Development of a Domain-Specific Language for Pipe-and-Filter Configuration Builders

Martin Zloch
Kiel University
Department of Computer Science
24098 Kiel, Germany

Abstract. This paper discusses the development of a domain specific language with Xtext for modelling configurations of the TeeTime Pipe-and-Filter framework. TeeTime is a framework that enables a fast integration of programs written in Java into a Pipe-and-Filter architecture. Therefore, different parts of a program must be integrated in stages which can be executed consecutively or simultaneously. The connection of those stages are called pipes. An important step in implementing a Pipe-and-Filter program is to define those pipes. This definitions were previously realised by Java code and had to be written by the programmer, requiring some understanding of the internal TeeTime structure. To abstract the composition of stages from the TeeTime implementation, we propose a domain specific language for this purpose. In this paper, we present this DSL and explain important design decisions. Also a generator for TeeTime configurations from configuration descriptions written in the DSL is proposed.

1 Introduction

The Pipe-and-Filter style is a commonly known software architectural style. It can be used in data analysis and many other use cases. To abstract the architecture from the business logic, the TeeTime [5] framework can be used. It offers structures to implement Pipe-and-Filter Java applications without having to write much architecture specific code and enables a strict separation of business logic from the architectural style. Domain specific languages are a popular way to allow users to describe things in a way that is more applicable to the specific task. Xtext [2] is a framework for integration of DSLs in the eclipse IDE.

One of the things that has to be implemented for the usage of TeeTime is a configuration, that describes the actual architecture of the software. We propose the usage of a DSL to automatically generate those configurations. In this way, we achieve the following advantages:

Abstraction of the framework The configuration is separated from the used framework. The same configuration written in the DSL could be used to generate code for other frameworks. This allows a quick switch of Pipe-and-Filter frameworks, given there is a generator for the same DSL.
**Fitting Notation** A DSL allows to choose the most fitting notation. We can use shorter code, that only contains the necessary information, is shorter, better formatted and more readable than Java code would be.

**Faster Learning** A programmer doesn’t need to know the internal structure of TeeTime and framework specific namings, classes and methods.

In this paper, we first explain the foundations needed to understand the DSL and the used technology in section 2. Afterwards, the necessary information for generating TeeTime configurations is evaluated in section 3. Then, in section 4, the previously gained knowledge is used to develop the DSL. The last development step is the code generation, which is explained in section 5. The following section 6 is about the evaluation of the DSL and the code generation. Future work is presented in section 7. In section 8 we summarise the conclusions of this paper.

## 2 Foundations

In this section, we present the foundations of this paper to develop and understanding of the concepts and tools used to realise a DSL for Pipe-and-Filter configuration builders.

### 2.1 Domain Specific Languages

Domain specific languages[3] are limited programming languages that focus on a particular domain. This means, that a DSL is a tool for a very closely defined task. Its goal is to support the features needed by the domain in the best way possible to allow focusing on the main subject.

### 2.2 Xtext

Xtext [1] is a framework to build domain specific languages and supporting tools. It defines a notation similar to EBNF notations to write the language grammar. From a grammar, it generates AST classes, a parser and an Eclipse editor with various features for writing files in the DSL.

### 2.3 The Pipe-and-Filter Architectural Style

The Pipe-and-Filter [4] architectural style separates a software in independent computation units called filters and their connections, called pipes. Input data is read by a start filter and then transferred to the next filter by a pipe until a final filter is reached. A special feature of this architectural style is, that pipes are first class elements. It implies a strong separation of parts of computation and therefore supports maintainability. Filters and Pipes may be executed in separate threads to allow scaling of computation intensive data transformations. Filters may have multiple input and output ports for data and pipes may connect any output with any input port, so that it is possible to model almost any kind of transmission graph.
2.4 The TeeTime Pipe-and-Filter Framework

TeeTime [5] is a Pipe-and-Filter framework for Java-based applications. It offers implementations of base classes for filters, which are called stages in TeeTime. These can be used to implement own stages with business logic. Another class called Configuration can be used to form the transmission graph with the connections of stages. These configurations also define which stages run in a separate thread and initialize the used stages. A DSL for the generation of configurations is the subject of this paper.

