RECFM User Guide

Code generation to handle positional files

VERSION 0.7.0

AUGUST 14, 2023

Introduction

Sometimes it may happen that you have to deal with positional files (or memory areas), see fig. 1, in these cases you need to waste a lot of time to make a class dedicated to each string-data with the setters and getters to read and write values[1].

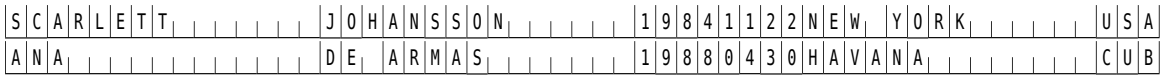

Figure 1: Positional data-file example

This group of programs aims to minimize the time to create these classes. In practice, the structure of the data-string is defined with a configuration file, this is fed to a plugin that generates the corresponding data-class, which can be used without anything else user intervention.

Programs are structured using service provider interface, see fig. 2, we have a plugin, or user program (*Service*), which directly sees the classes defined in the *Service Provider Interface* and retrieves them the implementation using the *ServiceLoader*, this way it doesn't have a specific dependency with one of the the implementations used. The *Service Provider* must implement the classes defined in the *Service Provider Interface*.

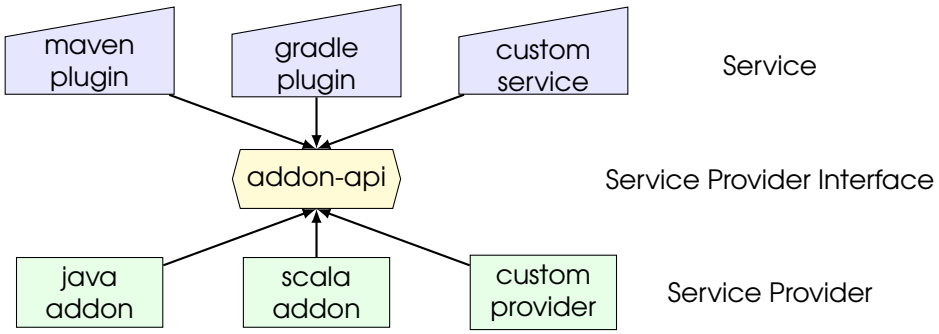

Figure 2: Structure service, service-provider-interface, service-provider

If the `maven-plugin` finds the library with the `java-addon` implementation running it will generate the sources in java, but if it finds the `scala-addon` implementation it will generate the scala sources.

The documentation is divided into three parts. In the first, I, a detailed description of the classes is given defined in the *service provider interface*, this part is useful for those wishing to develop a *custom service* or a *custom provider*. If you are only interested in how to generate code starting from config files it can be skipped.

In the second part, II, a description of two plugins used to generate the code is given. In particular how to define the layouts with the configuration files and how to activate the plugin.

In the third part, III, a description of the *service provider* that generates the java source showing also some additional features of generated classes beyond simple setters and getters.

[1] there is `com.ancientprogramming.fixedformat4j:fixedformat4j` which provides some basic functionality, but in most situations it is not flexible enough.

Contents

Part I

Service Provider Interface

Chapter 1

Service Provider Interface

The `recfm-addon-api` artifact provides a series of interfaces, some enums and java-beans to allow the client module to define the paths. The code is compiled so that it is compatible with java 5, but it provides the `module-info` to be usable properly even with java 9 and above.

1.1 CodeProvider

The starting point is the *CodeProvider* interface, retrieved from the *ServiceLoader*, see lst. 1, this interface provides the instance of the *CodeFactory* interface.

```
ServiceLoader<CodeProvider> loader = ServiceLoader.load(CodeProvider.class);
CodeProvider codeProvider = loader.iterator().next();
```

```
public interface CodeProvider {
     CodeFactory getInstance();
}
```

Listing 1: CodeProvider interface and retrieving the CodeProvider from the ServiceLoader

1.2 CodeFactory

The *CodeFactory* interface, see lst. 2, provides methods for defining all elements of the structure.

```
public interface CodeFactory {
    ClassModel newClassModel();
    TraitModel newTraitModel();

    AbcModel newAbcModel();
    NumModel newNumModel();
    NuxModel newNuxModel();
    CusModel newCusModel();
    DomModel newDomModel();
    FilModel newFilModel();
    ValModel newValModel();
    GrpModel newGrpModel();
    OccModel newOccModel();
    EmbModel newEmbModel();
    GrpTraitModel newGrpTraitModel();
    OccTraitModel newOccTraitModel();
}
```

