Introduction to Xtext

Antonio Natali

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy
{antonio.natali}@unibo.it

Abstract. Introduction to (meta)modelling with Xtext framework (EMF).

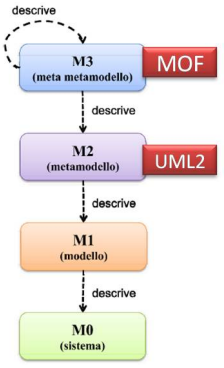
Keywords: Models, meta-models, domain models, EMF, UML, MOF, ECore, Xtext.

1 Introduction

Xtext [3] is a professional *Open-Source* project used in many different industries. It is used in the field of mobile devices, automotive development, embedded systems or Java enterprise software projects and game development, since its main goal is to provide a framework to implement *Domain Specific Languages* (DSL).

Domain specific languages. In *http://en.wikipedia.org/wiki/Domain-specific_language* we read: *In software development and domain engineering, a domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain*[1]*, a particular problem representation technique, and/or a particular solution technique with appropriate IDE support. . The concept isn't new: special-purpose programming languages and all kinds of modelling/specification languages have always existed, but the term has become more popular due to the rise of domain-specific modeling. .* See also *Â§Sviluppo model-centric, pg.205* of [2])

Xtext relies heavily on the *Eclipse Modelling Framework* (EMF) [1], but it can also be used as the serialization back-end of other EMF-based tools. EMF is a Java framework and code generation facility for building tools and other applications based on a structured model, while Ecore is the Eclipse version of *Essential MOF* (EMOF) (see also [2] pg.204).



UML Layers

[1] Such a domain can be more or less anything. The idea is that its concepts and notation is as close as possible to what software designers have in mind when they think about a solution in that domain.

Xtext provides a full implementation of a DSL running on the JVM. However, the compiler components of the language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the EMF, which effectively allows to use the framework together with other EMF frameworks like for instance the *Graphical Modeling Project* (GMF).

1.1 An overvire of EMF and ECore

From the *EMF Developer Guide* in the Eclipse Help we read that:

When talking about modelling, we generally think about things like Class Diagrams, Collaboration Diagrams, State Diagrams, and so on. UML (*Unified Modeling Language*) defines a (the) standard notation for these kinds of diagrams. Using a combination of UML diagrams, a complete model of an application can be specified. This model may be used purely for documentation or, given appropriate tools, it can be used as the input from which to generate part of or, in simple cases, all of an application.

Given that this kind of modeling typically requires expensive ***Object Oriented Analysis and Design*** (OOA/D) tools, you might be questioning our assertion, above, that EMF provides a low cost of entry. The reason we can say that is because an EMF model requires just a small subset of the kinds of things that you can model in UML, specifically simple definitions of the classes and their attributes and relations, for which a full-scale graphical modeling tool is unnecessary.

EMF

Once you specify an EMF model, the EMF generator can create a corresponding set of Java implementation classes. You can edit these generated classes to add methods and instance variables and still regenerate from the model as needed: your additions will be preserved during the regeneration. If the code you added depends on something that you changed in the model, you will still need to update the code to reflect those changes; otherwise, your code is completely unaffected by model changes and regeneration.

In addition to simply increasing your productivity, building your application using EMF provides several other benefits like model change notification, persistence support including default XMI and schema-based XML serialization, a framework for model validation, and a very efficient reflective API for manipulating EMF objects generically. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

EMF started out as an implementation of the OMG (*Object Management Group*) MOF (*Meta Object Facility*) specification but evolved from there based on the experience gained from implementing a large set of tools using it. EMF can be thought of as a highly efficient Java implementation of a core subset of the MOF API. However, to avoid any confusion, the MOF-like core meta model in EMF is called Ecore.

