The EMF Parsley DSL: an extensive use case of Xtext/Xbase powerful mechanisms

Lorenzo Bettini
DISIA – University Firenze, Italy
Vincenzo Caselli **Francesco Guidieri**
RCP-Vision, Firenze, Italy

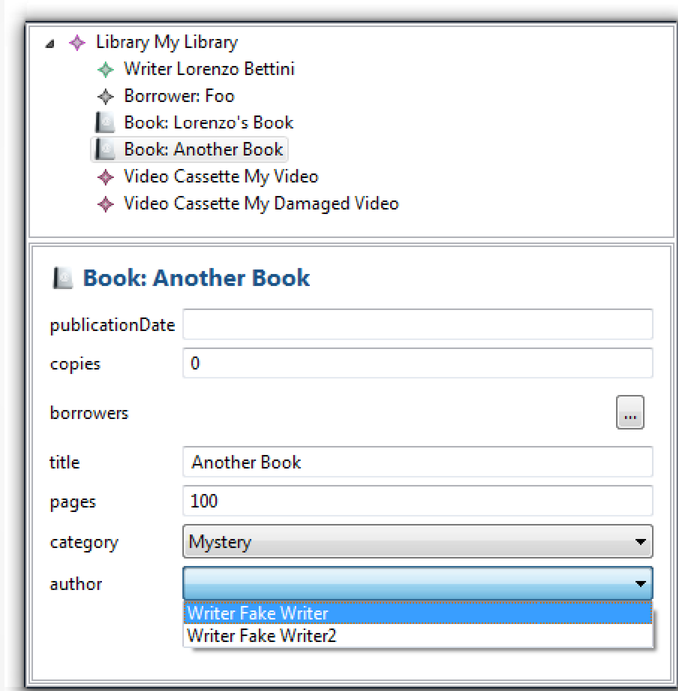
Main case study

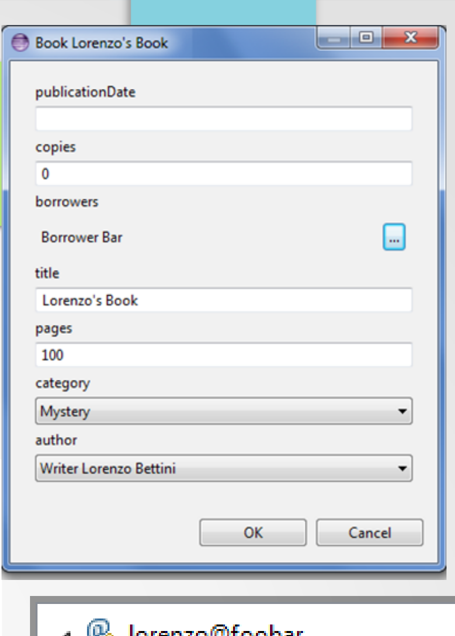
- EMF Parsley:
  - Quickly develop applications based on EMF models
  - Completely and easily customizable
  - Based on declarative customizations
  - Provides a DSL for easy configuration
  - Supports EMF persistences, XMI, CDO, etc.
  - Supports RAP