3 Requirements on a Pipe-and-Filter DSL

In this section, we explain what the requirements for a DSL of a Pipe-and-Filter configuration are. Therefore, we examine the CipherConfiguration example from the TeeTime project which is shown in Figure 1. The example is shortened for clarity. This configuration creates several stages that read a file, encrypt it, and then write the file to a given path. It was created to test the used stages and give an example for the usage of TeeTime.

```java
package teetime.examples.cipher;

import java.io.File;
import teetime.stage.io.File2ByteArray;

public class CipherConfiguration extends Configuration {
    public CipherConfiguration(String inputFile, String outputFile, String password) {
        final File input = new File(inputFile);
        final File output = new File(outputFile);

        final InitialElementProducer<File> init = new InitialElementProducer<File>(input);
        final CipherStage enc = new CipherStage(password, CipherMode.ENCRYPT);
        final ArrayFileWriter writer = new ArrayFileWriter(output);

        connectPorts(init.getOutputStream(), enc.getInputPort());
        connectPorts(enc.getOutputStream(), writer.getInputStream());
    }
}
```

Fig. 1. CipherConfiguration as an Example for a Configuration in Java
The developed DSL must be able to provide structures that support containment of all information needed to generate code which is equivalent to the CipherConfiguration from a configuration description, so we go through the code and extract the necessary information.

**Package** Line 1 in the file is the Java package. Since it is optional in Java, it should be optional in the DSL, too.

**Imports** Imports are necessary to improve readability of Java code as the fully qualified names of classes must be written only once and further on, the simple name of the classes suffice. The end user of the DSL shouldn’t need to inspect the code, if everything works right. So we could not do any imports and just write the fully qualified name all the time. But in real environments, often unexpected requirements pop up or a developer wants to understand the generated code. For this reason, the DSL should support imports. Since imports can be generated from the names of declarations of stages and attributes, we don’t need explicit imports as part of the DSL to generate them in the Java code.

**Class Declaration** One thing in the class declaration in line 6 is the name of the class. This is mandatory and in Java typically used as the file name. The `public` and `class` keywords are equivalent for all configurations and don’t require extra information.

TeeTime configurations inherit the TeeTime class `Configuration` from the package `teetime.framework.Configuration`. Since we want to support the generation of TeeTime Configurations, generated configurations must extend this class. Since this is necessary for all configurations and thus, we don’t need to explicitly declare it in the DSL.

**Constructor Declaration** The constructor name declared in line 7 is the same as the class and file name. Since the name is required for those already, it’s not needed a second time. The attributes however can not be derived from somewhere else and must be part of a configuration description.

**Helper Variables** The helper variables in line 10 and 11 of the given example are used to convert some of the constructor attributes. Alternatively, this can be done by performing the conversion in the class creating the configuration and passing the correct attributes directly to the constructor. This behaviour is more consistent, too. For more complex use cases it would be hard to detect, in which way such casts can be performed. Because of these reasons, we don’t want to support those helper variables and thus don’t need to put any information for it in the DSL.

**Stage Declaration and Initialization** For the declaration of stages, the fully qualified name and an id is required as shown in line 13 to 18. The initialization of stages may need some type attributes, the DSL must support optional type attribute declarations for stages. As some stages may be generic classes, the DSL should allow type parameter declarations.

**Connection declarations** Line 20 and 21 show, how TeeTime configurations use the method `connectPorts` from the class `Configuration` to defines connections between ports of different stages. In the most common case, stages
have one input port and one output port, called \textit{inputPort} and \textit{outputPort}. These are accessible by the getter methods \texttt{getInputPort} and \texttt{getOutputPort}. We assume this naming as the default to make the explicit declaration of port names optional. The connection between the stages stays mandatory, since it is the essential part of the configuration and can not be derived.

**Additional Features** To make Pipe-and-Filter software scalable, stages can be executed in different threads. TeyTime supports the declaration of stages, which should run in separate thread with the method \texttt{declareActive} of the class \texttt{AbstractStage} which all stages inherit. To support this, the DSL must contain a way to mark active stages.

