Using Domain Specific Languages in the context of Product Line Engineering

Markus Voelter
Independent/itemis
voelter@acm.org

Using
Domain Specific Languages
in the context of
Product Line Engineering

1. DSLs
2. PLE Concepts & DSLs
3. Customization and Configuration
4. Modular Languages

Markus Voelter
Independent/itemis
voelter@acm.org
1 DSLs
programming started close to the hardware

abstractions ~computing chips

abstractions ~computing bits
abstractions
～computing

abstractions
～computing?
abstractions  
~computing?
general purpose
domain specific

tailor made
effective++
specialized, limited
used by experts
together with other specialized tools
A DSL is a focussed, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are natural/suitable for the stakeholders who specify that particular concern.
A DSL is a **focussed**, **processable** language for describing a specific **concern** when building a system in a specific **domain**. The **abstractions** and **notations** used are natural/suitable for the **stakeholders** who specify that particular concern.
A DSL is a focussed, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are natural/suitable for the stakeholders who specify that particular concern.
A DSL is a **focussed, processable language** for describing a specific **concern** when building a system in a specific **domain**. The **abstractions** and **notations** used are **natural/suitable** for the **stakeholders** who specify that particular concern.
A DSL is a **focussed, processable language** for describing a specific concern when building a system in a specific domain. The **notations** used are natural/suitable for the **stakeholders** who specify that particular concern.
execute?
map

DSL Program
(aka Model)

map

automated!

GPL Program
Analysing Domains
Defining Languages
Adapting/Selecting
Building Editors
Transforming Models
Building Generators
Building Frameworks
Analysing Domains
Defining Languages
Adapting/Selecting
Building Editors
Transforming Models
Building Generators
Building Frameworks
... and using all of that to build apps
Some Examples

Embedded Protocol Handler

```plaintext
procedure writeRegisterNumber2 requestCode 0x29 (
  request: struct request1 {
    int8 aci_pattern1;
    2:000;
    };
    parentRequestCode;  
  };
  int8 response;
} ;
reply: struct dwCStr{} {
  int8 status;
  int8 registerAddress;
  int8 registerData [2];
} ;

request: struct request2{
  int8 registerType pattern 1
  4:0000;
  4:registerType;
  int8 registerAddress;
  int8 registerData [2];
} ;
```
subsystem the.moon.scenarios {

    private:
    component Armstrong {
        task anyRole: scheduled oncePer5
        response time: huge [-..-]
    }

}

subsystem the.world.scenarios {

    node earth {
        subsystem the.moon.scenarios
    }

}

import "c:/c:/world/decomp/"
import "classpath/vults.md"
import "classpath/software.mnc"

namespace xl
uses units
condition locked
condition blinkingLight

menu Normal: label "Standard Menu"
  item unlockNow sysTurnOffAlarm if locked
    button label "Unlock"
  subMenu Manual label "Manual Settings"
    item alarmLevel sys[AlarmLevel] valRange SoundLevel restrict 0..10
    item unlock sys[TurnOffAlarm] if blinkingLight bool
  end
  subMenu autolocking label "Automatic Locking"
    item startTme sys[TurnOnAlarm]
    valRange Time
    item endTme sys[TurnOffAlarm]
    valRange Time
    template areaSettings [size=5, area=1, sys[TurnOnAlarm] area=5 settings
    template areaSettings [size=5, area=2, sys[TurnOnAlarm] area=2 settings
  end
end

template [size: int, area: int, sv: moref] areaSettings
  item unlock sys[TurnOffAlarm] label="unlock "area" on/off"
  bool true = label[sv] "on" false = label "off"
  item test sys[AlarmLevel] label = "test"
  bool
  item alarmLevel sys[AlarmLevel]
  valRange SoundLevel restrict 0..10
end

menu Export extends Normal
  item master sys[UnlockNow] afterItem unlockNow
end

message TurnOffAlarm
message TurnOnAlarm
message AlarmLevel
message UnlockNow

component AlarmManager {
  receives TurnOffAlarm
  receives TurnOnAlarm
  receives AlarmLevel
}

component MasterSwitch {
  receives UnlockNow
}
A DSL for Hearing Aid Configuration

A DSL for Refrigerator Configuration
BPEL Designer

Block Diagrams
PLC Programming

State Charts
domain engineering

... domain analysis
... classifying variability
... defining DSLs

application eng.