Ecore

ECore is the Eclipse version of EMOF defined in itself (an instance of itself) that allows software designers to build (meta)models by starting from a very small set of concepts (entities):

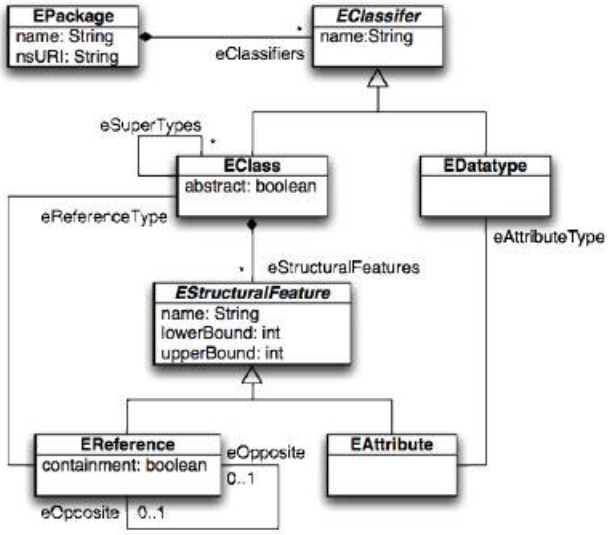
- EPackage: a container of information.
- EClass: a class of objects, with zero or more attribute and references (to other classes).
- EAttribute: an attribute, associated with a name and a type.

– `EReference`: an association or a containment.
– `EDataType`: a type.
– `EENum`: an enumerative data type.

EPackage. An `EPackage` acts as a namespace and container of `EClassifiers`.

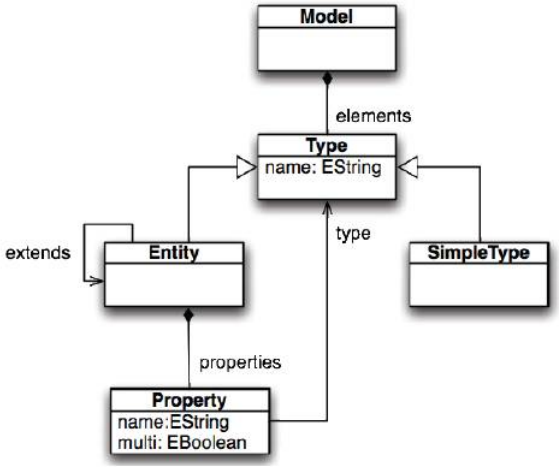
EClasses. The meta model defines the types of the semantic nodes as `Ecore` *EClasses*. An `EClass` can inherit from other `EClasses`. Multiple inheritance is allowed in `Ecore`, but of course cycles are forbidden.

The most relevant concepts of `Ecore` are summarized in the following diagram:



Ecore basic concepts

`EClasses` are shown as boxes in diagram. The following figure that gives an example of a user-define (meta)model:



Sample model diagram

In this example, *Model*, *Type*, *SimpleType*, *Entity*, and *Property* are `EClasses`.

EAttributes and EDataTypes. `EClasses` can have `EAttributes` for their simple properties. The example contains two `EAttributes` name and one `EAttribute` is *multi*.

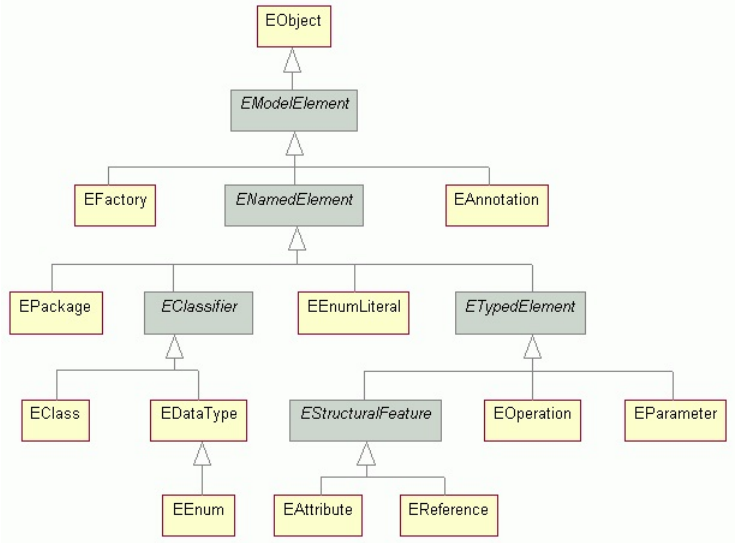
The domain of values for an `EAttribute` is defined by its `EDataType`. `Ecore` ships with some predefined `EDataTypes`, which essentially refer to `Java` primitive types and other immutable classes like *String*. To make a distinction from the Java types, the `EDataTypes` are prefixed with an `E`. In the example, that's `EString` and `EBoolean`. The common supertype of `EDataType` and `EClass` is `EClassifier`.

