An Overview of DSLs

Tony Clark

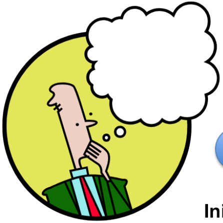
School of Engineering and Information Sciences University of Middlesex

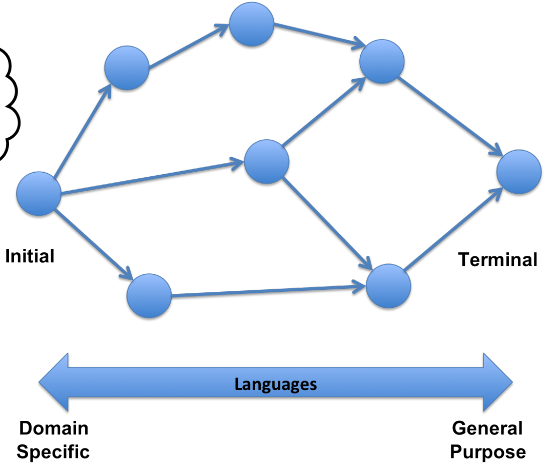
November 24, 2010

Outline

1. Domain Specific Languages
    - What are DSLs?
    - An Example DSL
    - A Word about Frameworks
    - General Properties
2. Two Types of Language Engineering
    - $\lambda - Calculus$
    - Modelling
3. Technologies for Language Engineering
    - The XMF Family
    - Language Factories
4. DSM-ing as Theory Building

What Are Domain Specific Languages?





Types of Domain Specific Languages

Internal	A host language is extended with sub-languages that are used for a specific aspect of the application. The host language is the *glue*.
External	An application consists of modules for different aspects. Each module is written in a different language.
Textual	A programming language or a domain specific configuration file.
Graphical	Using icons and graphical layout to represent 'initial' elements.

An Example DSL: Domain (due to Martin Fowler)

```
SVCLFOWLER 10101MS0120050313
SVCLHOHPE 10201DX0320050315
SVCLTWO x10301MRP220050329
USGE10301TWO x50214..7050329
```

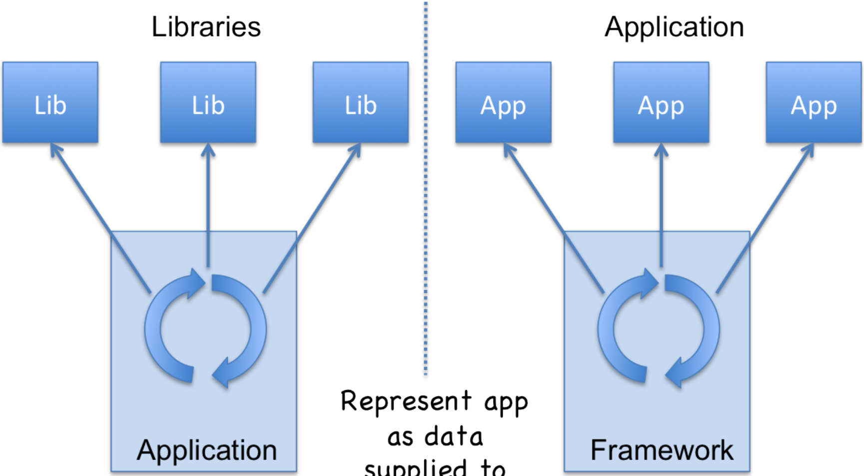
An Example DSL: A Framework

```
public void ConfigCallReader(Framework f) {
  f.registerStrategy(ConfigureServiceCall());
  f.registerStrategy(ConfigureUsage());
}
private ReaderStrategy ConfigureServiceCall() {
  ReaderStrategy r =
    new ReaderStrategy("SVCL",ServiceCall);
  r.addFieldExtractor(4,18,"CustomerName"));
  r.addFieldExtractor(19,23,"CustomerID"));
  r.addFieldExtractor(24,27,"CalltypeCode"));
  r.addFieldExtractor(28,35,"DataOfCallString"));
  return r;
}
```

An Example DSL: Language To Configure the Framework

```
@Reader CallReader
  map(SVCL,ServiceCall)
    4-18:CustomerName
    19-23:CustomerID
    24-27:CallTypeCode
    28-35:DataOfCallString
  end
  map(USGE,Usage)
    4-8:CustomerID
    9-22:CustomerName
    30-30:Cycle
    31-36:ReadDate
  end
  do
    ServiceCall
    Usage
end
```