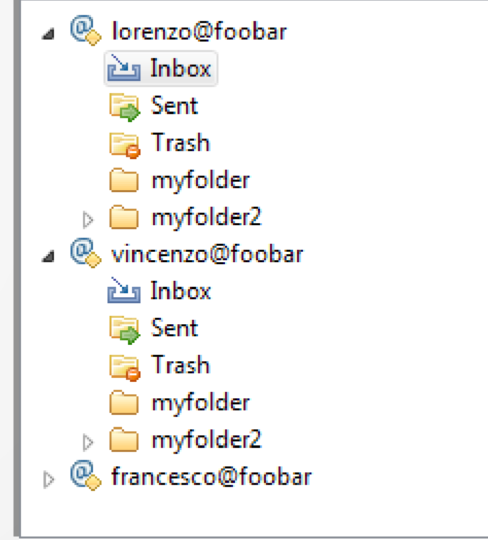
https://www.eclipse.org/emf-parsley

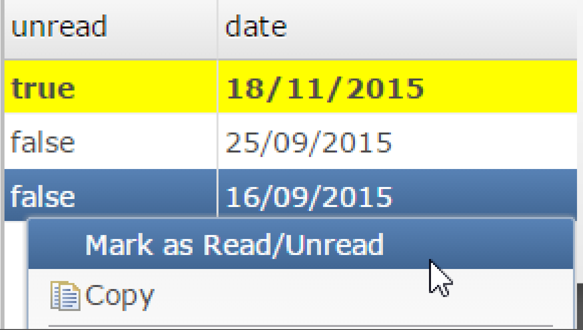
EMF Parsley

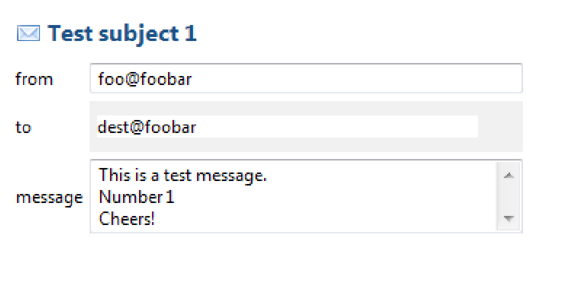
- Provides reusable and customizable Jface/SWT components
  - Tree
  - Form
  - Dialog
  - Editor
  - Combination of them
- Project wizard to get started











Under the hood

- Delegates to **EMF.Edit** by default
- Customizations based on **Dependency Injection** (Google Guice), NOT on extension points
- **Declarative customizations** (polymorphic dispatch), NOT on instanceof cascades
- **Xtext/Xbase DSL**
  - Code Generation oriented (NOT reflective => **Debuggable**!)

EMF Parsley DSL

- Rely on the EMF Parsley Java API
- Implemented in Xtext, using Xbase
  - Interoperable with the Java type system
  - IDE tooling (including Debugging)
- Specify customizations in one single file in a compact form
  - Generates the corresponding Java code
  - Generates the Guice bindings
- You can use the DSL and manually written Java code together

Static type checking

- All expressions are statically type-checked / inferred
  - Including EMF model feature access

```
38
39    featuresProvider {
40        features {
41            Library -> name
42            Writer -> name, address, books
43            Book -> author, titles, category
44        }
45    }
46
47    featureCaptionProvider {
48        text {
49            Book : author ->
50            Writer : name -> "Name:"
51        }
52    }
```

titles cannot be resolved.

1 quick fix available:

Change to 'title'

Press 'F2' for focus

DSL Java interoperability

- Thanks to Xbase:
  - Access to all Java types
  - According to project dependencies/classpath
  - DSL elements can extend your Java classes
  - DSL fully debuggable!

Imports management

- Automatic import and quickfixes

```
34  tableLabelProvider {
35      text {
36          MailMessage:date -> {
37              new SimpleDateFormat("dd/MM/yyyy")
38
39
40
41
42
43
```

Couldn't resolve reference to JvmConstructor 'SimpleDateFormat'.

2 quick fixes available:

Import 'SimpleDateFormat' (java.text)

Create Java class 'SimpleDateFormat'

JDT Integration

```
labelProvider {
    text {
        Book b -> { '"' + b.title + '"' }
        Writer w -> { w.name
    }
}

menuBuilder {
    val factory = EXTLibrary

    emfMenus {
        Writer w -> #[
            actionChange("Ne
                [
                    library
                    val book
                    library.books += book
```

String Book.getTitle()

Returns the value of the '*Title*' attribute.

Returns:
the value of the '*Title*' attribute.

See Also:
setTitle(String)
org.eclipse.emf.examples.extlibrary.EXTLibraryPackage.getBook_Title()

@model
@generated

Press 'F2' for focus

Packa... Plug-... *module.parsley

org.eclipse.emf.parsley.ex
org.eclipse.emf.parsley.ex

```
24  configurator {
25      resourceURI {
26          LabelsSaveableTreeView -> {
27              return URI
```

org.eclipse.emf.common.util.URI

A representation of a Uniform Resource Identifier (URI), as specified by RFC

URI - java.net
URI - org.eclipse.emf.common.util
URIDereferencer - javax.xml.crypto

EMF Parsley DSL: Building

- All artifacts are generated and kept in synch:
  - Java implementations
  - plugin.xml (more on that later)

- Fully integrated with Eclipse Building mechanism
  - All files are generated on DSL save

Very quick demo

- For a wider demo on Parsley, see
  - Website https://www.eclipse.org/emf-parsley
  - Talks and demos at previous EclipseCons

Use Xbase

- Most of the shown features are almost for free from Xbase
- As long as you follow its contracts!