In contrast to `EAttributes`, `EReferences` point to other `EClasses`. The *containment flag* indicates whether an `EReference` is a containment reference or a cross-reference. The superclass of `EAttibutes` and `EReferences` is `EStructuralFeature` and allows to define a name and a cardinality by setting *lowerBound* and *upperBound*. Setting the latter to `-1` means *unbounded*.

In the diagram, references are edges and containment references are marked with a *diamond*. At the model level, each element can have at most one container, i.e. another element referring to it with a containment reference. This infers a tree structure to the models, as can be seen in the sample model diagram. On the other hand, cross-references refer to elements that can be contained anywhere else. In the example, elements and properties are containment references, while type and extends are cross-references. For reasons of readability, we skipped the crossreferences in the sample model diagram.

Other than associations in `UML`, `EReferences` in `Ecore` are always owned by one `EClasse` and only navigable in the direction form the owner to the type. Bi-directional associations must be modelled as two references, being *eOpposite* of each other and owned by either end of the associations.

The Help of Eclipse reports thecComplete class hierarchy of the `Ecore` model:



Class hierarchy of `Ecore` (shaded boxes are abstract classes)

2 The Xtext framework

In a nutshell, `Xtext` [3] is a workbench to create and work with textual domain-specific languages (`DSLs`). It comes as a feature (set of *plugins*) to the popular Eclipse `IDE`.

2.1 A user-defined meta-model

In Xtext, meta models are either predefined by the user or inferred from a grammar like that described hereunder.

A user-defined meta-model

```
grammar it.unibo.Mydsl with org.eclipse.xtext.common.Terminals
generate mydsl "http://www.unibo.it/example/Mydsl"

DomainModel:
        (elements+=AbstractElement)*;
AbstractElement:
        PackageDeclaration | Type | Import;
Import:
        'import' importedNamespace=QualifiedNameWithWildCard;
PackageDeclaration:
        'package' name=QualifiedName '{'
        (elements+=AbstractElement)*
        '}';
Type:
        Entity | DataType;
DataType:
        'datatype' name=ID;
Entity:
        'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
        '}';
Feature:
        name=ID ':' type=[Type] (multi?='*')?;
QualifiedName:
        ID ('.' ID)*;
QualifiedNameWithWildCard:
        QualifiedName '.*'?;
```

This DSL allows us to define entities like "Person", "Car", "Book", and so on.

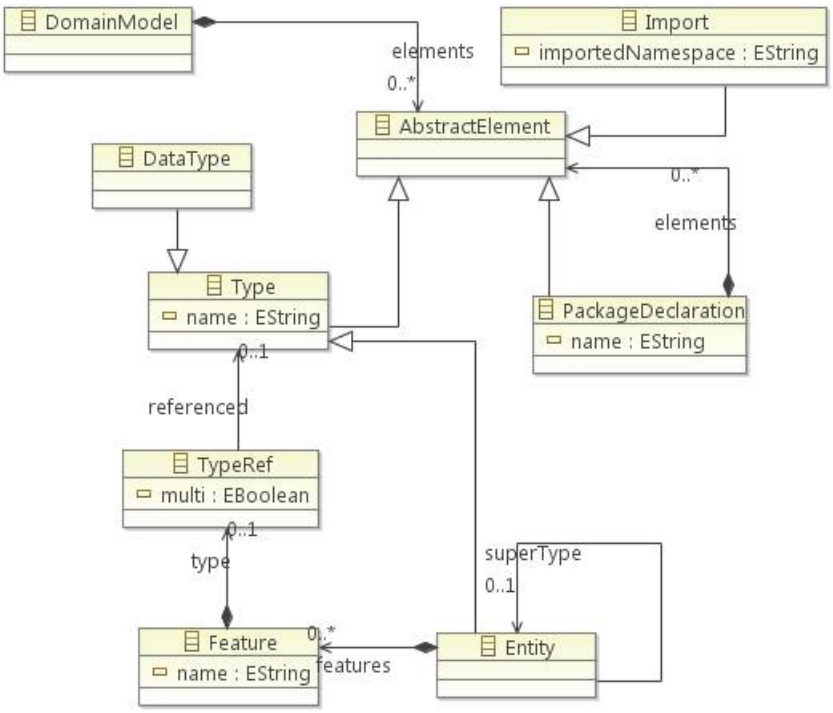
The grammar file is a plain text file with .xtext filename extension, and the grammar within is defined with a BNF-like syntax. While you can use any text editor to modify it, Xtext gives you a specialized editor for grammar files. It is aware of the Xtext language, gives you syntax-coloring, code completion, and more.