Listing 2: CodeFactory interface

1.3 Classes / Interfaces

The first method of the *CodeFactory* interface supplies the definition for a class, see lst. 3, and the second interface method *CodeFactory* provides the definition for an interface, see lst. 4.

```
public interface ClassModel {
    void setName(String name);
    void setLength(int length);
    void setOnOverflow(LoadOverflowAction onOverflow);
    void setOnUnderflow(LoadUnderflowAction onUnderflow);
    void setDoc(Boolean doc);
    void setFields(List<FieldModel> fields);

    void create(String namespace, GenerateArgs ga, FieldDefault defaults);
}
```

Listing 3: ClassModel interface

Both definitions require the name of the structure, its length, whether to generate or not the automatic documentation for the class, the list of fields that compose it and make available a method for generating the source code.

```
public interface TraitModel {
    void setName(String name);
    void setLength(int length);
    void setDoc(Boolean doc);
    void setFields(List<FieldModel> fields);

    void create(String namespace, GenerateArgs ga, FieldDefault defaults);
}
```

Listing 4: TraitModel interface

The class definition also takes two additional parameters to indicate how to behave if the size of the data provided was higher or lower than expected.

Before seeing the detail of the definition of the various fields, let's see the content of the other two parameters required for source code generation.

1.3.1 Global Arguments — GenerateArgs

The *GenerateArgs* class, see lst. 5, allows you to define some general parameters, common to all classes generate. The `sourceDirectory` parameter indicates the root source directory where to generate the sources, the three subsequent parameters identify the program (or plugin) that provided the definition of the layout, these parameters are shown as a comment at the beginning of the generated files.

```
@Builder
public class GenerateArgs {
    @NonNull public final String sourceDirectory;
    public final String group;
    public final String artifact;
    public final String version;
}
```

Listing 5: GenerateArgs class

1.3.2 Field Defaults — FieldDefault

In the definition of classes and fields, some parameters available in their definition change necessarily (the name of the field), others are almost always the same for the same type of field (such as are valid characters

in an alphanumeric field). To simplify the definition of the classes, and related fields, it is possible to omit the "slightly variable" parameters in the definition, however it is necessary to indicate which value to use for these parameters when they are omitted. The *FieldDefault* class, lst. 6, provides some classes dedicated to set the default of the "slightly variable" parameters.

```
@Data
public class FieldDefault {
    private ClsDefault cls = new ClsDefault();
    private AbcDefault abc = new AbcDefault();
    private NumDefault num = new NumDefault();
    private NuxDefault nux = new NuxDefault();
    private FilDefault fil = new FilDefault();
    private CusDefault cus = new CusDefault();
}
```

Listing 6: FieldDefault class

The first default concerns the default behavior of the class when it is created starting from a structure (string), and this has a dimension different from the expected one; if the length of the supplied structure is greater than the expected one, it is possible to throw an exception and ignore the excess content, if the length of the supplied structure is less than the expected one, it is possible to throw an exception or complete the missing part with the default values of the missing part.

```
@Data
public static class ClsDefault {
    private LoadOverflowAction onOverflow = LoadOverflowAction.Trunc;   // enum {Error, Trunc}
    private LoadUnderflowAction onUnderflow = LoadUnderflowAction.Pad;  // enum {Error, Pad}
    private boolean doc = true;
}
```

Listing 7: ClsDefault class

The other defaults allow you to set the default values of some parameters for five types of fields. Not having shown the details of the definition of the various types of field, it is not appropriate to introduce the content of the default classes at this point, they will be shown together with the corresponding field.

Chapter 2

Definition of single fields

In the definition of the class, and of the interface, the list of fields is set as `List<FieldModel>`, but the `FieldModel` interface is an empty box, it is only used to connect all the field definitions to it. In general all fields have an initial position (offset) and a dimension (length); many fields are referable by a name, but not all are necessarily named; when the fields have a name they can be primary or over-define (override) primary fields, in the initialization phase of the fields of a class only primary field definitions are considered.

In the field definitions, the initial position of the field (offset) is set as an `Integer`, ie in general it is not mandatory to set it, it can be calculated automatically by the *Service Provider*.

2.1 Alphanumeric field

An alphanumeric field has 4 basic parameters: `offset`, `length`, `name` and `override`.

