University of Twente

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

TRESE

trese.cs.utwente.nl

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

University of Twente

- 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

University of Twente

- 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

University of Twente

Naming patterns

- Design intention can be obtained from identifiers
- Example: a method changes object state
  Encoding: public void **set**Name(..)
  Pointcut: execution(public void **set***(..));
- Problems
  - Tight coupling between pointcut and base classes (fragile pointcuts)
  - Representing multiple semantic properties

Accessing design information (2): Encoding

University of Twente

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

University of Twente

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

University of Twente

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

University of Twente

Desired properties →

Techniques	Separability of Props.	Multiple Props.	Scattered Properties	Late Binding	Domain Spec.Props.
Naming Patterns	no	no	no	no	yes
Structural Patterns	no	yes	no	no	yes
Annotations	yes	yes	yes	no	yes
Deriving	yes	yes	no	yes	no

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)

University of Twente

- 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)

University of Twente

- Superimposition of annotations

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

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

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

[Transient]

Benefits:

- Modular specification of scattered annotations
- Late binding

Integration in Compose* (3)

University of Twente

- Derivation of annotations

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

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

```
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

Application: Decoupling pointcuts & advice

University of Twente

Advice

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

Pointcuts

Binding

```
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;
}
```

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

University of Twente

- 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