2.2 The (Ecore) representation of the (meta)model

Xtext provides a lot of generic implementations for your language's infrastructure but also uses code generation to generate some of the components. Those generated components are for instance the parser, the serializer, the inferred Ecore model (if any) and a couple of convenient base classes for content assist, etc. The generator also contributes to shared project resources such as the plugin.xml, MANIFEST.MF and the Guice modules (see Subsection 2.5). Xtext's generator uses a special DSL called MWE2 - the modelling workflow engine to configure the generator.

From the specification of the grammar, Xtext will produce (by running a mwe2 file, e.g. *GenerateMydsl.mwe2*) the implementation code for the Xtext grammar and configuration. The Xtext code generator is modular and uses so-called *generator fragments* to produce specific parts of the implementation. Which fragments are used and how they are configured can be seen in the *GenerateMydslmodelLanguage.mwe2* file.

The result is a set of files including the `Ecore` representation of the (meta)model (file `Mydsl.ecore`) shown hereunder in graphical form (*Mydsl.ecorediag*):



`Ecore` model of *mydsl* (meta-model)

2.3 Models and AST

The `Ecore` model (or meta model) of a textual language describes the structure of its *Abstract Syntax Tree* (`AST`). `Xtext` uses `Ecore`'s `EPackage` to define `Ecore` models. `Ecore` models are declared to be either inferred (generated) from the grammar or imported. By using the `generate` directive, one tells `Xtext` to derive an `EPackage` from the grammar.

`Xtext` uses `EMF` models as the in-memory representation of any parsed text files. This in-memory object graph is called the *Abstract Syntax Tree* (`AST`). Depending on the community, this concepts is also called *document object graph* (`DOM`), *semantic model*, or simply *model*. `Xtext` uses *model* and `AST` interchangeably.

The `AST` should contain the essence of a model written in textual form according the concrete syntax defined by the grammar, like that reported hereunder:

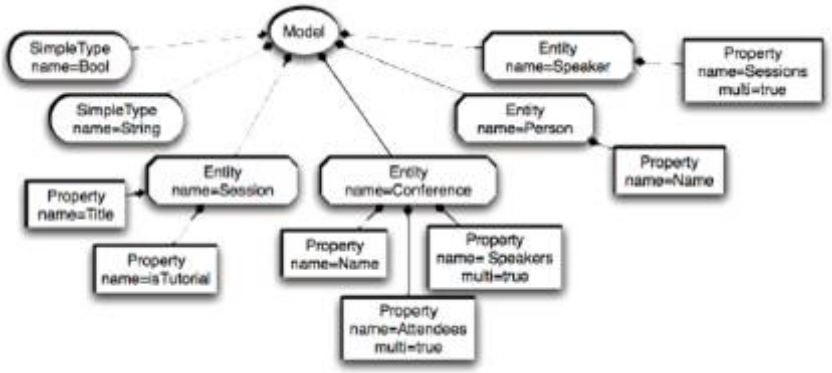
A model in textual form (file *um0.mydsl*)

```
package my.types {
        datatype String
        datatype Boolean
}

package my.entities {
import my.types.*

entity Session {
```

```
        title: String
        isTutorial : Boolean
}
entity Conference {
        name : String
        attendees : Person*
        speakers : Speaker*
}
entity Person {
        name : String
}
entity Speaker extends Person {
        sessions : Session*
}
}
```

The `AST` abstracts over syntactical information, e.g it does not include any keyword (`import`, `package`, {, }, `datatype`, `entity`, `:`, `.`, `.*`, `*`). It is used by later processing steps, such as validation, compilation or interpretation. Since it is an `EMF` model, it is made up of instances of `EObjects` - instance of an `EClass`.

