An Eclipse-based IDE for Featherweight Java implemented in Xtext *

Lorenzo Bettini

Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy

Abstract. We present our implementation of Featherweight Java (a lightweight version of Java which is typically used when formalizing Java-like languages) in Eclipse by relying on Xtext (a framework for development of programming languages in Eclipse). Xtext eases the task of implementing a compiler and an IDE based on Eclipse by providing a high-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse, and allows the programmer to customize every aspect.

1 Introduction

Implementing a programming language, even if only on a prototypical form, is always time consuming: writing the parser using a compiler generator (like, e.g., Flex & Bison [20] and ANTLR [21]), building the abstract syntax tree (AST), perform all the visits on the AST (e.g., type checking), and finally generate code into a target language or build the interpreter. Moreover, having an integrated development environment (IDE) for the language is something that nowadays programmers are used to. Mechanisms like code completion, program outline, syntax highlighting, and so on, are typical of many professional programming editors, and surely increment productivity. Building an IDE from scratch for a language makes a little sense, since it would require to reimplement many recurrent functionalities, thus it is always better to rely on an existing one.

With this respect, Eclipse provides an extensible and powerful framework for building programming language editors, covering all the aspects of professional IDEs for enhancing productivity. However, the procedure for building a plugin for your own programming language for Eclipse is still quite laborious and requires a lot of manual programming. XTEXT [5], a framework for development of programming languages as well as other domain-specific languages (DSLs), eases this task by providing a high-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse. The plugins generated by XTEXT already implement most of the recurrent artifacts for a language IDE, and they can be easily customized by “injecting” (relying on Google-Guice [1]) our own specific language mechanisms implementations. In particular, the code we need to write for customizing the IDE is minimal.

In this paper, we present the prototypical implementation of FEATHERWEIGHT JAVA (FJ) [19, 23], a lightweight functional version of Java, which focuses on a few

* Work partially supported by MIUR (PRIN 2009 DISCO).
Listing 1: An example of FJ program.

class A extends Object { }
class B extends Object { }

class Pair extends Object {
    Object fst; Object snd;
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }
    Pair setsnd(Object newscd) {
        return new Pair(this.fst, newscd);
    }
}

new Pair(new A(), new B()).setfst(new A()).fst

basic features: mutually recursive class definitions, inheritance, object creation, method
invocation, method recursion through this, subtyping and field access. In particular,
a FJ program is a list of class definitions and a single main expression (see the exam-
ple in Listing 1). Since in FJ the class constructor has a fixed shape, we simplified the
language by assuming such constructors as implicit. The minimal syntax, typing and
semantics make the type safety proof simple and compact, in such a way that FJ is a
handy tool for studying the consequences of extensions and variations with respect to
Java (“FJ’s main application is modeling extensions of Java”, [23], page 248). For this
reason, we personally used FJ as the starting point for studying language extensions to
Java [10, 9, 7, 11, 8] and we felt the need of a rapid development framework for building
editors and compilers for our languages.

This paper can be seen as an experience report on using XTEXT for building a
fully-fledged Eclipse-based IDE for a general purpose language; although FJ is not
a complete object-oriented programming language (like Java itself), still it deals with
typical object-oriented features, thus it still requires a wide effort during the imple-
mentation. Furthermore, while XTEXT defines itself as a framework mainly for do-
main specific languages (DSL), this paper also provides an evidence for XTEXT use
for general purpose languages (with this respect, XTEXT only ships with DSL ex-
amples). The implementation of FJ is available as an open source project at http://fj-
eclipse.sourceforge.net.

2 Implementing FJ with XTEXT

In this section, we describe the implementation of FJ using the XTEXT [5] framework
for Eclipse. We will not provide all the details of the implementation; Instead, we will
stress what we need to provide to XTEXT (i.e., specific class and method implemen-
tations) so that the framework can do the rest of the job in implementing all the IDE
functionalities.

Syntax and Parsing. The first task in XTEXT is to write the grammar of the lan-
guage using an EBNF-like syntax. For completeness we show the complete XTEXT
grammar for FJ in Listing 2. The reader who is familiar with ANTLR [21] will note
that the syntax of XTEXT grammars is very similar to ANTLR’s syntax, though a little
bit simpler.
Starting from this grammar, XTEXT generates an ANTLR parser [21]. The generation of the abstract syntax tree is handled by XTEXT as well. In particular, during parsing, the AST is automatically generated in the shape of an EMF model (Eclipse Modeling Framework [24]). Thus, the manipulation of the AST can use all mechanisms provided by EMF itself. Note that the automatic generation of the AST by XTEXT already reduces the programmer’s job a lot; furthermore, by relying on a modeling framework such as EMF, the programmer does not even have to create all the class hierarchy for modeling the AST. In particular, XTEXT connects the EMF model representing the AST with the textual form of the program in the text editor. These two forms of the program are kept in synchronization automatically by the framework. This gives the language developer for free IDE functionalities such as the outline view (we refer to a later paragraph in this Section and to Figure 1). But most of all, it allows programmat-
ically modifying the program by acting on the model representing it, without the need of accessing its textual form in the editor (see the quickfix implementation in Listing 5 in Section 3), which will be automatically updated to reflect the changes in the model.