DSL structural & behavioral aspects

- With Xtext it's easy to deal with **structural** aspects!
- What about **behavioral** aspects?
  - Grammar for expressions is tedious and recurrent
  - Deal with recursion
  - Implement the type system for expressions
  - Implement the code generator
  - What if we want to target the Java platform?

Enter Xbase

- A reusable OO expression language
- That you can embed in your DSL
  - Just inherit from the Xbase grammar
- You get for free:
  - Syntax for a **Java-like** expression language
  - Interoperability with the Java type system
    - Xbase type system implements the Java type system
    - Access any Java library in your classpath
    - Automatic Java code generator
    - Debugger!

Xbase: Java-like with less “noise”

- Semicolons optional
- Powerful type inference
- Everything is an expression
- Lambdas
- Extension methods

Example

```
val personList = newArrayList(
    new Person("James", "Smith", 50),
    new Person("John", "Smith", 40),
    new Person("James", "Anderson", 40),
    new Person("John", "Anderson", 30),
    new Person("Paul", "Anderson", 30))
val result = personList.
    filter[firstname.startsWith("J")].
    sortBy[age].
    take(3).
    map[surname + ", " + firstname].
    join("; ")
println(result)
```

Everything is Statically Type-checked (inferred)

```
Anderson, John; Smith, John; Anderson, James
```

Use Xbase

- Inherit your grammar from Xbase
  - You get Xbase expressions parsed in your DSL
- But what about all the other aspects?
  - Type system
  - Validation
  - Scoping
- You must implement a **model inferrer**...

The Xbase Java type model

- It models Java types
  - Classes, interfaces, enums,
- It models Java type elements
  - Methods (signatures), fields
- It is automatically populated by Xbase
  - With all the existing Java types in the classpath
    - Available as sources
    - Available as binaries (Java libraries)
- You need to make your DSL elements part of this type model

The model inferrer

- All you need to do is
  - Map your DSL model elements to the Java type model elements
  - Connect any Xbase (block of expressions) with a Java type model method
- Xbase will be able to
  - Perform all the type checking
  - Provide a proper scope for **this** and **super**
  - Generate the Java code

Example

- Inside “menuBuilder” we declare a variable “factory”
  - The “menuBuilder” element is mapped to a Java type model class
    - The “factory” is mapped to a Java type model field of the containing mapped class
    - Each “emfMenus” entry is mapped to a type model method of the containing mapped class
    - Result: we can access “factory” from within an “emfMenus” entry

```
menuBuilder {
    val factory = EXTLibraryFactory.eINSTANCE

    emfMenus {
        Writer w -> #[
            actionChange("New book", w.eContainer as Library,
                [
                    library |
                    val book = factory.createBook
                    library.books += book
                    book.title = "A new book"
                    book.author = w
                ]
            ),
```

Use annotations in your DSL

- Inherit your grammar from XbaseWithAnnotations (which extends Xbase grammar)
- Use XAnnotation in your grammar
- Map XAnnotation to Java type model annotation in your inferrer
- Example in Parsley DSL

```
FieldSpecification:
  annotations+=XAnnotation*
  (writeable?='var'|'val')
  extension?='extension'
  type=JvmTypeReference
  name=ValidID
  ('=' right=XExpression)? ';'?;
```

```
menuBuilder {
  @Inject(optional=true)
  var extension ILabelProvider lprov

  val factory =
   EXTLibraryFactory.eINSTANCE
```

Implement the mapping

- Use Xtend and the provided API (example: Domainmodel)

```
DomainModel:
  importSection=XImportSection?
  elements+=AbstractElement*;

AbstractElement: PackageDeclaration | Entity;

PackageDeclaration:
  'package' name=QualifiedName '{' elements+=AbstractElement* '}';

Entity:
  'entity' name=ValidID
    ('extends' superType=JvmParameterizedTypeReference)? '{'
     features+=Feature*
  '}';

Feature: Property | Operation;

Property: name=ValidID ':' type=JvmTypeReference;

Operation:
  'op' name=ValidID '(' (params+=FullJvmFormalParameter (','
    params+=FullJvmFormalParameter)*)? ')' (':' type=JvmTypeReference)?
  body=XBlockExpression;
```