Given the example model above (file *um0.mydsl*), the `AST` looks similar to this:



Sample AST (of the model written in *um0.mydsl*)

Thus, text files, like the model written in *um0.mydsl*, parsed by `Xtext` are represented as object graphs in memory (`AST` or *model*) implemented using the `EMF`, which can be seen as a very powerful version of *JavaBeans*. `EMF` not only provides the typical getter and setter methods for the different features of a model element but also comes with an long list of advanced concepts and semantics, which are extremely useful in the context of `Xtext`.

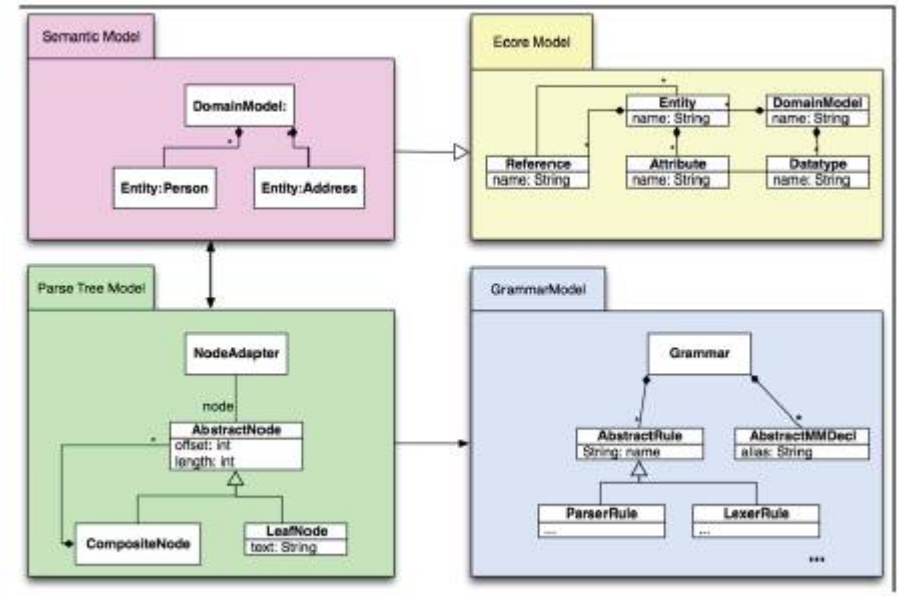
2.4 Parse tree

In many situations the information from the `AST` is sufficient, but in some situations we need additional syntactical information. In `Xtext` not only an `AST` is constructed while parsing but also a so called *parse tree,* which contains all the textual information chunked in so called tokens. The parse tree, also called *node model*, consists of two different kinds of nodes:

- *LeafNodes*, that represent the leafs of the parse tree.
- *CompositeNodes*, that can contain other composite nodes and leaf nodes. It is created for almost each grammar element.

The super type of both node types is `AbstractNode`. One can find a couple of convenience methods in *NodeUtil* and *ParseTreeUtil*.

The parse tree effectively acts as a *tracing model* between the text, the `AST` and the grammar. The following diagram illustrates the four different kinds of models.



Xtext models

2.5 Dependency injection

`Xtext` itself and every language infrastructure developed with `Xtext` is configured and wired-up using dependency injection. All `Xtext` components are assembled by means of *Dependency Injection* (`DI`). This means basically that whenever some code is in need for functionality (or state) from another component, one just declares the dependency rather then stating how to resolve it, i.e. obtaining that component.

The most parts of `Xtext` are implemented as services. A service is an object which implements a certain interface and which is instantiated and provided by `Guice`. Nearly every concept of the `Xtext` framework can be understood as this sort of service: The *XtextEditor*, the *XtextResource*, the *IParser* and even fine grained concepts as the *PrefixMatcher* for the content assist are configured and provided by `Guice`.

`Xtext` may be used in different environments which introduce different constraints. Especially important is the difference between `OSGi` managed containers and plain vanilla `Java` programs. To honor these differences, `Xtext` uses the concept of `ISetup`-implementations in normal mode and uses Eclipse's extension mechanism when it should be configured in an `OSGi` environment.

For each language there is an implementation of `ISetup` generated. It implements a method called *createInjectorAndDoEMFRegistration()*, which can be called to do the initialization of the language infrastructure. This class is intended to be used for runtime and for unit testing, only. The setup method returns an `Injector`, which can further be used to obtain a parser, etc. It also registers the *ResourceFactory* and generated `EPackages` at the respective global registries provided by `EMF`. So basically you can just run the setup and start using `EMF` API to load and store models of your language.

