Implementing Domain-Specific Languages with Xtext and Xtend

Lorenzo Bettini

Chapter No. 4
"Validation"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.4 "Validation"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Lorenzo Bettini is an assistant professor (Researcher) in computer science at Dipartimento di Informatica, Università di Torino, Italy. Previously, he was a Postdoc and a contractual researcher at Dipartimento di Sistemi e Informatica, Università di Firenze, Italy.

He has a Masters Degree in computer science and a PhD in theoretical computer science.

His research interests cover design, theory, and the implementation of programming languages (in particular, object-oriented languages and network-aware languages).

He has been using Xtext since version 0.7. He has used Xtext and Xtend for implementing many Domain Specific Languages and Java-like programming languages.

He is also the author of about 60 papers published in international conferences and international journals.

You can contact him at http://www.lorenzobettini.it.

For More Information:
Implementing Domain-Specific Languages with Xtext and Xtend

Xtext is an open source Eclipse framework for implementing Domain Specific Languages together with their integration in the Eclipse IDE. Xtext lets you implement languages quickly by covering all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a full Eclipse IDE integration (with all the typical IDE features such as editor with syntax highlighting, code completion, error markers, automatic build infrastructure, and so on).

This book will incrementally guide you through the very basics of DSL implementation with Xtext and Xtend, such as grammar definition, validation, and code generation; the book will then cover advanced concepts such as unit testing, type checking, and scoping. Xtext comes with good and smart default implementations for all these aspects. However, every single aspect can be customized by the programmer.

Although Java can be used for customizing the implementation of a DSL, Xtext fosters the use of Xtend, a Java-like programming language completely interoperable with the Java type system which features a more compact and easier to use syntax and advanced features such as type inference and lambda expressions. For this reason, we will use Xtend throughout the book.

Most of the chapters have a tutorial nature and will describe the main concepts of Xtext through uncomplicated examples. The book also uses test driven development extensively.

This book aims at being complementary to the official documentation, trying to give you enough information to start being productive in implementing a DSL with Xtext. This book will try to teach you some methodologies and best practices when using Xtext, filling some bits of information that are not present in the official documentation.

The chapters are meant to be read in order, since they typically refer to concepts that were introduced in the previous chapters.

All the examples shown in the book are available online, see the section Downloading the example code. We strongly suggest that you first try to develop the examples while reading the chapters and then compare their implementations with the ones provided by the author.

For More Information:
What This Book Covers

After a small introduction to the features that a DSL implementation should cover (including integration in an IDE), the book will introduce Xtend since it will be used in all the examples. The book proceeds by explaining the main concepts of Xtext; for example, validation, code generation, and customizations of runtime and UI aspects. The book will then show how to test a DSL implemented in Xtext with Junit in order to follow a Test Driven Development strategy that will help you to quickly implement cleaner and more maintainable code. The test-driven approach is used in the rest of the book when presenting advanced concepts such as type checking and Scoping. The book also shows how to build and release a DSL so that it can be installed in Eclipse and hints on how to build the DSL headlessly in a continuous integration server. The last chapter briefly introduces Xbase.

Chapter 1, Implementing a DSL, gives a brief introduction to Domain Specific Languages (DSL) and sketches the main tasks for implementing a DSL and its integration in an IDE. The chapter also shows how to install Xtext and gives a first idea of what you can do with Xtext.

Chapter 2, Creating Your First Xtext Language, shows a first example of a DSL implemented with Xtext and gives an introduction to some features of the Xtext grammar language. The chapter describes the typical development workflow of programming with Xtext and provides a small introduction to EMF (Eclipse Modeling Framework), a framework on which Xtext is based.

Chapter 3, The Xtend Programming Language, describes the main features of the Xtend programming language, a Java-like language interoperable with Java. We will use Xtend to implement every aspect of an Xtext DSL.

Chapter 4, Validation, describes validation, in particular, the Xtext mechanism to implement validation, that is, the validator. This chapter is about implementing additional constraint checks that cannot be done at parsing time. It also shows how to implement quickfixes corresponding to the errors generated by the validator.