Frameworks: Inversion of Control



Represent app as data supplied to framework: a DSL

Benefits and Drawbacks

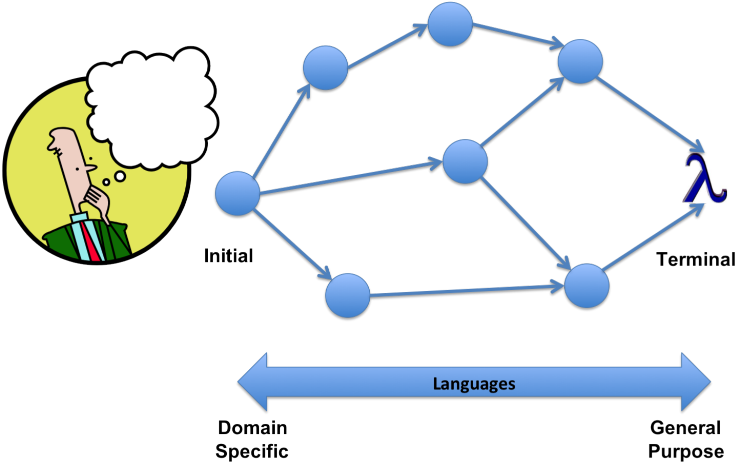
Benefits:

- Speed of application development.
- Domain specific tools: quality.
- Ease of maintenance.

Drawbacks:

- DSL development effort.
- Lack of tool support.
- DSL development expertise.
- DSL maintenence expertise.

λ – *Calculus*: A Terminal Candidate



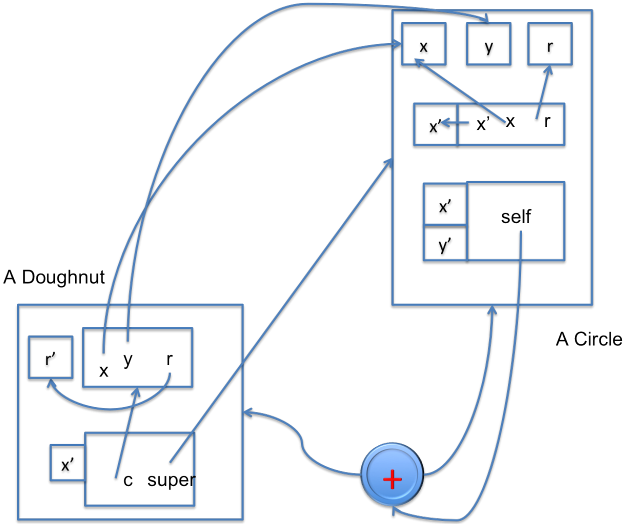
$\lambda -$ *Calculus* Definition

```
E ::= V              variables
  |   fun(V) E       functions
  |   E E            applications
```

Simple DSL

```
class Circle(x,y,r) {
  meth xin(x') { ret [x-r,x+r].contains(x'); }
  meth yin(y') { ret [y-r,y+r].contains(y'); }
  meth in(x',y') { ret self.xin(x') & self.yin(y'); }
}
class Doughnut(x,y,r,r') extends Circle {
  var c = new Circle(x,y,r');
  meth xin(x') { ret super.xin(x') & !c.xin(x'); }
  meth yin(y') { ret super.yin(y') & !c.yin(y'); }
}
```

What Do Run-Time Values Look Like?



λ – *Calculus* Extensions: Definitions

```
E ::= V                variables
  |   fun(V) E         functions
  |   E E              applications
  |   let D            top-level definitions
D ::= V = E            value definitions
  |   V(V) = E         function definitions
```

Translation: Class Definitions

```
class Circle(x,y,r) { ... }
class Doughnut(x,y,r,r’) ... { ... }
```

```
let Circle(x,y,r) = fun(self) ...
let Doughnut(x,y,r,r’) = fun(self) ...
```