2.6 Validation Rules

One of the main advantages of DSLs is the possibility to statically validate domain specific constraints. The `@Check` annotation advises the framework to use the method as a validation rule.

3 Processing Xtext Models

There are typically two useful things one can do with `Xtext` models: One is translating them to another programming language, i.e. writing a *code generator*, the other is loading them at runtime and use them dynamically.

`Xtext` provides an implementation of `EMF`'s resource, the `XtextResource`. This does not only encapsulate the parser that converts text to an `EMF` model but also the serializer working the opposite direction. That way, an `Xtext` model just looks like any other `Ecore`-based model from the outside, making it amenable for the use by other `EMF` based tools. In fact, the `Xpand` templates in the generator plug-in created by the `Xtext` wizard do not make any assumption on the fact that the model is described in `Xtext`, and they would work fine with any model based on the same `Ecore` model of the language. So in the ideal case, you can switch the serialization format of your models to your self-defined `DSL` by just replacing the resource implementation used by your other modelling tools.

The generator fragment `ResourceFactoryFragment` registers a factory for the `XtextResource` to `EMF`'s resource factory registry, such that all tools using the default mechanism to resolve a resource implementation will automatically get that resource implementation.

Using a self-defined textual syntax as the primary storage format has a number of advantages over the default `XMI` serialization, e.g.

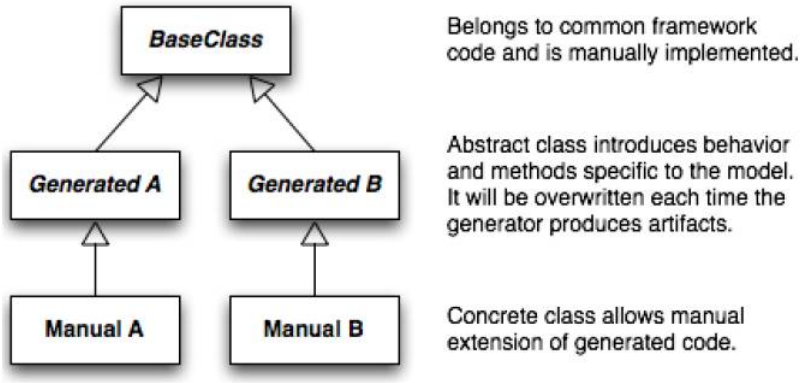
- You can use well-known and easy-to-use tools and techniques for manipulation, such as text editors, regular expressions, or stream editors.
- You can use the same tools for version control as you use for source code. Comparing and merging is performed in a syntax the developer is familiar with.
- It is impossible to break the model such that it cannot be reopened in the editor again.
- Models can be fixed using the same tools, even if they have become incompatible with a new version of the `Ecore` model.

3.1 Code generation (Xtend2)

`Xtext` is based on `EMF` and integrates seamless into this framework. When parsing `DSL` models, an `EMF` metamodel is instantiated, so at runtime there is an object graph that can be processed for code generation or interpretation. The Eclipse Modelling Project has already different frameworks with which you could write code generators like *Xpand*, *Acceleo* and *JET*. All of these frameworks could be used as well, but we want to show a new language that comes with `Xtext 2.0`: `Xtend2`.

`Xtend2` is supposed to be the successor to `M2T Xpand`, which has already a history of more than a decade. All the experience in designing code generation languages along with the `Xtext` tooling have been put into designing `Xtend2`. Its name is `Xtend2`, since `Xtend` is the name of a functional language that was developed for the still popular *openArchitectureWare* framework, and is now part of the `M2T Xpand` framework. For simplicity we will often call it just `"Xtend"`.

`Xtext` follows the *Generation-Gap Pattern*, which basically means that generated and manually written code is completely separated and generated code is produced against a framework (here `Xtext`).



Generation-Gap Pattern

The issue of (code) generation will be addressed in the next section.

3.2 Interact with Xtext models programmatically

`EMF` models can be persisted by the means of a so called *Resource*. `Xtext` languages implement the *Resource interface* which is why you can use the *EMF API* to load a model into memory (and also save them):

1 - Loading and using a model.

The first step initializes the language infrastructure to run in *standalone mode*.