Chapter 5, Code Generation, shows how to write a code generator for an Xtext DSL using the Xtend programming language. The chapter also shows how a DSL implementation can be exported as a Java standalone command-line compiler.

Chapter 6, Customizations, describes the main mechanism for customizing Xtext components, Google Guice, a Dependency Injection framework. In particular, the chapter shows how to customize both the runtime and the UI aspects of an Xtext DSL.

Chapter 7, Testing, describes how to test a DSL implementation using Junit and the additional utility classes provided by Xtext. The chapter shows the typical techniques for testing both the runtime and the UI aspects of a DSL implemented in Xtext.

For More Information:
Chapter 8, *An Expression Language*, covers the implementation of a DSL for expressions, including arithmetic, boolean, and string expressions. The chapter shows how to deal with recursive rules and with typical problems when writing Xtext grammars. The implementation will be described incrementally and in a test-driven way. The chapter also shows how to implement a type system for checking that expressions are correct with respect to types and how to implement an interpreter for these expressions.

Chapter 9, *Type Checking*, covers the implementation of a small object-oriented DSL, which can be seen as a smaller version of Java that we call SmallJava. This chapter shows how to implement some type checking techniques that deal with object-oriented features, such as inheritance and subtyping.

Chapter 10, *Scoping*, covers the main mechanism behind visibility and cross-reference resolution in Xtext. Since scoping and typing are often strictly connected and interdependent especially for object-oriented languages, the chapter is based on the SmallJava DSL introduced in the previous chapter. The chapter describes both local and global scoping and how to customize them.

Chapter 11, *Building and Releasing*, describes how you can release your DSL implementation by creating an Eclipse p2 repository so that others can easily install it in Eclipse. The chapter is based on the Xtext wizard that creates the infrastructure to build a p2 repository with Buckminster. The wizard will also create all the needed files to build your projects and test them in a headless way so that you can easily run your builds in a continuous integration server.

Chapter 12, *Xbase*, describes Xbase a reusable expression language interoperable with Java. By using Xbase in your DSL, you will inherit all the Xbase mechanisms for performing type checking according to the Java type system and the automatic Java code generation.

For More Information:
In this chapter we will introduce the concept of validation, and in particular, the Xtext mechanism to implement validation: the validator. With validation you can implement additional constraint checks of a DSL which cannot be done at parsing time. Xtext allows you to implement such constraint checks in an easy and declarative way; furthermore, you only need to communicate to Xtext the possible errors or warnings and it will take care of generating the error markers accordingly in the IDE. The validation will take place in the background while the user of the DSL is typing in the editor, so that an immediate feedback is provided. We will also show how to implement quickfixes corresponding to the errors and warnings generated during validation, so that we can help the user to solve problems due to validation errors.

Validation in Xtext

As we hinted in Chapter 1, Implementing a DSL, parsing a program is only the first stage in a programming language implementation. In particular, the overall correctness of a program cannot always be determined during parsing. Trying to embed additional constraint checks in the grammar specification could either make such specification more complex or it could be simply impossible as some additional static analysis can be performed only when other program parts are already parsed.

Actually, the best practice is to do as little as possible in the grammar and as much as possible in validation (we will use this practice in Chapter 9, Type Checking and Chapter 10, Scoping). This is because it is possible to provide far better error messages and to more precisely detect problems that are eligible for quickfixes.

The mechanism of validation will be used extensively in all example DSLs of this book. Typically, even for small DSLs, a validator has to be implemented to perform additional constraint checks.

For More Information:
In Xtext, these constraints checks are implemented using a validator, which is a concept inherited from the corresponding EMF API (see Steinberg et al., 2008). In EMF, you can implement a validator which performs constraint checks on the elements of an EMF model. Since Xtext uses EMF for representing the AST of a parsed program, the mechanism of the validator naturally extends to an Xtext DSL. In particular, Xtext enhances the EMF API for validation, by providing a declarative way to specify rules for constraints of your DSL. Moreover, Xtext comes with default validators, some of which are enabled by default, to perform checks which are common to many DSLs (for example, cross-reference checks). Your custom validator can be composed with the default ones of Xtext.