The `onOverflow` and `onUnderflow` parameters indicate how the setter should behave when a value with a size greater or less than that foreseen for that field is supplied. The `onOverflow` parameter can assume the values `Error` and `Trunc`, in the first case the code is expected to generate an exception, in the second case the value is found (on the right) ignoring the characters in excess of the expected size. The `onUnderflow` parameter can assume the values `Error` and `Pad`, in the first case the code is expected to generate an exception, in the second case spaces are added (on the right) to reach the expected size.

The `check` parameter allows you to specify checks to restrict the set of characters allowed for the value. This control is activated during the validation of the data string, calling the setters and getters. Possible values are `None`, `Ascii`, `Latin1`, `Valid`, in the first case no check is done, in the second case only ASCII characters are accepted, in the third case only ISO-8859-1[1] characters are accepted, and in the last case valid UTF-8 characters are accepted.

```java
public interface AbcModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setOnOverflow(OverflowAction onOverflow);
    void setOnUnderflow(UnderflowAction onUnderflow);
    void setCheck(CheckAbc check);
    void setNormalize(NormalizeAbcMode normalize);
    void setCheckGetter(Boolean checkGetter);
    void setCheckSetter(Boolean checkSetter);
}
```

Listing 8: interface AbcModel (alphanumeric field)

The `normalize` parameter allows you to indicate how to normalize the value of the field in the getter phase. The `normalize` parameter can take on 3 values `None`, `Trim` and `Trim1`. The first value indicates not to perform any modification of the data, the second value indicates to remove all spaces on the right until a character

[1] more precisely the unicode characters from `\u0020` to `\u007e` and from `\u00a0` to `\u00ff`

other than space is found, if the value is composed only of spaces an empty string is produced, the last value, similarly to the previous value, removes the spaces on the right until a character other than a space is found, but if the value is composed only of spaces, it returns a string composed of a space.

The `checkGetter` parameter indicates whether or not to activate the control indicated with the `check` parameter when the getter is called; if the data-string is previously validated, this check can be deactivated. The `checkSetter` parameter indicates whether or not to activate the control indicated with the `check` parameter when called the setter.

```
@Data
public class AbcDefault {
    private OverflowAction onOverflow = OverflowAction.Trunc;
    private UnderflowAction onUnderflow = UnderflowAction.Pad;
    private CheckAbc check = CheckAbc.Ascii;
    private NormalizeAbcMode normalize = NormalizeAbcMode.None;
    private boolean checkGetter = true;
    private boolean checkSetter = true;
}
```

Listing 9: class AbcDefault (default campo alfanumerico)

The *AbcDefault* class, see lst. 9, set the default values for the parameters `onOverflow`, `onUnderflow`, `check`, `normalize`, `checkGetter`, `checkSetter`, in case they are not set by the client.

2.2 Numeric field

Even a numeric field has 4 basic parameters: `offset`, `length`, `name` and `override`.

The `access` parameter indicates how to generate the setters and getters. In the data-string the numeric field has a representation in string format, in the generated code it is possible to choose whether the setters and getters manage the value as a string (with numeric characters) or to convert the data-string fragment, corresponding to the field, into a native numeric representation (`byte`, `short`, `int`, `long`) or handle both. The `access` parameter can take on the values `String`, `Number` and `Both`. In the first case, setters and getters are generated that handle the value as a (numeric) string, in the second case as a native numeric, and in the last case both are generated (it will be indicated by the provider how to distinguish the string getter from the numeric one). If an access is used that provides string type setter/getter, it is checked in the setter phase that the supplied string is numeric, and in the getter phase that the returned string is numeric.

```
public interface NumModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setAccess(AccesMode access);
    void setWordWidth(WordWidth width);
    void setOnOverflow(OverflowAction onOverflow);
    void setOnUnderflow(UnderflowAction onUnderflow);
    void setNormalize(NormalizeNumMode normalize);
}
```

Listing 10: NumModel interface (numeric field)

The `wordWidth` parameter makes sense only if an access mode has been chosen that generates numeric setters/getters, it basically indicates the minimum size to be used in numeric representations. The `wordWidth` parameter can assume the values `Byte`, `Short`, `Int` and `Long`, the values correspond to the use of the corresponding native types. For example, if a numeric field is represented by a 4-character string, it can be converted into numeric format as `short`, if the `wordWidth` parameter is set to `Int`, `int` type setters/getters are generated; if the parameter value had been `Byte` or `Short`, `short`-type setters/getters would have been generated.