... using the DSL to express systems
platform

... target environment
... abstractions of solution space
... can influence DSL concepts
  (Arch DSLs)

config knowledge

... transformations
... generators
... interpreters
core asset

... languages
... generators
... editors

variation point

... bind how
... kind of varibility
... bind when
PLE Concepts & DSLs

Bind How
Kinds of Variability
Bind When
Variability Mechanisms
Removal

... optionally take away from

Variability Mechanisms
Removal

... optionally take away from overall whole

Challenge:
overall whole can get big and unwieldy
Variability Mechanisms

Injection

... optionally add to minimal core

Challenge:
how to point into the core and add something to it
Parametrization
... define values for predefined params

Challenge: types for parameters can be non trivial (DSLs)
PLE Concepts & DSLs

Bind How
Kinds of Variability
Bind When
Configuration vs. Customization

Variability

- Routine Configuration
- Creative Construction

- Guidance, Efficiency
- Complexity, Flexibility

Configuration Parameters
Property Files
Wizards
Feature-Model Based Configuration
Graph-Like Languages
Frameworks
Manual Programming
Tabular Configurations

Configuration

... selecting options
... setting param values
Configuration

... selecting options
... setting param values

Configuration

Feature Models
Customization

... DSLs
... instantiation
... connections
DEMO I

Example DSL – Fountains
PLE Concepts & DSLs

Bind How
Kinds of Variability
Bind When
### Binding Time

<table>
<thead>
<tr>
<th></th>
<th>flexibility</th>
<th>performance</th>
<th>code size</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>source time</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>compile time</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>link time</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>load time</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>run time</td>
<td>+++</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

### Generation

resulting code can be easily inspected
Generation

resulting code can be easily debugged

Generation

resulting code can be optimized and more efficient
Generation

Templates can be derived from existing code

Generation

work around limitations of target language
Interpretation vs. Generation

Generation

no changes to target environment

(leaves no trace)

Generation

reuse runtime infrastructure

(garbage collection, monitoring...)
Interpretation vs. Generation

Interpretation
faster
turnaround
no regeneration
test
build
deploy

Interpretation
for platform independence
an interpreter might be
less porting
effort
Interpretation vs. Generation

Interpretation
possibly direct feedback from
in-IDE interpreter
DEMO II

In-IDE Interpreter in Fountains
Customization and Configuration

In-Language
Varying models
DSLs as Parameters
The language provides **explicit** support!

(we can learn that from GPLs)

**Conditionals**
conditional execution

**Specialization**
overriding, overwriting

**Leaving Holes**
for variant to fill in

**Inject Stuff**
several places at a time?
<table>
<thead>
<tr>
<th>In language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditionals</strong></td>
</tr>
<tr>
<td>#ifdef</td>
</tr>
<tr>
<td><strong>Specialization</strong></td>
</tr>
<tr>
<td>inheritance</td>
</tr>
<tr>
<td><strong>Leaving Holes</strong></td>
</tr>
<tr>
<td>template methods</td>
</tr>
<tr>
<td><strong>Inject Stuff</strong></td>
</tr>
<tr>
<td>AOP</td>
</tr>
</tbody>
</table>
DEMO III

A DSL with “variant blocks”
Customization and Configuration

In-Language Varying models
DSLs as Parameters

Two Levels
~ problem space
vs. software space

Problem Space: Configuration
Solution Space: Customization
Two Levels
~ problem space vs. software space

```c
#if defined (ACE_HAS_TLI)
    static ssize_t t_snd_n (ACE_HANDLE handle, const void *buf,
                            size_t len, int flags,
                            ACE_Time_Value *timeout = 0,
                            size_t *bytes_transferred = 0);
#endif /* ACE_HAS_TLI */
```

Two Levels Removal
Two Levels Removal

```java
component DelayCalculator {
    provides default: IDelayCalculator
    requires screens[0..n]: IInfoScreen
    provides mon: IMonitoring feature monitoring
}
```
Two Levels Removal

```csharp
namespace monitoringStuff feature monitoring {

    component MonitoringConsole {
        requires devices: [*]: IMonitor
    }

    instance monitor: MonitoringConsole

    dynamic connect monitor.devices query {
        type = IMonitor
    }
}
```
Two Levels Injection

```cpp
namespace monitoring {

    component MonitoringConsole ...
    instance monitor: ...
    dynamic connect monitor.devices ...

    aspect (*) component {
        provides mon: IMonitoring
    }
}

component DelayCalculator {
    ...
}
component AircraftModule {
    ...
}
component InfoScreen {
    ...
}

aspect (*) component {
    provides mon: IMonitoring
}
component DelayCalculator {
    ...
    provides mon: IMonitoring
}
component AircraftModule {
    ...
    provides mon: IMonitoring
}
component InfoScreen {
    ...
    provides mon: IMonitoring
}
```
Two Levels