Initialization

```
new DomainmodelStandaloneSetup().createInjectorAndDoEMFRegistration();
```

In fact, `EMF` is designed to work in Eclipse and therefore makes use of *Equinox* extension points in order to register factories and the like. In a vanilla `Java` project there is no Equinox, hence we do the registration programmatically. The generated *MydslStandaloneSetup* class does just that. You don't have to do this kind of initialization when you run your plug-ins within Eclipse, since in that case the extension point declarations are used. The other thing the *StandaloneSetup* takes care of is creating a *Guice injector*.

2 - Create a ResourceSet.

Now that the language infrastructure is initialized and the different contributions to `EMF` are registered, we want to load a Resource. To do so we first create a *ResourceSet*, which as the name suggests represents a set of Resources. If one Resource references another Resource, `EMF` will automatically load that other Resource into the same ResourceSet as soon as the cross-reference is resolved. Resolution of crossreferences is done lazy, i.e. on first access.

ResourceSet

```
ResourceSet rs = new ResourceSetImpl();
```

3 - Loading a resource.

The next step loads the Resource using the resource set. We pass in a `URI` which points to the file in the file system.

Resource

```
Resource resource =
        rs.getResource(URI.createURI("./um0.mydsl"), true);
```

EMF's URI is a powerful concept. It supports a lot of different schemes to load resources from file system, web sites, jars, OSGi bundles or even from Java's classpath. And if that is not enough you can come up with your own schemes. Also a URI can not only point to a resource but also to any EObject in a resource. This is done by appending a so called *URI fragment* to the URI. The second parameter denotes whether the resource should automatically be loaded if it wasn't already before. Alternatively we could have written

Resource

```
Resource resource =
        rs.getResource(URI.createURI("./um0.mydsl"), false);
resource.load(null);
```

The *load* method optionally takes a map of properties, which allows to define a contract between a client and the specific implementation. In Xtext, for instance, we use the map to state whether cross-references should be eagerly resolved. In order to find out what properties are supported, it's usually best to look into the concrete implementations. That said, in most cases you don?t need to pass any properties at all.

4 - Referencing the root.
The last step assigns the root element of the model to a local variable.

```
EObject eobject = resource.getContents().get(0);
```

Actually it is the first element from the contents list of a Resource, but in Xtext a Resource always has just one root element.

Working with the Grammar.
Also the grammar is represented as an EMF model and can be used in Java. In fact, each node of the node model references the element from the grammar which was responsible for parsing or lexing that node:

Working with the grammar

```
DomainModel domainModel = (DomainModel) eObject;
CompositeNode node = NodeUtil.getNode(domainModel);
ParserRule parserRule = (ParserRule) node.getGrammarElement();
assertEquals("DomainModel", parserRule.getName());
```

3.3 Code generation

Xtext follows the *Generation-Gap Pattern*, which basically means that generated and manually written code is completely separated and generated code is produced against a framework. Files that are regenerated by the Xtext framework are placed in the src-gen folder of the projects, and files that are meant to be manually maintained are in the src folder.

There are two main approaches that code generation frameworks: *Template based* and *Visitor based*. In practice, template based approaches often scale better for large-scale projects. Not because of their performance, more because they are easier to understand, and thus better to communicate and maintain. Xtend2 offers such a template based approach, which is called *Rich Strings*. Unlike its predecessor, Xpand, and most other frameworks, there is no gap between template language and integration of generator logic code. All is done with Xtend2 and the expression language behind, Xbase, with full integration of the Java language. Rich Strings are introduced by 3 single quotes ’’’.

3.4 Dispatch.

Note that, while `Java` only support so-called *Single Dispatch*, `Xtend2` supports *Polymorphic Dispatch*[2] by means of the keyword `dispatch`. Dispatch functions make a set of overloaded functions polymorphic. That is the run-time types of all given arguments are used to decide which of the overloaded methods is being invoked. This essentially removes the need for the quite invasive visitor pattern.

References

1. EMF. EMF home page.
   http://www.eclipse.org/modeling/emf/.
2. A. Natali and A. Molesini. *Costruire sistemi software: dai modelli al codice*. Esculapio, 2009.
3. Xtext. Xtext home page.
   http://www.eclipse.org/Xtext/.

[2] see *http://en.wikipedia.org/wiki/Dynamic_dispatch#Single_and_multiple_dispatch*)