**Default validators**

Let us go back to the Entities DSL of Chapter 2, Creating your First Xtext Language. Since we expressed cross-references in our Entities grammar, we can see the Xtext linker/validator in action in the generated editor. If we enter an incorrect reference (for example, the name of a super entity that does not exist), we get the error "Couldn't resolve reference to...". This check on cross-references is performed by one of the default validators provided by Xtext (cross-reference resolution is the main subject of Chapter 10, Scoping).

Another standard validator provided by Xtext is the one that checks whether the names are unique within your program. This check validates that names are unique per element type; for example, you can have an attribute named the same as an entity, but not two entities with the same name.

A unique name validator is not enabled by default, but it can be turned on by modifying the MWE2 file as shown in the following code snippet. In our example, it is GenerateEntities.mwe2; in particular, we need to uncomment the composedCheck specification which concerns NamesAreUniqueValidator:

```plaintext
fragment = validation.ValidatorFragment auto-inject {
    composedCheck =
        "org.eclipse.xtext.validation.NamesAreUniqueValidator"
}
```

After that, of course, you need to run the MWE2 workflow.

For More Information:
If you now try to declare two entities with the same name in the Entities DSL editor, you will get an error as shown in the following screenshot:

![Screenshot showing an error message for duplicate entity names](image)

It is interesting to note that, besides the element type, this validator also takes into consideration the containment relations, for example, two attributes declared in two different entities are allowed to have the same name (as you can see from the preceding screenshot, both \texttt{A} and \texttt{B} have the attribute \texttt{s}, and this is allowed).

Technically, everything that is referenceable via name is named in a namespace, implied by the containment relation. This leads to qualified names, which will be explained in Chapter 10, Scoping.

The default behavior of this validator should suit most DSLs. If your DSL needs to have more rigid constraints about names, or in general about duplicate elements, you will have to implement a customized \texttt{NamesAreUniqueValidator} class or simply disable \texttt{NamesAreUniqueValidator} and implement these name checks in your own validator (an example of a custom duplicate name check is shown in Chapter 9, Type Checking).

For More Information:
Validation

Custom validators

While the default validators can perform some common validation tasks, most of the checks for your DSL will have to be implemented by you, according to the semantics you want your DSL to have.

These additional checks can be implemented using the Xtend class that Xtext has generated for you in the validation subpackage in the src folder of the runtime plug-in project. In our example, this class is called EntitiesValidator. Remember that since this class is in the src folder, it will not be overwritten by future MWE2 workflow executions. Xtext performs validation by invoking each method annotated with @Check, passing all instances having a compatible runtime type to each such method. The name of the method is not important, but the type of the single parameter is important. You can define as many annotated methods as you want for the same type; Xtext will invoke them all. Inside such methods you can implement the semantic checks for that element. If a semantic check fails, you can call the error method, which will be explained shortly.

For example, we want to make sure that there is no cycle in the hierarchy of an entity; thus, we write the following annotated method in our validator:

```java
class EntitiesValidator extends AbstractEntitiesValidator {
    @Check
def checkNoCycleInEntityHierarchy(Entity entity) {
        if (entity.superType == null)
            return // nothing to check
        val visitedEntities = <Entity>newHashSet()
        visitedEntities.add(entity)
        var current = entity.superType
        while (current != null) {
            if (visitedEntities.contains(current)) {
                error("cycle in hierarchy of entity "+current.name+",
                     EntitiesPackage::eINSTANCE.entity_SuperType)
                return
            }
            visitedEntities.add(current)
            current = current.superType
        }
    }
}
```

In the preceding method we traverse the hierarchy of an entity (of course, if an entity has no superType, there is nothing to check) by recording all the entities we are visiting. If during this visit we find an entity that we have already visited it means that the hierarchy contains a cycle and we issue an error. It is crucial to leave the while loop in that case, otherwise the loop will never end (after all, we found a cycle and we would traverse the hierarchy endlessly).