$\lambda -$ *Calculus* Extensions: Records

```
E ::= V                variables
   |    fun(V) E       functions
   |    E E            applications
   |    let D          top-level definitions
   |    let D in E     local definitions
   |    { D* }         records
   |    E + E          addition (overloaded)
   |    E.V            field ref
D ::= V = E            value definitions
   |    V(V) = E       function definitions
```

Translation: Class Bodies

```
let Circle(x,y,r) = fun(self) {
  xin(x') = ...;
  yin(y') = ...;
  in(x',y') = ...
}
let Doughnut(x,y,r,r') = fun(self)
  let super = Circle(x,y,r)(self)
  in let c = ...
     in super + {
       xin(x') = ...
       yin(y') = ...
     }
```

$\lambda -$ *Calculus* Extensions: Recursion

```
E ::= V             variables
  |   fun(V) E      functions
  |   E E           applications
  |   let D         top-level definitions
  |   let D in E    local definitions
  |   { D* }        records
  |   E + E         addition (overloaded)
  |   E.V           field ref
  |   Y E           fixed point: (Y(E))(E') = E'
D ::= V = E         value definitions
  |   V(V) = E      function definitions
```

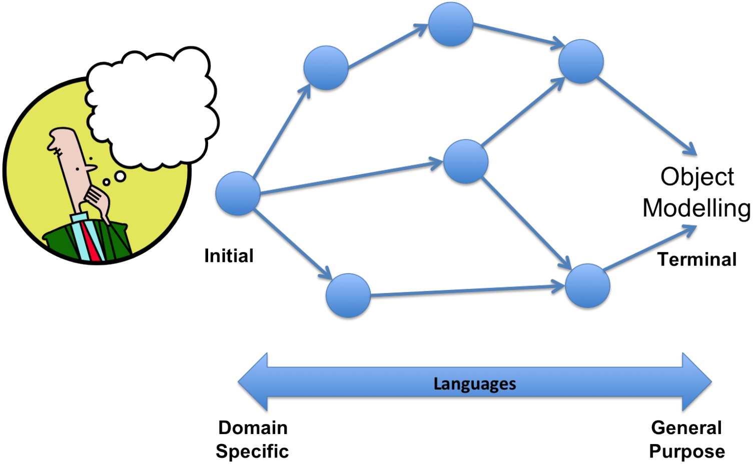
Translation: Method Bodies

```
let Circle(x,y,r) = fun(self) {
  xin(x’) = [x-r,r+x].contains(x’);
  yin(y’) = [y-r,r+y].contains(y’);
  in(x’,y’) = (self.xin)(x’) & (self.yin)(y’)
}
let Doughnut(x,y,r,’) = fun(self)
  let super = Circle(x,y,r)(self)
  in let c = Y(Circle(x,y,r’))
     in super + {
       xin(x’) = (super.xin)(x’) & !c.xin(x’)
       yin(y’) = (super.yin)(y’) & !c.yin(y’)
     }
```

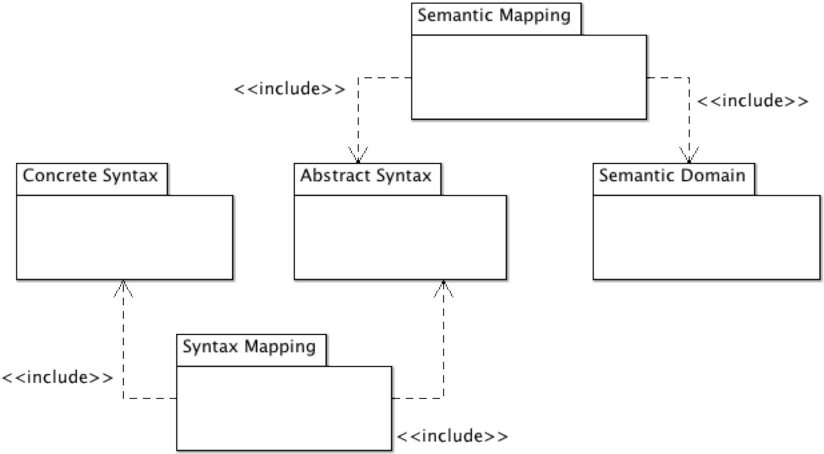
λ-calculus: Conclusion

- Design a domain specific language (e.g. classes, methods etc)
- Translate it onto the basic λ-calculus.
- Invent:
  - syntax extensions for λ (e.g. records, definitions, Y).
  - evaluation extensions for λ (not shown, but SECD is a good place to start).
- Each to implement directly or use a blueprint for translation.
- To start: write an interpreter for λ and extend it.