In summary, the information in a configuration description must contain the following:

- package (optional)
- class name
- constructor attributes (optional)
- fully qualified names of stages
- ids of stages
- parameter for stages (optional)
- attributes for stage initialization (optional)
- connections of stages with optional port names
- activeness of stages (optional)

4 The DSL Syntax and Grammar

In this section, we introduce a DSL written in Xtext (subsection 2.2) that supports all features from section 3. We show the grammar forming the DSL and explain the resulting syntax.

4.1 Top-Level Configuration Elements

In our DSL, there is only one configuration description per file. This leads to a mapping from one configuration description file to one generated configuration file which improves the clarity in projects. The suggested procedure is to name the configuration description like the generated configuration, then there can’t be two descriptions generating a configuration with the same name. Since the DSL generates Java code and the programmer needs to know Java for using it, we use a Java like syntax were possible and reasonable. This improves readability and decreases the time for learning grammar syntax.

The first part of a configuration declaration is everything essential for the class: The package, the class name, and arguments for the constructor. The grammar shown in Figure 2 defines an optional package declaration, a mandatory type name that is the name of the configuration, and an optional list of arguments.
The package and type are qualified names which are defined by a string, called ID in Xtext, that may contain dots. Qualified names don’t have to be actual fully qualified names. This allows easier integration of shorter notations when the simple name is sufficient. These are explained in subsection 4.4. If there are any constructor arguments, they must be put in parentheses and contain a type and an id.

The elements of a configuration, like stage and connection definitions, are put one per line in curly braces. This separates the configuration properties from the stage and connection definitions and marks them as associated in a Java like style.

There are three types of configuration elements:
1. stage definitions
2. connection definitions
3. import declarations

These are explained in the next subsections. The expression in the grammar is shown in Figure 3.
4.2 Stage Declaration

As explained in section 3, a configuration description must contain the following information:

1. fully qualified names of stages
2. ids of stages
3. parameter for stages (optional)
4. attributes for stage initialization (optional)
5. activeness of stages (optional)

We define the grammar for stage definitions like shown in Figure 4:

```
Stage : active = 'active' ? type = QualifiedName (parameter = Parameter)?
     id = ID ( ' ( ' argumentList = QualifiedNameList ' ) ' ) ?
     ;

Parameter :
    '<' qualifiedNameList = QualifiedNameList '>'
    ;

QualifiedNameList :
    name1 = QualifiedName ( ',' othernames += QualifiedName )*
    ;
```

Fig. 4. Grammar for Stage Definitions

The first part is an optional active flag to mark active stages. The class name of the stage is inserted mandatorily as a QualifiedName. The id is mandatory too but may not contain dots, as this is not valid in Java.

The type parameters are optional and may be more then one, separated by comma. If there is any parameter, the parameters must be put in brackets. This is equivalent to the notation in Java.

After that, the optional argument list is defined in parentheses. Arguments are qualified names to allow string containing dots, like constants, and variables from the configuration constructor without dots.

4.3 Connection Definitions

To allow both shorter configuration descriptions and random connections of ports, our grammar allows two notations of connection definitions.

The first are pairwise definitions with optional output and input port names as shown in Figure 5.
This notation allows to define precisely which output port is connected to which input port. In this way, a stage with multiple input or output ports can be connected with multiple other ports.

The second notation is defined by Figure 6.

This notation allows definitions like stage1->stage2->stage3->stage4, which is easier to read and shorter to write. Since this notation is limited by the 2D space of the written form, it supports only stages with one input and one output port. The exceptions are, that the first may have multiple input ports and the last may have multiple output ports. Therefore, and because a notation with a defined input and output port would lose its benefits regarding readability, we don’t allow explicit port names in this notation.

4.4 Import Declarations

With only the previous configuration elements, stage and connection definitions, it is possible to define a complete configuration, if the fully qualified name is used in every type definition.

To allow authors of configuration descriptions to write fully qualified names only once, we introduced import declarations as a third sort of configuration elements. The grammar is shown in Figure 7.
An import declaration starts with the keyword import, as in Java, and may end with a semicolon. It contains the imported class or package as a fully qualified name.

