Master project: Partial Declarations for Aspectual Adapters

Arnout Roemers
a.roemers@student.utwente.nl

University of Twente
Software Engineering Dept.,
Enschede, The Netherlands

April 26, 2012

Content

Content

Domain Specific Languages

- Programming languages
- Closely reflect specific domain
- Examples: BNF, CSS and SQL
- Might require code generation

DSLs - Advantages

- They offer high abstraction
- Improves developer productivity
- Improves program validity
- Yields better understanding by domain experts

DSLs - High development costs

- Implementation takes time and is complex
  - i.e. Compilers, editors, debuggers, et cetera
- Relatively higher costs than development of General Purpose Languages (such as C++ or Java)
  - A DSL tends to evolve more
  - A DSL is less used
- Difficult to say when productivity gain outweighs development costs
  - Holds back adoption in industry

DSLs - High development costs

- Implementation takes time and is complex
  - i.e. Compilers, editors, debuggers, et cetera
- Relatively higher costs than development of General Purpose Languages (such as C++ or Java)
  - A DSL tends to evolve more
  - A DSL is less used
- Difficult to say when productivity gain outweighs development costs
  - Holds back adoption in industry
- So: we need to lower the cost

DSLs - Back-end reuse problem

- Reusability of code generators lowers the cost of DSL implementation
- Reuse of *front-end* is common research topic
  - Modular language constructs
  - Automatic generated syntax aware and semantics aware editors
- Not so much research for reuse of the *back-end*
  - The generated code

DSLs - Back-end reuse problem

- Reusability of code generators lowers the cost of DSL implementation
- Reuse of *front-end* is common research topic
  - Modular language constructs
  - Automatic generated syntax aware and semantics aware editors
- Not so much research for reuse of the *back-end*
  - The generated code
- Back-end reuse problem: Improving the reuse of code generators and its generated code integration flexibility

DSLs - Back-end reuse problem

- Solution has ideally the following properties
  - Generated code applicable to many system designs
  - DSL not polluted with system design domain
  - No changes to code generation process
  - No explicit changes in (legacy) base system
- Requires the following
  - Standardized framework supporting loose coupling
  - Glueing for binding/integration
- How should this binding glue look like?
  - A second DSL or model?
  - What framework does this need?
  - What DSL implementation approaches can we utilise?

Content

Implementation approaches

- Research topics
  - What approaches to DSL implementation are currently out there?
  - How do those fit for the back-end reuse problem?
- Internal approaches
  - Use host language's native language constructs and/or tool set
- External approaches
  - Use tools outside the host language's native tool set and of which it is oblivious

Internal approaches

- Fluent API
  - e.g. Scala-Query library
- Abstract Syntax Tree expansion (macros)
  - e.g. Template Haskell
  - Converge [Tratt, 2008]
- Meta-Object Protocol interception
  - e.g. *An Architecture for Composing Embedded Domain-Specific Languages* [Dinkelaker et al, 2010]
- Extensible compiler
  - e.g. scalac (Scala compiler)

External approaches - How to generate

- Preprocessor
  - e.g. C++ templates
  - MetaBorg [Bravenboer & Visser, 2004]
- Interpreter
- Model transformer
  - e.g. Xtext
  - Clearwater [Swint et al, 2005]

External approaches - What to generate

- API calls to fixed library
- Utilising Event-Driven Architecture
  - e.g. EScala [Gasiunas et al, 2011]
- Hooks or callbacks
- Dependency injection
  - e.g. Google Juice
- Utilising Aspect-Oriented Programming
  - e.g. generating standard aspects (AspectJ)
  - developing Domain-Specific Aspect Languages, like XAspects [Shonle et al, 2003]
- Utilising Feature-Oriented Programming
  - e.g. API calls to a FOP engineered library

Evaluation - External 'what to generate' - Criteria

- Reuse
  - Generated code → how well is the generated code applicable to varying system designs
  - Generator → how stable is the generation process when applying its generated code to varying systems
  - Host language → how common are the language features required
- Type-safety
  - How confident can one be that the binding is working
  - How type-invasive is the approach
- Encapsulation
  - How well can one reason about the generated code in a modular fashion
- Coupling
  - How well can one update bound components without affecting others