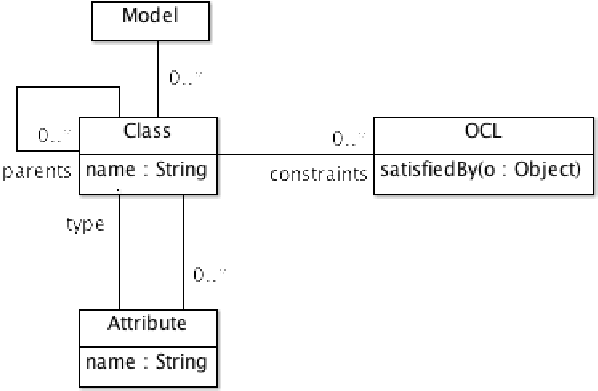
Objects: A Terminal Candidate



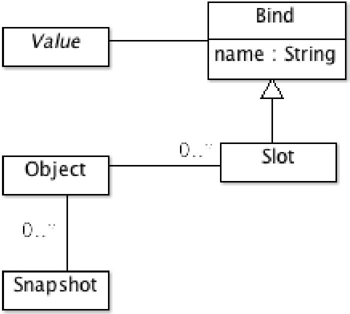
Modelling Languages



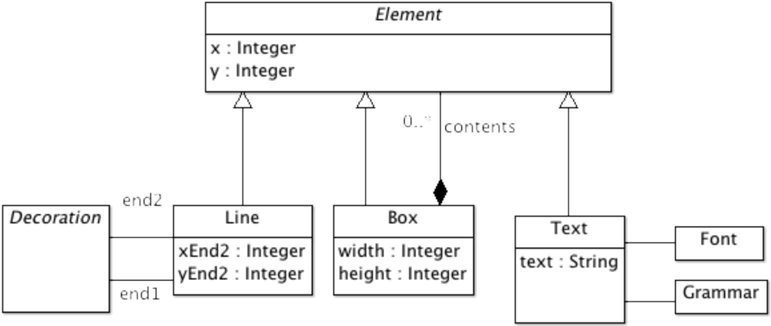
A Simple Modelling Language: Abstract Syntax



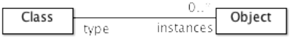
A Simple Modelling Language: Semantic Domain



A Simple Modelling Language: Concrete Syntax

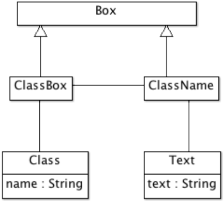


Semantic Mapping



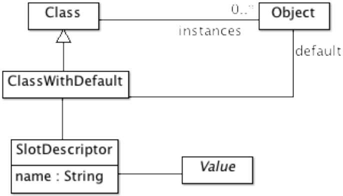
```
context Class inv instance_has_all_slots:
 attributes.name = instances.slots.name
context Class inv instance_satisfies_constraints:
 constraints->forAll(c |
  instances->forAll(o | c.satisfiedBy(o)))
context Class inv instances_conform_to_super:
 parents->forAll(c |
  c.instances->includesAll(instances))
context Object inv slot_values_are_type_correct:
 slots->forAll(s |
  s.value oclIsTypeOf(Object) implies
   s.value.type.instance->includes(s.value)
```

Syntax Mapping



```
context Class inv display_name:
  classBox.className.text.text = name
context ClassBox inv name_at_top:
  className.x = x and
  className.y = y and
  className.width = width
context ClassName min_height:
  height >= text.getHeight()
```

A Language Extension: Default Instances



```
context ClassWithDefault inv default_is_instance:
  instances->includes(default)
context ClassWithDefault inv default_well_formed:
  slotDescriptor.name->subSet(default.slots.name)
context ClassWithDefault inv defaul_slots_OK:
  slotDescriptor->forAll(d |
    default.slots->exists(s |
      s.name = d.name and s.value = d.value))
```

Object Modelling: Conclusion

- Design a language in terms of models for abstract syntax, concrete syntax amd semantic domain.
- Define mappings between them.
- Know about:
    - class-models (modelling languages often extend this).
    - object-models (a universal semantic domain).

