

# Xtext workshop – level: Beginner

## SIG Domain Language Engineering

Peter Vermeulen, Meinte Boersma

October 4th, 2010





# Objectives

## “Beginner” objectives

- Create simple DSL: grammar → parser, editor, meta model
- Create some code generation templates
- Enhance language: scoping, validation



# Topics

## “Beginner” topics

- Outline of Xtext Eclipse projects
- Xtext (slightly-less-)basics (with exercises)
- Code generation with Xpand and Xtend (with exercise)
- Xtext: scoping + validation (with exercise)
- Recap and conclusion
- Where to find more info



# Development environment

## Required components (on CD-ROM):

- JDK  $\geq$  1.5 (JDK6)
- Eclipse 3.6 Helios + Xtext 1.0.1 (Itemis distro)
- reference implementation:

<http://code.google.com/p/xtext-workshop-code1/>



# Outline of Xtext Eclipse projects

Demo



# Xtext basics for mockup 1

## Type rules:

- Look like this:

*Type*: 'keyword' *attribute1*=ID|STRING|...  
( *attribute2* += *Type2* ) +;

- Key concepts:
  - type in meta model
  - keywords
  - attribute assignment (=, +=)
  - group using () and define cardinality: +, \*, ?, *default-1*



# Xtext grammar exercises template

## Exercise: implement mockups, one by one

- Browse source in Google code:  
[trunk/org.xtext.workshop.notes/mockups/mockup-?.webgui](https://trunk/org.xtext.workshop.notes/mockups/mockup-?.webgui)
- Change `WebGui.xtext`
- Run MWE2 workflow `GenerateWebGui.mwe2`
- Run second Eclipse instance
  - copy mockups into generic Eclipse project
  - validate and test
- Rinse & repeat



# Xtext basics for mockup 2

## Cross-references:

- Look like this:

*attribute3* = [ *Type* ]

- References a type
- Default scope: *all Type-s* (customize through scoping)





# Xtext basics for mockup 2

## Group of alternatives:

- Look like this:

( Alt1 | Alt2 | ... )

- Tries to match alternatives, first one wins
- Combine with a type rule to “do OO”:

*Supertype: Subtype1 | Subtype2;*



# Xtext basics for mockup 3

## Boolean assignment:

- Looks like this:

```
attribute4 ?= 'abstract'
```

- Detects optional “stuff”, e.g. Java-like modifiers:

```
( public ?= 'public' )? 'class' name=ID ...
```

- Note: surround with ( ) ? to actually make right-hand side of assignment optional



# Xtext basics for mockup 4

## Terminal rules:

- Built-in:
  - ID: matches Java-like identifiers
  - STRING: matches strings delimited with ""
  - INT: matches an integer (optional sign)
- You can make your own terminal rules:

**terminal** *terminalRule* : *regexp*;

- “returns” a String
- e.g. useful for value literals (floats etc.)

# Xtext basics

## More types of rules (not in exercises):

- Enum rules:

**enum** *Enum* : *literal1* | *literal2* = 'keyword2' ;

- Data type rules:

*Datatype* : *rule1* \* *rule2* + ;

- no assignments, “returns” a String
- difference with terminal rules: “calls” other datatype or terminal rules
- e.g. useful for value literals (floats etc.)



# Xtext basics

## Hints:

- First rule is root element of your DSL
- `name` attribute is special: used for outline, exported names, etc.



# Code generation: Xpand and Xtend

## What's what:

- Xpand: template language
- Xtend: “functional” (OCL-like) language, useful for creating helper functions for Xpand templates and model-2-model transformations

## Nice feature of both:

- Both have *polymorphic dispatch*: which template/function is called depends on *runtime* type (instead of compile-time type as with Java)



# Code generation: Xpand

## Important stuff:

- Template instructions and *expressions* are inside « . . . »
  - Windows: Ctrl-Shift-<, Ctrl-Shift->
  - Mac: Alt-\, Alt-Shift-\
  - (both): Ctrl-Space ↑ Enter to get empty pair from content assist
- «**DEFINE** *name* **FOR** *type*» . . . «**ENDDEFINE**» defines a callable template fragment
- everything inside  
«**FILE** 'foo/Bar.java' » . . . «**ENDFILE**» ends up in file foo/Bar.java (auto-created)



# Code generation: Xpand (cont.'d)

## More important stuff:

- refer to model, e.g.: `«entity.name.toFirstUpper()»`
- `«EXPAND name FOR x»` calls template fragment *name* with *x*
- `«EXPAND name FOREACH collectionOfXs»` calls template fragment *name* for all *X*-s in the collection
- `«FOREACH collectionOfXs» . . . «ENDFOREACH»` expands the dots for all *X*-s in the collection
- more constructs: **IMPORT**, **EXTENSION**, **IF**, **LET**



## Basics:

- Define functions as follows:

```
String toJavaType(DataType this) : 'java.lang.String';
```

- ***collectionOfXs.typeSelect(subTypeOfX)*** selects specified sub type of X from collection of X-s
- ***collection.select( item | boolExpr )*** selects all items in collection which satisfy the boolean expression (*item* = temporary “loop” variable)



# Code generation: exercise

## Generate:

- HTML page for a *Page*, or
- POJO for an *Entity*

## Use a separate Xtend file for:

- `fileName(Page/Entity)`
- `toJavaType(Type)`



# Xtend (cont.'d)

## More advanced:

- Ternary operator: *expr* ? *a* : *b*
- More statements: **switch**, **let**, **if**
- Use **JAVA** keyword to invoke public static Java methods
- Use `org::eclipse::extend::util::stdlib::io::info()`  
from `org.eclipse.xtend.util.stdlib` plugin for  
`System.out.println()` -debugging

# Scoping

## Implement scoping:

- Look up `WebGuiScopeProvider` class
- Add methods with signature

`IScope scope_Type_crossReference (Type2, EReference)`

- Exercise: limit scope of referenced *Feature*-s to `context.features`



# Validation

## Implement validation:

- Look up `WebGuiJavaValidator` class
- Add methods with signature

```
@Check public void checkName(Type)
```

- Call `warning/error` to flag them
- Exercise: validate that name of *Entity* starts with a capital



# Recap

## Name “Xtext” is overused:

- grammar language
- generator for parser, editor, meta model
- runtime library used by generated DSL plugins

All this together is the *Xtext framework*, but

DON'T PANIC! 😊



# Conclusion

## When (not) to use Xtext:

- Domain is large enough with limited variation (number of concepts/constructs).
- Go for lots of “little languages” (sub domains) and interconnect these.
- Don't try and make another GPL: really large grammars are tricky to get right and who needs *another* GPL.



# Conclusion

## Pros and cons:

- Advantage over UML or internal DSLs: you start from scratch, don't need to restrict an existing language.
- Disadvantage of Xtext: no pre-cooked language libraries/fragments available, and you can only inherit from one other language.





# More information

## Where to find more:

- Eclipse help
- User Guide on <http://xtext.org/>
- [Eclipse forum](#)
- Blogs (e.g. <http://dslmeinte.wordpress.com/>, [Planet oAW](#))



**Together. Free your energies**

[www.capgemini.com](http://www.capgemini.com)