For More Information:
The method error (we refer to Xtext documentation for further details) has many overloaded versions. In this example, we use the version that requires the following:

- A message for the error (and it is up to you to provide meaningful information).
- The EMF feature of the examined EObject which the error should be reported against, that is, which should be marked with error. In this case, the feature containing the error is the superType feature.

Access to classes and features are obtained from the EPackage class that is generated for our DSL's metamodel (in our example, EntitiesPackage). Using this EPackage, EMF features can be obtained in two ways:

- Using the static instance of the package and then the method corresponding to the feature (as we did in the preceding code):
  
  ```java
  EntitiesPackage::eINSTANCE.entity_SuperType.
  ```

- Using the static fields of the inner interface Literals:
  
  ```java
  EntitiesPackage$Literals::ENTITY__SUPER_TYPE.
  ```

In both cases, in your Xtend programs, you can rely on the content assist to select the feature easily.

We can now try the above validation check in the Entities DSL editor by defining entities which contain a cycle in the hierarchy, as shown in the following screenshot:

![Screenshot of the Entities DSL editor showing entities A, B, and C with a cycle error]

In the screenshot, there are three errors indicated:
- A cycle in hierarchy of entity 'A'
- A cycle in hierarchy of entity 'B'
- A cycle in hierarchy of entity 'C'

For More Information:
You can see that the highlighted element in the editor is the entity name after the keyword `extends`, since that corresponds to the feature `superType`.

The three error markers also show that Xtext calls our `@Check` annotated method for all the elements of type `Entity` in the program.

Calling the `error` method with the appropriate information will let Xtext manage the markers for Xtext based resources (clearing them before reparsing, keeping track of dirty versus saved state, and so on). Markers will appear wherever they are supported in the IDE: in the right and left editor ruler, in the Problems view, and in the package explorer.

If you want to issue warnings instead of errors (which are considered to mean that the model is invalid), simply call the `warning` method that has the same signature as the `error` method. For example, in our Entities DSL, we follow a standard convention about names: the name of an entity should start with a capital letter, while the name of an attribute should be lowercase. If the user does not follow this convention, we issue a warning; the program is considered valid anyway. To implement this, we write the following methods in the `EntitiesValidator` class (note the use of imported static methods as extension methods from the class `Character`):

```java
import static extension java.lang.Character.*

class EntitiesValidator extends AbstractEntitiesValidator {
    ...  
    @Check
def checkEntityNameStartsWithCapital(Entity entity) {
        if (entity.name.charAt(0).lowerCase)
            warning("Entity name should start with a capital",
                    EntitiesPackage::eINSTANCE.entity_Name)
    }
    
    @Check
def checkAttributeNameStartsWithLowercase(Attribute attr) {
        if (attr.name.charAt(0).upperCase)
            warning("Attribute name should start with a lowercase",
                    EntitiesPackage::eINSTANCE.attribute_Name)
    }
}
```

For More Information:
The following screenshot shows how warning markers are created instead of error markers:

In Xtext 2.4 an “info” severity level has been added (the method to call is info). In this case an information marker is shown only in the editor’s ruler, while the corresponding file is not marked.

This is just an example of a simple validator implementation; in the rest of the book, we will see many other implementations that perform more complex constraint checks (among which, type checking, as shown in Chapter 8, An Expression Language and in Chapter 9, Type Checking).

Quickfixes
As we said in Chapter 1, Implementing a DSL, a quickfix is a proposal to solve a problem in a program. Quickfixes are typically implemented by a context menu available from the error marker, and they are available both in the editor ruler and in the Problems view.

Since quickfixes are tightly connected to validation, we describe them in this chapter. Moreover, they allow us to get familiar with the manipulation of the EMF model representing the AST of a program.