XMF

XMF language definition features:

- Reflexive: Meta-Obejct Protocol (MOP)
- Language definition features: internal and external languages.
- Access to abstract syntax.
- Libraries for language processing.

XMF-Mosaic:

- Modelling IDE built using XMF.
- Graphical languages modelled using meta-models.
- Tool Models.

XMF Example: State Machine Syntax Class

```
@Class StateMachine
 @Grammar extends OCL.grammar
  StateMachine ::= ’(’ n=Name ’)’ es=Exp* {
   [| StateMachine(<n>,<es>) |] }.
 end
end
@Class Transition
 @Grammar extends OCL.grammar
  Transition ::= ’(’ sn=Name ’,’ tn=Name ’)’ {
   [| Transition(<sn>,<tn>) |] }.
 end
end
@Class State
 @Grammar extends OCL.grammar
  State ::= n=Name { [| State(<n>) |] }.
 end
end
```

XMF Example: State Machine Use

```
@Class StateMachineGenerator
  @Attribute counter:Integer end
  @Operation mkStateMachine()
    self.counter := counter + 1;
    @StateMachine(Off)
      @State Off end
      @State On end
      @Transition(On,Off) end
      @Transition(Off,On) end
    end
  end
end
```

Language Factories

- A component-based approach to the definition and construction of languages, tools.
- Language Factories aim to support:
  - **Reuse** of common language components.
  - **Agile** language engineering.
  - Language **refactoring**.
  - Language **analysis** including **impact analysis**.
- **Product Lines** for Languages
- Users:
  - Language Factory Developers
  - Language Factory Users
  - Application Developers
  - Application Users

A Language Factory

```
lang:
  ast: ...
  grammar: ...
  semantics eval(env): ...
  semantics java: ...
  constraints: ...
```

Example Language

```
component Landing_gear(height:int, speed:float) {
  stm {
    state Moving_Up
    state Moving_Down
    state Deployed
    state Stowed
    transition up from Deployed to Moving_Up
      height_change[height>500ft and speed>100kn/s]
    transition down from Stowed to Moving_Down
      deploy
  }
}
```

XPL: A Calculus for DSL Analysis

```
E ::= V                   variables
   |   fun(V) E           functions
   |   E E                applications
   |   { R* }             grammars
   |   intern E { S }     language use
R ::= ...                 syntax synthesizers
S ::= ...                 strings
```

Other Technologies

- UML and Profiles
- EMF, GMF
- XText
- Converge
- MetaEdit+
- View based language environments.

Research Problems

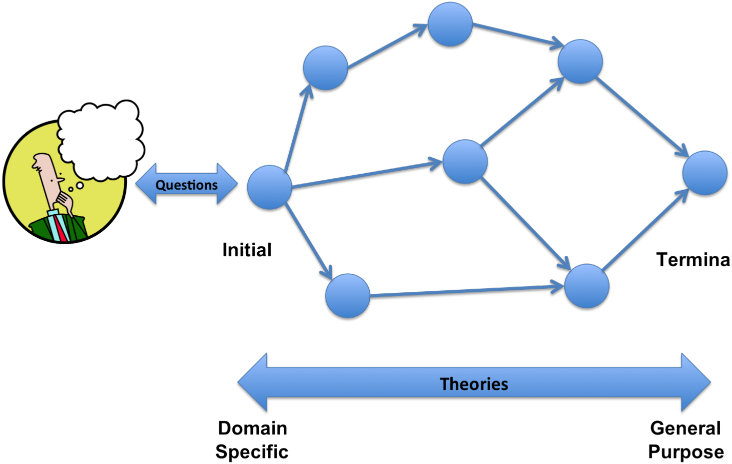
- Modularity and reuse.
- Text-based systems and parsing.
- Type systems.
- Tooling and complexity.
- Interoperability.
- Code generation vs models at run-time.