5 Code Generation

In this section we present the implementation decisions regarding the code generation. The generator takes a configuration description and creates a single Java file from it that is the generated TeeTime configuration. Therefor we wrote methods for each part of a generated Java file. These parts are:

1. Package
2. Imports
3. Stage declarations as attributes
4. Getters for stages
5. Parameters of the constructor
6. Stage initialization
7. Connection declarations
8. Active declarations

These methods are called each once and take the required information from the configuration description to generate their part of the file. The methods for stage declarations, initializations, getters, connection, and active declarations each have a for-loop over all elements of the configuration description and calling another method that generates one part for that element. For example, the getter generation method generates a getter for each stage defined by the configuration description. The generation of imports has some differences that are discussed in subsection 5.2.

Most of these methods is simple enough to be understood without further explanation, so we discuss only the non-trivial parts in the following subsections.

5.1 Path of the Generated File

The generator takes the package name and type of a configuration from the configuration description to build the path of the generated file like shown in Figure 8.
```java
if (configuration.package != null) {
    iFileSystemAccess.generateFile(
        configuration.package.replaceAll("\\\." , "/") + "/" +
        configuration.type + ".java",
        configuration.compileConfig)
} else {
    iFileSystemAccess.generateFile(configuration.type + ".java",
        configuration.compileConfig)
}
```

**Fig. 8. Generation of the Filename**

The `generateFile` method interprets slashes as path separators, so the dots in the package name are replaced with them. The `replaceAll` method takes a regular expression in the first argument, so we have to escape it. Configuration.compileConfig is the call of the main generation method that generates the complete content of the Java file.

### 5.2 Imports

Imports are not as easy to generate as e.g. stage declarations because of the following two reasons.

The first one is, that an import may be generated multiple times, if the same type is used twice with it's fully qualified name. We want to avoid those duplicates to improve the generated codes clarity.

The second reason is, that there are multiple entries in a configuration that can lead to an import, since we allow the programmer to use fully qualified names for used types or manual import handling with the `import configuration element`. This is solved by separating the generation of imports in two parts: First, import candidates of all sources are collected. The import elements of the description are used directly. The generator collects fully qualified names from all stage declarations from

- The stage type
- Stage parameters
- Stage constructor arguments

and then collects types of all constructor arguments of the configuration. Then, in the second part, for each entry of the import collection, an import statement is generated. To avoid duplicates we use `java.util.Set` as collection.

### 5.3 Stage Declaration and Stage Getters

In Figure 1, stages are declared in the constructor. We change this behaviour to allow access to the stages at runtime. The stage declarations are attributes to the configuration and we generate getters for them. In this way, further configuration of the stages is possible without changing the generated configuration file.
6 Evaluation

In the previous sections, we have proposed a DSL for TeeTime Pipe-and-Filter Configurations and discussed the generation of code from a configuration description written in the DSL. In this section, we evaluate the functionality of the DSL and the generator by writing a configuration description that contains features offered by TeeTime as described in section 3. Afterwards we inspect the description to evaluate the DSL.

6.1 Example Configuration

The most simple example of a configuration contains no stages, has no constructor arguments and is in the default package. The description for this example is shown in Figure 9.

```java
 EvaluationConfiguration {}  
```

Fig. 9. Description of an empty Configuration

The description from Figure 9 generates the file EvaluationConfiguration.java shown in Figure 10.

```java
import teetime.framework.Configuration;

public class EvaluationConfiguration extends Configuration {
    public EvaluationConfiguration() {
    }
}
```

Fig. 10. Generated empty Configuration

The generated class meets the expectations and shows, that package name and constructor arguments are at most optional. Figure 11 shows the next description. It generates a shortened and thus not functional version of the CipherConfiguration. A full example was successfully tested too, but is not shown in this paper.
It contains examples for the following features:

- Package declaration
- Constructor arguments
- Import declarations
- Active stage
- Non active stage
- Stage with fully qualified name
- Stage with simple name
- Stage with type parameter
- Stage without type parameter
- Stage with variable as constructor argument
- Stage with constant as constructor argument
- Stage with multiple constructor arguments
- Stage without constructor arguments
- Connection declaration with explicit port names
- Connection declaration in short notation