For More Information:
Validation

In our Entities DSL we can provide a quickfix for each warning and error issued by our validator; moreover, as we will see later, we can also provide quickfixes for errors issued by Xtext default validators.

Xtext provides an easy mechanism to implement a quickfix connected to an error or warning issued by a validator. The Xtext generator generates an Xtend stub class for quickfixes into the UI plug-in project; in our Entities DSL example, this class is org.example.entities.ui.quickfix.EntitiesQuickfixProvider.

A quickfix is triggered by an issue code associated with an error or warning marker. An issue code is simply a string that uniquely identifies the issue. Thus, when invoking the error or warning method, we must provide an additional argument which represents the issue code. In the validator, this is typically done by defining a public String constant, whose value is prefixed with the package name of the DSL and ends with a sensible name for the issue. It might also make sense to pass additional issue data that can be reused by the quickfix provider to show a more meaningful description of the quickfix and to actually fix the program. It is worth noting that issue data is very useful when validation needs to compute something that is costly; the quickfix may then avoid having to compute it again. Thus, we use another version of the method error and warning which takes four arguments, and we modify our validator as follows (only the modified parts are shown):

```java
class EntitiesValidator extends AbstractEntitiesValidator {

    public static val HIERARCHY_CYCLE = "org.example.entities.HierarchyCycle";
    public static val INVALID_ENTITY_NAME = "org.example.entities.InvalidEntityName";
    public static val INVALID_ATTRIBUTE_NAME = "org.example.entities.InvalidAttributeName";

    @Check
def checkNoCycleInEntityHierarchy(Entity entity) {
        ...
        error("cycle in hierarchy of entity "+current.name+"",
          EntitiesPackage::eINSTANCE.entity_SuperType,
          HIERARCHY_CYCLE,
          current.superType.name)
        ...
    }

    @Check
def checkEntityNameStartsWithCapital(Entity entity) {
        if (entity.name.charAt(0).toLowerCase)
        ...
    }
}
```

For More Information:
warning("Entity name should start with a capital letter",
    EntitiesPackage::eINSTANCE.entity_Name,
    INVALID_ENTITY_NAME,
    entity.name)
}

@Check
def checkAttributeNameStartsWithLowercase(Attribute attr) {
    if (attr.name.charAt(0).toUpperCase)
        warning("Attribute name should start with a lowercase",
            EntitiesPackage::eINSTANCE.attribute_Name,
            INVALID_ATTRIBUTE_NAME,
            attr.name)
}

The issue constant is passed as the third argument to the methods error and warning. Issue data is optional and you can pass a variable number of issue data arguments. To implement a quickfix, we define a method in EntitiesQuickfixProvider annotated with @Fix and a reference to the issue code this quickfix applies to. The name of the method is not important, but the parameter types are fixed.

For example, for the warning concerning the first letter of an entity name, which must be capital, we implement a quickfix which automatically capitalizes the first letter of that entity:

```java
class EntitiesQuickfixProvider extends DefaultQuickfixProvider {
    @Fix(EntitiesValidator::INVALID_ENTITY_NAME)
    def void capitalizeEntityNameFirstLetter(Issue issue, IssueResolutionAcceptor acceptor) {
        acceptor.accept(issue,
            "Capitalize first letter", // label
            "Capitalize first letter of ", // description
            "Entity.gif", // icon
            [context |
                val xtextDocument = context.xtextDocument
                val firstLetter = xtextDocument.get(issue.offset, 1);
                xtextDocument.replace(issue.offset, 1,
                    firstLetter.toFirstUpper);
            ]);}
}
```

For More Information:
Let us analyze what this code does. The first parameter of a quickfix provider method is the Issue object that represents the error information; this is built internally by Xtext using the information passed to error or warning in your validator. The second parameter is an acceptor. Acceptor is a concept, and you will see different types of acceptors used in many places in Xtext; you usually only have to invoke the method accept on an acceptor, passing some arguments.