- Use JvmTypesBuilder for creating Java type model elements and map them to your DSL elements

```
class DomainmodelJvmModelInferrer extends AbstractModelInferrer {

    @Inject extension JvmTypesBuilder
    @Inject extension IQualifiedNameProvider

    def dispatch infer(Entity entity,
                       extension IJvmDeclaredTypeAcceptor acceptor,
                       boolean prelinkingPhase) {
        accept(entity.toClass( entity.fullyQualifiedName )) [
            documentation = entity.documentation
            if (entity.superType !== null)
                superTypes += entity.superType.cloneWithProxies
...
```

```
// let's add a default constructor
members += entity.toConstructor []
// now let's go over the features
for ( f : entity.features ) {
  switch f {
    // for properties we create a field
    Property : {
       members +=  f.toField(f.name, f.type)
    }
    // operations are mapped to methods
    Operation : {
      members += f.toMethod(f.name, f.type ?: inferredType) [
        documentation = f.documentation
        for (p : f.params) {
          parameters += p.toParameter(p.name, p.parameterType)

        // here the body is implemented using a user expression.
        // Note that by doing this we set the expression into the context
        // of this method. The parameters, 'this' and all the members of
        // this method will be visible for the expression.
        body = f.body
     ]
   }
   ...
```

Artifact generation

- From a single DSL input file you generate several files: **OK**
  - In our context: we generate several Java files
- From several input files you generate a single file: **DON'T!**
  - In our context: we would need to generate a single plugin.xml in a project
    - From all the “parts” sections of all the EMF Parsley input files of the same project.

What we want

```
module mymodule1 {
  parts {
    viewpart my.view1 {
      viewname "My View 1"
      Viewclass ...
    }
  }
}
```

```
module mymodule2 {
  parts {
    viewpart my.view2 {
      viewname "My View 2"
      Viewclass ...
    }
  }
}
```

plugin.xml generated in the project

```
<plugin>
   <extension point="org.eclipse.ui.views">
      <view
            category="org.eclipse.emf.parsley"
            class="..."
            id="my.view1"
            name="My View 1">
      </view>
      <view
            category="org.eclipse.emf.parsley"
            class="..."
            id="my.view2"
            name="My View 2">
      </view>
   </extension>
</plugin>
```

What we do

```
module mymodule1 {
  parts {
    viewpart my.view1 {
      viewname "My View 1"
      Viewclass ...
    }
  }
}
```

```
plugin.xml_gen generated in a sep dir
<plugin>
 <extension point="org.eclipse.ui.views">
  <view category="org.eclipse.emf.parsley"
    class="..."
    id="my.view1"
    name="My View 1">
  </view>
 </extension>
</plugin>
```

```
module mymodule2 {
  parts {
    viewpart my.view2 {
      viewname "My View 2"
      Viewclass ...
    }
  }
}
```

```
plugin.xml_gen generated in a sep dir
<plugin>
 <extension point="org.eclipse.ui.views">
  <view category="org.eclipse.emf.parsley"
    class="..."
    id="my.view2"
    name="My View 2">
  </view>
 </extension>
</plugin>
```

Then a custom Eclipse builder reads all plugin.xml_gen and merges them into the single plugin.xml (we use PDE internal APIs for dealing with plugin.xml)

Xbase type inference

- Xbase has a powerful Java type inference system
  - E.g., you can avoid specifying types of lambda parameters in most cases
- Use it in your DSL:
  - Define a Java API using generics (this is the runtime library of your DSL)
  - Map your DSL elements to methods that have access to your Java API
  - And the corresponding XExpression body will automatically exploit Xbase type inference

Example

- Our Java API class EditingMenuBuilder defines this generic method:

```
<T extends Notifier> IMenuContributionSpecification actionChange(
  String text, T element, IAcceptor<T> changeImplementation)
```

- “emfMenus” entries are mapped to methods of a class extending EditingMenuBuilder
- Xbase type inference instantiates T with Library
- The lambda parameter library is of inferred type Library

```
menuBuilder {
    val factory = EXTLibraryFactory.eINSTANCE

    emfMenus {
        Writer w -> #[
            actionChange("New book", w.eContainer as Library,
                [
                    library |
                    val book = factory.createBook
                    library.books += book
                    book.title = "A new book"
                    book.author = w
                ]
```

Conclusions

- Xbase is not only grammar:
  - Complete integration with Java and JDT
  - Powerful type system
- Follow its contracts and
  - You won’t need to customize many aspects
    - Usually no need to customize scoping
  - Most of the work consists in carefully
    - Define a Java API for your runtime
    - Specify mappings in your model inferrer

Small advertising O:-)

- **Xbase is also covered**
- Discount codes (active until June 30)
- 50% for the eBook: IDSX50eBK
- 15% for the print book: IDSX15pBK

Questions? and

THANKS!