimposition
  selectors
  transFields={F | fieldType(F, T), isTypeWithName(T, 'SessionID')};
  annotations
  transFields <= Transient;
}

Benefits:
- Modular specification of scattered annotations
- Late binding
**Integartion in Compose* (3)**

- **Derivation of annotations**

```java
[BusinessObject]
public class User {
    String name;
    String email;
    SessionID session;
}

concern PersistenceView {
    superimposition
    selectors
    persFields = { F | classHasAnnotationWithName(C, 'BusinessObject'), hasField(C, F), not(fieldHasAnnotationWithName(F, 'Transient')) };
    annotations
    persFields <- Persistent;
}
```

**Benefit:** Reasoning to derive design information

Motivation -> Problem -> Analysis -> Realization -> Conclusion
Application: Decoupling pointcuts & advice

Motivation -> Problem -> Analysis -> Realization -> Conclusion

```plaintext
concern SecurityLog{
  [Monitoring] filtermodule AccessMonitoring{..}
  ...
}

concern Debugging {
  [Monitoring] filtermodule LoggingModule{..}
  superimposition
  selectors
    criticalClasses = { AnyRes |
      isClassWithName(Res, 'Resource'),
      inInheritanceTree(Res, AnyRes) };
    monitoringModules = { FM |
      isFilterModule(FM),
      hasAnnotationWithName(FM, 'Monitoring') };
  filtermodules
    criticalClasses <= monitoringModules;
}
```

Advice

Pointcuts

Binding
Conclusion: Benefits, contribution

- What did we gain?
  - The ability to express pointcuts based on *design information*
  
    - Pointcuts based on explicit design information are less fragile
    - Aspects are more reusable
  
    - Decoupling of annotations from base code, when the programmer wants it
Conclusion: Limitations, future work

- Limitations
  - Disciplined programming still required to keep annotations associated with proper elements (when they cannot be derived)
  - Annotations may require parameters for passing context; this is hard to include when superimposing annotations
  - Current implementation can only use the (type)name of annotations