The first three arguments passed to the method accept of the acceptor are the label (shown in the quickfix pop up for this fix), a description (which should show what the effect of selecting this quick fix would mean, or something that makes the user confident it is a fix they want to apply), and an icon (if you do not want an icon, you can pass an empty string; how to use custom icons in your DSL UI will be explained in Chapter 6, Customizations). Note that for the description, we use the first element of the issue data (an array) since we know that in the validator we passed the name of the supertype as the single issue data (when implementing the quickfix you must be consistent with the information passed by the validator).

The fourth argument is the most interesting one, since it is actually the modification code performed by the quickfix when selected by the user. In Java, the fourth argument would be an inner class, but in Xtend, we can pass a lambda. Quickfixes can perform the correction based on the source text (textual modification), or on the model (semantic modification). These are explained in the next two sections.

**Textual modification**

You can specify a lambda which takes a single parameter of the type IModificationContext. Due to the powerful type inference mechanisms of Xtend, it is enough to just specify the name of the parameter (and Xtend will infer its type). We could have also specified no parameter at all, since by default, lambdas have automatically one parameter named it. In our example, the name of the parameter was explicitly stated for clarity.

In the preceding code, we use the IDocument argument, which is passed in the given modification context, to get access to the text we want to modify in our quickfix. We have been given the offset and length of where the error/warning is marked in the Issue object. We can now use the document methods get(offset, length) and replace() to perform the capitalization of the first letter.

For More Information:
This quickfix is shown in the following screenshot.

![Screenshot of quickfix in Xtext editor](image)

Using this strategy for implementing quickfixes has the drawback that we need to deal with the actual text of the editor.

**Model modification**

The alternative strategy relies on the fact that the program is also available in memory as an EMF model: if we modify the model, the Xtext editor will automatically update its contents. For this reason, we can specify a lambda which takes two parameters: the EMF element (that is, the `EObject`) that contains the error, and the modification context. The `EObject` element is the one the warning was reported against.

For instance, to uncapitalize the first letter of an attribute, we can write the quickfix using the following strategy:

```java
@Fix(EntitiesValidator::INVALID_ATTRIBUTE_NAME)
def void uncapitalizeAttributeNameFirstLetter(Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue,
        "Uncapitalize first letter", // label
        "Uncapitalize first letter of ", // description
        "Attribute.gif", // icon
        [element, context | (element as Attribute).name = issue.data.get(0).toFirstLower ]
    );
}
```

For More Information:
Validation

In this case, the element is the `Attribute` object against which the warning was reported. Therefore, we simply assign the fixed name to the name of the attribute. Note that with this strategy, we only manipulate the EMF model, without having to deal with the contents of the editor: Xtext will then take care of updating the editor's contents.

The ability to directly modify the EMF model of the program makes more complex quickfixes easier to implement. For example, if we want to implement the quickfix to remove the supertype of the entity which contains a cycle in the hierarchy, we just need to set the `superType` feature to null, as shown in the following code snippet (you can see the quickfix in the following screenshot):

```java
@Fix(EntitiesValidator::HIERARCHY_CYCLE)
void removeSuperType(Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue,
        "Remove supertype",
        "Remove supertype '!issue.data.get(0)'!",
        "delete_obj.gif",
        [ element, context |
            (element as Entity).superType = null;
        ]
    );
}
```

For More Information:
Note that the semantic change results in there not being any supertype in the model element and thus the source text extends is also removed. Implementing the same quickfix by modifying the text of the program would require more effort and would be more error prone. On the other hand, textual modifications allow fixing things that are not present in the semantic model. We could, for example, allow some punctuation to be optional in the grammar in order to provide better error messages such as "missing comma" as opposed to "syntax error". Such lenient behavior makes it possible for the parser to produce a model, and there is something semantic to base validation as opposed to a syntax error where we get no model at all. Also, semantic changes always include formatting, and this may have other unwanted side effects (we will deal with formatting in Chapter 6, Customizations).