There is a direct correspondence between the names used in the rules of the grammar and the generated EMF model JAVA classes. For instance, if we have the following grammar snippet:

```
Selection returns Expression: receiver=Expression '.' message=Message;
```

```
Message: MethodCall | FieldSelection;
```

```
MethodCall: name=[Method] '(' (args+=Argument (',' args+=Argument)*)? ')';
FieldSelection: name=[Field];
```

the XTEXT framework generates the following EMF model JAVA interface (and the corresponding implementation class):

```java
public interface MethodCall extends Message {
    Method getName();
    void setName(Method value);
    EList<Argument> getArgs();
}
```

Besides, XTEXT generates many other classes for the editor for the language that we are implementing. The editor contains syntax highlighting\(^2\), background parsing with error markers, outline view, code completion. Most of the code generated by XTEXT can already be used off the shelf, but other parts can or have to be adapted by customizing some classes used in the framework. The usage of the customized classes is dealt with by relying on Google-Guice [1], so that the programmer does not have to maintain customized abstract factories [17]. This way, it is very easy to insert custom implementations into the framework (“injected” in Google-Guice terminology), with the guarantee that the custom classes will be used consistently throughout the code of the framework.

**Validation.** The validation mechanisms for the language must be provided by the language developer. In our case, this is the FJ type system. Implementing the validation mechanism in a compiler usually requires to write specific visitors for the abstract syntax tree. EMF already simplifies this task by providing a switch-like functionality to efficiently execute methods with dynamic dispatch according to the actual type of an AST node. Thus, there is no need to add code to implement a visitor structure [17]. In order to use this functionality it is enough to inherit from the generated class implementing the switch functionality (in our case the generated EMF FjSwitch base class), and provide the methods for all the model classes we want to deal with (the generated

---

1. If the reader compares this simplified grammar snippet with the complete grammar in Listing 2 she will note a different rule for method invocation; this is due to the fact that in order to deal with left recursion, which LL-parsing tools cannot handle directly, we need to “left-factor” the grammar [6].
2. Note that typically, when implementing manually a plugin for a language in Eclipse, writing the parser does not automatically provides a syntax highlighter, which must be implemented manually in Java.
switch functionality provides default cases for the classes for which we do not provide a case and also handles class inheritance in the model class hierarchy). For instance, we used this technique to implement the type checker (the generic parameter of the switch class represents the return type of the case methods) as (partially) shown in Listing 3. We implemented case methods for some classes of the model representing the AST of an FJ program, and we then have the method typeCheck that takes as argument an EObject (the base class for all the classes of a generated EMF model) which simply calls method doSwitch. In particular, this class relies on other objects (whose classes are not shown here) that we implemented for inferring a type of an AST node (typeSystem) and for checking the subtyping (subclass) relation (subtyping). In our implementation we use strings to return possible type errors, thus, if the method returns an empty string it means that the checked expression is well-typed.

XTEXT leverages this mechanism by only requiring methods with a @Check annotation in a customized validator, that will be called automatically for validating the model according to the type of the AST node being checked. The validation takes place in the background, together with parsing, while the user is writing a FJ program, so that an immediate feedback is available. For instance, the following method in the FJJavaValidator checks that the cast expression is correct:

```java
public class FJTypeChecker extends FjSwitch<String> {
    ...
    public String caseField(Field field) {...}
    public String caseMethod(Method method) {...}
    public String caseCast(Cast cast) {
        TypeResult objectType = typeSystem.getType(cast.getObject());
        if (!subtyping.isSubtype(objectType.getType(), cast.getType())
            && !subtyping.isSubtype(cast.getType(), objectType.getType()))
            return "expression type "
                + objectType.getType() + " and "
                + cast.getType() + " are unrelated";
        return null;
    }
    ...
    public String typeCheck(EObject object) {
        String errors = doSwitch(object);
        return errors;
    }
}
```

Listing 3: The typechecker implementation (snippet).
Note that the only important things in this method definition are the @Check annotation and the parameter: the internal validator of XTEXT will invoke this method when it needs to validate an AST node representing a cast expression, i.e., a Cast instance, which corresponds to the Cast rule in the grammar of Listing 2. The implementation of the validation in our case simply delegates to the type checker class shown above. Calling the method error will make XTEXT generate the appropriate error marker (and since we specify the element in the AST which did not pass validation, XTEXT is able to put the marker in the right place without any further information by the programmer).