Peter Naur: Programming as Theory Building

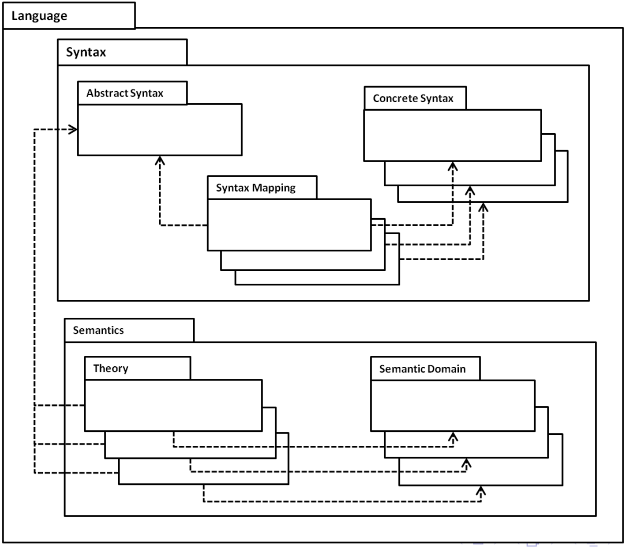
Naur's Thesis:

- Programmers design theories then map them to implementations.
- Theories allow questions to be asked (wider scope than a run).
- The theories are the things that capture the domain.
- Don't change the program directly, change the theory and calculate the impact.
- Actually multiple theories for different viewpoints.
- The initial theory is unattainable, but can be incrementally approximated.

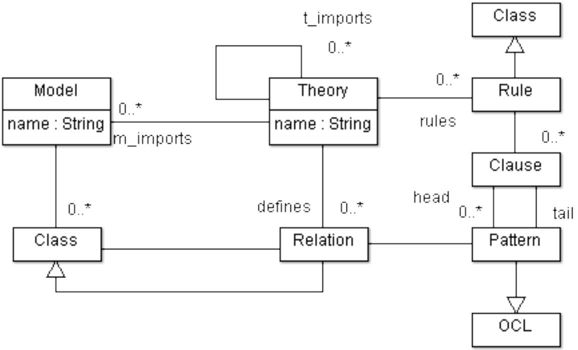
DSM-ing as Theory Building



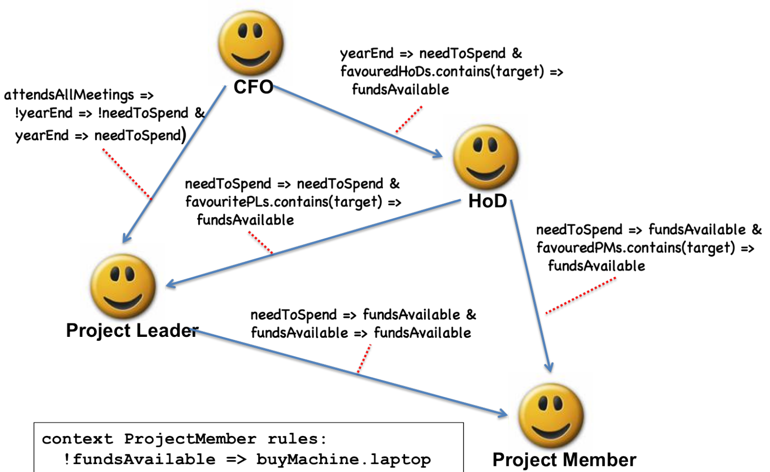
DSL Architecture



Theory Modelling



Example: Influencing Models



attendsAllMeetings =>
!yearEnd => !needToSpend &
yearEnd => needToSpend)

yearEnd => needToSpend &
favouredHoDs.contains(target) =>
fundsAvailable

needToSpend => needToSpend &
favouritePLs.contains(target) =>
fundsAvailable

needToSpend => fundsAvailable &
favouredPMs.contains(target) =>
fundsAvailable

needToSpend => fundsAvailable &
fundsAvailable => fundsAvailable

```
context ProjectMember rules:
  !fundsAvailable => buyMachine.laptop
   fundsAvailable => buyMachine.desktop
   fundsAvailable => machineChoice.apple
  !fundsAvailable => machineChoice.PC
```

Conclusion

- Domain Specific activities are about *theory building* and *theory transformation*.
- To be effective you need to *know your terminal theory well*.
- Terminal theories known to be effective (and universal):
    - λ-calculus
    - object models.
- Other, less universal, targets are just as effective (e.g. Java).
- Practical issues:
    - really translate (code generation)
    - simulate the transformation (interpret the model)
- General problems:
    - tools to support the process.
    - up-front resource and expertise.