Quickfixes for default validators

As mentioned before, we can also provide quickfixes for errors issued by default Xtext validators. You might have noted that if you refer to a missing entity in the Entities DSL editor, Xtext already proposes some quickfixes: if there are other entities in the source file, it proposes to change the reference to one of the existing entities. We can provide an additional quickfix that proposes to automatically add the missing entity. In order to do this, we must define a method in our quickfix provider for the issue org.eclipse.xtext.diagnostics.Diagnostic.LINKING_DIAGNOSTIC. We want to add the missing entity to the current model; to make things more interesting, the quickfix should add the missing entity after the entity which refers to the missing entity. For example, consider the following source file:

```xtend
entity MyFirstEntity {
    FooBar s;
    int[] a;
}

entity MyOtherEntity {
}
```

The referred `FooBar` entity in the attribute definition is not defined in the program, and we would like to add it after the definition of `MyFirstEntity` (and before `MyOtherEntity`), since that is the entity which contains the attribute definition referring to the missing entity.

Let's first present the code for this quickfix (we will use Xtend features for getters, setters, template strings, and the with operator to make our logic more compact):

```xtend
import static extension org.eclipse.xtext.EcoreUtil2.*

@Fix(Diagnostic::LINKING_DIAGNOSTIC)
```

For More Information:

def void createMissingEntity(Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue,
        "Create missing entity",
        "Create missing entity",
        "Entity.gif",
        [ element, context |
            val currentEntity =
                element.eGetContainerOfType(typeof(Entity))
            val model = currentEntity.eContainer as Model
            model.entities.add(
                model.entities.indexOf(currentEntity)+1,
                EntitiesFactory::eINSTANCE.createEntity() => [
                    name = context.xtextDocument.get(issue.offset, issue.length)
                ]
            )
        ]
    )
}

Consider that the EObject element passed to the lambda is the program element which refers to the missing entity, thus, it is not necessarily an Entity (for instance, if the missing entity is in an attribute’s type specification, as in the preceding Entities program snippet, then the EObject element is an AttributeType). To get the containing entity, we could walk up the containment relation of the EMF model till we get to an Entity element. Alternatively, we can use one of the many static utility methods, here imported as extension methods, provided by Xtext in the class EcoreUtil2 (this complements the standard EcoreUtil class of EMF). In particular, we use getContainerOfType, which does this walking up in the containment relation for us, until it finds an element of the specified type. For retrieving the root Model element we can simply cast the container of the found entity (since in our Entities DSL, an Entity can only be contained in a Model). Then, we insert the newly created entity in the desired position (that is, right after the position of the current entity).

Spend some time to take a look at the classes EcoreUtil and EcoreUtil2, since they provide many useful methods you will need when dealing with an EMF model.

For More Information:
To create the missing entity, we must know its name. For this issue (which is not generated by our own validator), the issue data does not contain any information about the missing element’s name. However, the issue offset tells us in which position in the document the missing element’s name is referred. Thus, the name of the missing element can be retrieved using this offset (the length is also contained in the issue) from the editor’s document.

You can now check what this quickfix does (see the following screenshot; this also shows the default quickfixes provided by Xtext which propose to change the name of the referred entity to one available in the current source).

Summary

In this chapter you learned how to implement constraint checks, using the Xtext validator mechanism based on @Check annotated methods. Just by implementing a custom validator and calling the method error or warning with the appropriate information, Xtext produces error and warning markers that result in marking the text regions as well as showing the markers in the various views in Eclipse.

We also showed how to implement quickfixes. Since Xtext automatically synchronizes the DSL editor's contents with the EMF model of the AST, we can simply modify such model without dealing with the textual representation of the program.

In the next chapter we will write a code generator for the Entities DSL implemented in Xtend, relying on its advanced features for code generation: starting from a program written in our Entities DSL, we will generate the corresponding Java code. You will see that Xtext automatically integrates your code generator into the building infrastructure of Eclipse.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.