Cross references. Binding the symbols (e.g., the binding of a field reference to its declaration) is important in compiler development. When defining the grammar of the language in XTEXT, we can already specify that a specific token is actually a reference to a specific declared element. Going back to the snippet at the beginning of the section, we note that the name of the method in a method invocation is enclosed in square brackets ([Method]). This means that the token representing the method’s name in a method invocation is not simply an identifier: it is a reference to the name of a declared Method (see also the complete grammar in Listing 2). For field selection we can do a similar thing (referring to the name of a declared Field). This allows providing more detailed information in the grammar itself, and in the generated EMF model, the name of the method in a method invocation expression will not be simply a string, but it will be a cross reference to an instance of Method. (See the return type of getMethod in the generated MethodCall class.)

In particular, EMF uses “proxies” to represent references and it can delay the resolution (binding) of references when they are accessed. XTEXT already provides an implementation for binding references, which basically binds a reference to a symbol n to the first element definition with name n occurring in the model3. However, this usually has to be adapted in order to take the visibility (scope) of names in a program into account. For instance, a field is visible only in the methods of a class, such that different hierarchies can safely have fields with the same name. XTEXT supports the customization of binding in an elegant way with the abstract concept of “scope”. The actual binding is still performed by XTEXT, but it can be driven by providing the scope of a reference, i.e., all declarations that are available in the current context of a reference. The programmer can provide a customized AbstractDeclarativeScopeProvider. XTEXT, when it needs to bind/resolve a symbol, will search for methods to invoke, using reflection, according to a convention on method name signatures. Suppose, we have a rule ContextRuleName with an attribute ReferenceAttributeName assigned to a cross reference with TypeToReturn type, that is used by the rule ContextType. You can create one or both of the following two methods

\[
\begin{align*}
\text{IScope scope}_<\text{ContextRuleName}>_<\text{ReferenceAttributeName}> \\
\quad (<\text{ContextType}> ctx, \text{EReference} ref)
\end{align*}
\]

\[
\begin{align*}
\text{IScope scope}_<\text{TypeToReturn}>_<(\text{ContextType}> ctx, \text{EReference} ref)
\end{align*}
\]

The XTEXT binding mechanism looks for the first method (by reflection), if this does not exist, then it looks for the second. If no such method exists, the default linking semantics (see above) is used. These methods are supposed to return the set of all “visi-

3 XTEXT, by convention, relies on the attribute name of model elements.
public IScope scope_MethodCall_name(Selection sel, EReference ref) {
    TypeResult selectionExpressionType = typeSystem.getType(sel.getReceiver());
    Class receiverType = selectionExpressionType.getClassref();
    if (receiverType != null)
        return Scopes.scopeFor(auxiliaryFunctions.getMethods(receiverType));
    else // return an empty scope
        return Scopes.scopeFor(new LinkedList<EObject>());
}

Listing 4: Scope provider for a method invocation expression.

ble” elements in that part of the program (represented by the passed context); using the returned scope, XTEXT will then take care of resolving a cross reference (or issue an error in case the reference cannot be solved). If XTEXT succeeds in resolving a cross reference, it also takes care of implementing the functionalities Eclipse users are used to, e.g., by Ctrl+clicking on a reference (or using F3) we can jump to the declaration for that symbol.

For instance, if we consider the grammar rule for method invocation illustrated at the beginning of this section, we can drive the resolution of the method name in a method invocation statement in any expression where such statement can occur by defining the method shown in Listing 4 (The code should be understandable without the knowledge of XTEXT).

**IDE functionalities.** The scope provider will be used by XTEXT not only to solve references, but also to implement code completion. Thus, a programmer achieves many goals by implementing only the abstract concept of scope. Note that the code above can also return an empty scope, e.g., if the receiver expression in a method call cannot be typed. In that case, the XTEXT framework generates an error due to an unresolvable method name during validation, and an empty code completion list in case the programmer requests content assistance when writing the method name of a method invocation expression. This mechanism is handled by the framework itself, so that the programmer is completely relieved from these issues, once the correct scope provider is implemented.