Evaluation - External 'what to generate' - Result

- Not complete yet
- An Aspect-Oriented approach seems most flexible
  - In combination with another approach
  - AOP used for glueing
  - Makes it very applicable to varying system designs
  - Many OO languages support a form of AOP
  - Non-type-invasiveness is possible
  - Encapsulation is possible [Lieberherr et al, 2003]

Evaluation - External 'what to generate' - Result

- Not complete yet
- An Aspect-Oriented approach seems most flexible
  - In combination with another approach
  - AOP used for glueing
  - Makes it very applicable to varying system designs
  - Many OO languages support a form of AOP
  - Non-type-invasiveness is possible
  - Encapsulation is possible [Lieberherr et al, 2003]
- "An AOP approach" is still very broad

Content

Aspectual Adapters

- Kardelen Hatun is working on declarative adapters
- Adaptees are selected by AOP point-cuts
- Client calls to adapted methods in the adaptees are altered using AOP, so these calls use the adapters instead
- Clients of the adaptees get their required behaviour, in a non-behaviour-invasive way with respect to the adaptee itself
- Clients keep their original reference and adaptees are not altered in any way, so this can be considered non-type-invasive (e.g. reflection and polymorphism work just as expected).

Aspectual Adapters - Example

```
abstract class Shape {
  public int posX() { .. }
  public int posY() { .. }
}

class Circle extends Shape {
  public int radius() { .. }
}

class Rectangle extends Shape {
  public int lengthX() { .. }
  public int lengthY() { .. }
}

interface ShapeArea {
  public int getArea()
}
```

Aspectual Adapters - Example

```
aspect Shapes {

  instance pointcut shapes(Shape) =
    call(public Shape+.new()) && target(Shape);

  declare adapter: RectangleAdapter[ShapeArea]
      adapts shapes[Rectangle] {

    public double ShapeArea.getArea() {
      return shapes.lengthX() * shapes.lengthY();
    }
  }

  declare adapter: CircleAdapter[ShapeArea]
      adapts shapes[Circle] {

    public int ShapeArea.getArea() {
      return Math.pi * shapes.radius() * shapes.radius();
    }
  }
}
```

Partial Declarations

- Adapter behaviour and adaptee interface are tightly coupled
- Need level of indirection to separate platform-independent and platform-specific code
- Hope to achieve:
  - Loose coupling
  - Reusable adapters
  - Better encapsulation of adapter declarations

Partial Declarations - Example

```
aspect Shapes {

  instance pointcut shapes(Shape) =
    call(public Shape+.new()) && target(Shape);

  declare adapter: RectangleAdapter[ShapeArea]
      adapts shapes[Rectangle] {

    public double ShapeArea.getArea() {
      return shapes.lengthX() * shapes.lengthY();
    }
  }

  declare adapter: CircleAdapter[ShapeArea]
      adapts shapes[Circle] {

    public int ShapeArea.getArea() {
      return Math.pi * shapes.radius() * shapes.radius();
    }
  }
}
```

Partial Declarations - Example

```
aspect Shapes {

  instance pointcut shapes($SHAPECLASS$) =
    call(public $SHAPECLASS$+.new()) &&
        target($SHAPECLASS$);

  declare adapter: RectangleAdapter[ShapeArea]
      adapts shapes[$RECTANGLECLASS$] {

    public double ShapeArea.getArea() {
      return shapes.$LXMETHOD$ * shapes.$LYMETHOD$;
    }
  }

  declare adapter: CircleAdapter[ShapeArea]
      adapts shapes[$CIRCLECLASS$] {

    public int ShapeArea.getArea() {
      return Math.pi * shapes.$RADMETHOD$ *
          shapes.$RADMETHOD$;
    }
  }
}
```

Partial Declarations - Wrap up

- Aspectual Adapters with Partial Declarations can be generated as part of the DSL component
- Filling the Partial Declarations glues the adapters to the base system
- Questions:
  - How flexible is this approach with respect to the back-end reuse problem?
  - How to change behaviour in the base system itself? Declarative decorators?
  - What static analysis is required to ensure correct filling of the partial declarations?

Thank you