The generated class `de/dsl/eval/EvaluationConfiguration.java` shown in Figure 12 implements the targeted features correctly. Thus we have shown, that the DSL provides ways to contain all needed information to model the features mentioned above and the generator is able to generate runnable code from configuration descriptions, which is similar to the `CipherConfiguration` in Figure 1.
import java.io.File; //
import teetime.stage.io.File2ByteArray;

public class EvalConfiguration extends Configuration{
    private final InitialElementProducer<File> init;
    private final File2ByteArray f2b;
    private final CipherStage enc;
    private final ByteArrayFileWriter writer;

    public InitialElementProducer<File> getInit(){
        return init;
    }
    public File2ByteArray getF2b(){
        return f2b;
    }
    public CipherStage getEnc(){
        return enc;
    }
    public ByteArrayFileWriter getWriter(){
        return writer;
    }

    public EvalConfiguration(File input, File output, String password) {
        init = new teetime.stage.InitialElementProducer<File>(input);
        f2b = new File2ByteArray();
        enc = new teetime.stage.CipherStage(password, CipherMode.ENCRYPT);
        writer = new teetime.stage.io.ByteArrayFileWriter(output);

        connectPorts(init.getOutputPort(), f2b.getInputPort());
        connectPorts(f2b.getOutputPort(), enc.getInputPort());
        connectPorts(enc.getOutputPort(), writer.getInputPort());

        init.declareActive();
    }
}

Fig. 12. Generated Code from Figure 11

6.2 Syntax Evaluation

In Figure 11 we see a configuration description that generates a slightly varied variant of the cipher configuration shown in Figure 1. In comparison, there are several differences. The length of the configuration is reduced mainly by omitting implicit Java keywords like \texttt{final} and implicit constructor calls and assignments. Shorter code means faster development, easier debugging and more clarity. Another benefit from using the DSL is the intuitive and brief notation of connections. The short notation allows writing a connection of any number of stages in one line, while the Java version needs a line for each pair of connected stages.
Also, the notation can be seen as an optical representation of the architecture of the software. The DSL uses no TecTime dependant features. Thus, the DSL can easily be enhanced to support other Pipe-and-Filter frameworks just by writing a new generator. Existing configurations written in the DSL would be compilable under the new framework without changing them. A weakness of the DSL is, that the short notation does not support stages with multiple input or output ports. Thus the visual representation is only possible for linear configurations. Also, it is currently not possible to manually define connections.

7 Future Work

We explained in subsection 6.2 that one weakness of the DSL is, that the short notion can’t handle multiple input and output ports. A possible way to enable this could look like proposed in Figure 13.

```
stage1—>stage2
outputPort1—>stage3
outputPort2—>stage4
```

Fig. 13. Proposal for a Multi Output Port Stage Configuration Notation

This notation raises several other problems, like how the combining of stages can be noted. This leads to the question, if a graph representation is better. Another approach would be automatic connection of the ports by their numbers, if the used stages contain their ports in a list. Future work will be to evaluate, which approach is the best. To allow manual definitions of connection types, we propose the notation shown in Figure 14 as future work.

```
stage1—[ConnectionType(constructorArgument)]—>stage2
```

Fig. 14. Proposal for Explicit Connection Types

Another opportunity to enhance the DSL further is stricter limitation of notations. Especially a distinction of fully qualified names and simple names would help Authors of configuration descriptions to chose the correct form. Some other possible mistakes, like missing imports and unconnected ports, could be prevented by extended validators.
8 Conclusions

In this paper, we proposed a DSL that contains all information required for standard TeeTime configurations, like the CipherConfiguration example. We have developed a generator that uses a configuration description written in this DSL to create a fully functional TeeTime configuration in Java and tested it by generating a configuration with the same functionality as the cipher example.

In conclusion, we fulfilled our goal to develop a first version of a DSL for the TeeTime Pipe-and-Filter framework that is independent of the framework itself.

References