Finally, the code generation phase is dealt with in XTEXT by relying on XPAND [4], a code generation framework based on “templates”, specialized for code generation based on EMF models. For FJ we did not need to implement a code generator (and indeed, this step is completely optional in XTEXT, and the code generation might be carried on also with other tools), since an interpreter would be more appropriate. However, since we wanted to focus on the typechecking part for FJ and on the IDE functionalities, we are still working on the interpreter.

Figure 1 shows a screenshot of the FJ editor; note the code completion functionalities, the outline and the error markers (note also the folding of multiple line elements, like classes and methods, which is handled automatically). Also the outline view is handled transparently by XTEXT; the default implementation of the outline view is to simply show the EMF model representing the abstract syntax tree of the program. This is not always the right thing, especially for a general purpose language. Again, the out-
line view can be customized by not showing specific elements (in our case we do not show the body of methods) and by providing a customized label provider (in our case we show the name of fields and their types and the names of methods with their signatures). Besides that, the synchronization between the outline view and the contents of the editor is handled completely by XTEXT.

3 Evaluation

Our experience with XTEXT was in general quite positive. The main issue (which is typical of such high level frameworks) is that some time is required to get acquainted with the concepts of the framework. However, once this knowledge is achieved, developing a language compiler and an IDE using XTEXT is extremely fast. Surely it is much faster than implementing an IDE from scratch by only relying on the eclipse framework: as illustrated in Section 2 all the connections among the typical eclipse artifacts for editors, builders, etc., are established and handled by XTEXT directly.

As we described in Section 2 by using our own validator, XTEXT can create the error markers in the editor (and also in the “Problem View”) related to the EMF element which caused the error. We can provide our own “quickfixes” by simply deriving from an XTEXT default class and providing a method which refers to a specific error detected by our validator. For instance, the code shown in Listing 5 provides a quickfix in case of duplicate classes in a program, by proposing to remove one class.\footnote{The Class class in the code snippet is not \texttt{java.lang.Class}, but the Class of our EMF model, representing a FJ class.} Note how easy it is...
public class FJQuickfixProvider extends DefaultQuickfixProvider {

    @Fix(FJJavaValidator.DUPLICATE_CLASS_NAMES)
    public void removeDuplicateClass(final Issue issue, IssueResolutionAcceptor acceptor) {
        acceptor.accept(issue, "Remove class", ...)
        new ISemanticModification() {
            public void apply(EObject element, IModificationContext context) {
                Class duplicateClass = (Class)element;
                Program prog = (Program)duplicateClass.eContainer();
                prog.getClasses().remove(duplicateClass);
            }
        };
    }
}

Listing 5: An example of quickfix implementation.

![Image](image1.png)

Fig. 2. The quickfix in action.

to implement that fix: since we deal with EMF model objects, which are connected by XTEXT to the program text in the editor, we can simply manipulate the EMF model (in this case we remove the class from the program) and the program text will reflect this change; in particular, we do not need to manipulate the eclipse text editor document at all. Figure 2 shows our quickfix in action.

XTEXT seems to be the right tool to experiment with language design and to develop implementations of languages. Furthermore, experimenting with new constructs in the language being developed can be handled straightforwardly. It requires to modify the grammar, regenerate XTEXT artifacts and to deal with the cases for the new constructs. Finally, XTEXT allows the programmer to customize every aspect of the developed language implementation using specialized code (which is flexibly “injected” in XTEXT using Google Guice), even though XTEXT hides many internal details of IDE development with Eclipse. Even EMF mechanisms are still open to adaptation. For instance, we developed a customized EMF resource factory for synthesizing the implicit class Object (in FJ the class Object is implicitly defined in every program, and it does not contain any field or method).

XTEXT provides some useful functionalities to write Junit tests for many language development components; in particular, it generates a stand-alone class (in our case FJ-
public class CastTest extends AbstractXtextTests {
    protected void setUp() throws Exception {
        super.setUp();
        with(new FJStandaloneSetup());
    }

    public void testCastFail() throws Exception {
        Program program = (Program) getModel("class B { } class A { } (B) new A()");
        Expression main = program.getMain();
        String errors = new FJTypeChecker().typeCheck(main);
        assertEquals("expression type A and B are unrelated", errors);
    }
}

Listing 6: The Junit test for typechecking of a cast expression.