Removal

Injection

namespace monitoring feature monitoring {
    component MonitoringConsole ...
    instance monitor: ...
    dynamic connect monitor.devices ...

    aspect (*) component {
        provides mon: IMonitoring
    }
}

Two Levels

~ varying models, not text

... fewer var points
... semantically meaningful
... more manageable
Two Levels
~ varying models, not text

... simplifies traceability
you can „point“ easily
to any element ->
generic traceability approach

... simplifies traceability
DEMO IV

Feature Annotations
Customization and Configuration

In-Language
Varying models
DSLs as Parameters
Feature Model w/ Parameters

... parameters with simple types
... complex types: DSL/metamodel
Customization and Configuration

Trafo Variability

Model-Based Implementation
create System transformPs2Cbd( Building building ): 

   hasFeature("burglarAlarm") ? ( handleBurglarAlarm() -> this) : this;

handleBurglarAlarm( System this ): 
let conf = createBurglarConfig(): { 
  configurations.add( conf ) -> 
    ... 
    conf.connectors.add( connectSimToPanel( createSimulatorInstance(), createControlPanelInstance() ) ) -> 
      hasFeature( "siren" ) ? conf.addAlarmDevice("AlarmSiren") : null -> 
      hasFeature( "bell" ) ? conf.addAlarmDevice("AlarmBell") : null -> 
      hasFeature( "light" ) ? conf.addAlarmDevice("AlarmLight") : null 
};
Transformation
Variability

Generator
Variability
Modular Languages
We don’t want to model, we want to program!
We don’t want to model, we want to program!

... at different levels of abstraction
... from different viewpoints
... integrated!

We don’t want to model, we want to program!

... with different degrees of domain-specificity
... with suitable notations
... with suitable expressiveness
We don’t want to model, we want to program!

And always: precise and tool processable

We don’t want to model, we want to program!
PLE for DSLs suitable to related domains
Big Language?

with many first class concepts!
Small Language?

with a few, orthogonal and powerful concepts

Modular Language

with many optional, composable concepts
Viewpoints
Viewpoints

suitable abstractions and notations for each

Viewpoints

Integrated via symbolic references and seamless transitions
**Viewpoints**

**General Purpose**
- predefined library
- configure

**Viewpoints**

**Domain Specific**
- custom purpose-built
- create/include
Custom Notations

real business expert integration

Viewpoints
Domain Specific

Viewpoints
General Purpose

Domain Specific

LEGO Robot Control
Viewpoints

General Purpose

Components

State Machines

Sensor Access

Domain Specific

LEGO Robot Control

C

General Purpose
Language Integration by...

Referencing

L1 \rightarrow L2

Language Integration by...

Cascading

L2 \downarrow L1
Language Integration by...

**Extension**

```
L1
   ↘
   |
   L2
```

Language Integration by...

**Embedding**

```
L1
 |
|
L2
```
Language Integration by...

Contributing

L1 → E → L2
A Modular Language
THE END.

.coordinates
web www.voelter.de
email voelter@acm.org
skype schogglad
twitter markusvoelter
xing http://www.xing.com/profile/Markus_Voelter
linkedin http://www.linkedin.com/pub/0/377/a31

THE END.
THE END.

coordinates
web  www.voelter.de
email voelter@acm.org
skype schogglad
twitter markusvoelter
xing  http://www.xing.com/profile/Markus_Voelter
linkedin http://www.linkedin.com/pub/0/377/a31

THE END.

coordinates
web  www.voelter.de
email voelter@acm.org
skype schogglad
twitter markusvoelter
xing  http://www.xing.com/profile/Markus_Voelter
linkedin http://www.linkedin.com/pub/0/377/a31
THE END.

coordinates

web  www.voelter.de
email voelter@acm.org
skype schogglad
twitter markusvoelter

xing  http://www.xing.com/profile/Markus_Voelter
linkedin http://www.linkedin.com/pub/0/377/a31