StandaloneSetup) for the developed language with all the functionalities to correctly initialize all the EMF mechanisms so that the language can be tested as a stand-alone application (i.e., outside from Eclipse). This way, we can easily test non-UI parts of our language with no need of manually running an Eclipse application, writing a code snippet in our language, and checking that the error mark shows up\(^5\). This speeds up the development of the language; for instance, in Listing 6 we show a snippet of the test class for our type checker (compare the expected error with the type checker code for cast expression shown in Listing 3).

We would like to conclude this Section by stressing that XTEXT lets the language developer concentrate on the aspects that are typical of her own language, while relying on the framework for all the other recurrent jobs. Just to mention a fact: for developing our FJ eclipse based IDE, we did not have to write a single extension point (which, instead, is usually required when developing an Eclipse plugin).

### 4 Related Work

There are other tools for implementing both domain specific and general purpose languages and their text editors and IDE functionalities (we also refer to [22] for a comparison). Tools like IMP (The IDE Meta-Tooling Platform) [2] and DLTK (Dynamic Languages Toolkit) [15] only deal with IDE functionalities and leave the parsing mechanism completely to the programmer, while XTEXT starts the development cycle right from the grammar itself. TCS (Textual Concrete Syntax) [3] is similar to XTEXT since it enables the specification of textual concrete syntaxes for DSLs by attaching syntactic information to metamodels; however, with XTEXT one describes the abstract and concrete syntax at once, and it is completely open to customization of every part of the generated IDE (besides, TCS seems to be no longer under active development). Another framework, closer to XTEXT is EMFText [18]. EMFText basically provides the

\(^5\) For the moment, XTEXT does not provide corresponding utility test classes for UI parts, e.g., code completion, quickfixes, etc. However, one can start from the UI tests of the source code of XTEXT itself to write Junit tests also for these UI components of the developed language.
same functionalities. But, instead of deriving a meta-model from the grammar, it does the opposite, i.e., the language to be implemented must be defined in an abstract way using an EMF meta model. (A meta model is a model describing a model, e.g., an UML class diagram describing the classes of a model). Note that XText can also connect the grammar rules to an existing EMF meta model, instead of generating an EMF meta model starting from the grammar. Furthermore, XText has also a wizard to generate an XText grammar starting from a EMF meta model. XText seems to be better documented than EMFText (indeed, both projects are still young and always under intense development), and more flexible, especially since it relies on Google Guice. On the other hand, EMFText offers a “language zoo” with many examples that can be used to start the development of another language. In this respect, the examples of languages implemented using XText, that we found on the web, are simpler DSLs, and not programming languages like FJ. Thus, this paper can also be seen as a report of effective usage of XText for implementing more complex programming languages.

EriLex [25] is a software tool for generating support code for embedded domain specific languages and it supports specifying syntax, type rules, and dynamic semantics of such languages. EriLex does not generate any artifact for IDE functionalities, and it concentrates on other aspects of language development such as type systems and operational semantics. MPS (Meta Programming System) [16] is another tool for developing a DSL, and it also provides IDE functionalities, but it does not target Eclipse.

Neverlang [14] is based on the fact that programming language features can be easily plugged and unplugged. A complete compiler/interpreter can be built in Neverlang as the result of a compositional process involving several building blocks. With respect to composition functionalities, XText allows the programmer to mix grammars (so called “grammar mixins”) and also to reuse recurrent syntax artifacts (like the standard terminal definitions, see the “with” statement at the beginning of Listing 2). However, these compositional functionalities are not yet as powerful as the ones provided by Neverlang.

5 Conclusions and Future Work

Our implementation of of FJ, available from http://fj-eclipse.sourceforge.net, was the starting point for the implementation of more involved programming languages, such as SUGARED WELTERWEIGHT RECORD-TRAIT JAVA, http://swrtj.sourceforge.net, presented in [13], which is based on the calculus presented in [12], formalized by extending FJ.

We are planning to keep on using XText also for implementing other programming languages that we studied formally (and also language extensions). As for the FJ implementation, we will use it as a case study for experimenting with several XText features (e.g., we plan to enhance the language itself with “import” functionalities and to implement an interpreter for FJ programs, based on the file inclusion support provided by XText).

It would be interesting to extend XText itself in order to provide a richer framework for developing type systems (like [25, 16]), e.g., by providing a DSL (developed in XText) for type rules that act on the model of the language. We would like also to
investigate more powerful language/grammar composition functionalities of XTEXT by taking into consideration Neverlang [14] as the main comparison.

Acknowledgments. We are grateful to the developers of XTEXT, in particular, Sven Efftinge and Sebastian Zarnekow, for their prompt help and support during the development of FJ.

References

17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
19. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GI. ACM Transactions on Programming Languages and Systems, 23(3):396–450, 2001.