The `onOverflow` and `onUnderflow` parameters indicate how the setter should behave when a value larger or smaller than that expected for that field is supplied. The `onOverflow` parameter can assume the values

`Error` and `Trunc`, in the first case the code is expected to generate an exception, in the second case the value is truncated (on the left) ignoring the digits in excess of the expected size. The `onUnderflow` parameter can assume the values `Error` and `Pad`, in the first case the code is expected to generate an exception, in the second case zeros are added (on the left) to reach the expected size.

The `normalize` parameter makes sense only if an access mode has been chosen that generates string setters/getters, it allows you to indicate how to normalize the value of the field in the getter phase. The `normalize` parameter can take on 2 values `None` and `Trim`. The first value indicates not to perform any modification of the data, the other indicates to remove all the zeros on the left until a digit other than zero is found, if the value is composed only of zeros a string composed of a zero is produced .

```
@Data
public class NumDefault {
    private AccesMode access = AccesMode.String;
    private WordWidth wordWidth = WordWidth.Int;
    private OverflowAction onOverflow = OverflowAction.Trunc;
    private UnderflowAction onUnderflow = UnderflowAction.Pad;
    private NormalizeNumMode normalize = NormalizeNumMode.None;
}
```

Listing 11: class NumDefault (default numeric field)

The *NumDefault* class, see lst. 11, sets the default values for the `access`, `wordWidth`, `onOverflow`, `onUnderflow` and `normalize` parameters if they are not set by the client.

2.3 Custom field (alphanumeric)

Even a custom field has the 4 basic parameters: `offset`, `length`, `name` and `override`.

A custom field is a generalization of an alphanumeric field, and can be configured to emulate a numeric or nullable numeric field. The first sensitive parameter to consider is `align`, the parameter indicates how the `onUnderflow = Pad` field must be aligned. The parameter can assume 2 values `LFT` and `RGT`, the first value indicates that the field must be aligned to the left, the second value must be aligned to the right. The value of this parameter not only affects the `onUnderflow` parameter (indicating from which direction the padding characters must be added), but also on `onOverflow` (indicating from which direction the excess characters must be removed) and `normalize` (indicating from which direction the padding characters be removed).

The `padChar` parameter indicates the padding character to add (in case of `onUnderflow = Pad`) or remove (in case of `normalize = Trim`).

The `initChar` parameter indicates the character to use to initialize the field.

```
public interface CusModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setAlign(AlignMode align);
    void setPadChar(Character padChar);
    void setInitChar(Character initChar);
    void setCheck(CheckCus check);
    void setRegex(String regex);
    void setOnOverflow(OverflowAction onOverflow);
    void setOnUnderflow(UnderflowAction onUnderflow);
    void setNormalize(NormalizeAbcMode normalize);
    void setCheckGetter(Boolean checkGetter);
    void setCheckSetter(Boolean checkSetter);
}
```

Listing 12: CusModel interface (custom field)

For the `check` parameter the same considerations apply as for the corresponding parameter in the alphanumeric case. In this case the possible values are `None`, `Ascii`, `Latin1`, `Valid`, `Digit` and `DigitOrBlank`. The first

four values are identical to the alphanumeric case, the `Digit` value limits the accepted characters to numeric ones (from 0 to 9), as for a numeric field; the `DigitOrBlank` value requires that the characters be numeric or all spaces, such as for a nullable numeric field.

The `regex` parameter can be valued with a regular expression that must be satisfied by the value of the field. If this parameter is present, the `check` parameter is ignored.

```
@Data
public class CusDefault {
    private AlignMode align = AlignMode.LFT;
    private char padChar = ' ';
    private char initChar = ' ';
    private CheckCus check = CheckCus.Ascii;
    private OverflowAction onOverflow = OverflowAction.Trunc;
    private UnderflowAction onUnderflow = UnderflowAction.Pad;
    private NormalizeAbcMode normalize = NormalizeAbcMode.None;
    private boolean checkGetter = true;
    private boolean checkSetter = true;
}
```

Listing 13: class CusDefault (custom field default)

For the parameters `onUnderflow`, `normalize`, `checkGetter`, `checkSetter` the same considerations apply as for the corresponding fields in the alphanumeric case. Be careful because the action of the `onOverflow`, `onUnderflow` and `normalize` parameters also depends on the value of the `align` and `initChar` parameters.

The *CusDefault* class, see lst. 13, sets the default values for the `align`, `padChar`, `initChar`, `check`, `onOverflow`, `onUnderflow`, `normalize`, `checkGetter` and `checkSetter` parameters if they are not set by the client.

2.4 Nullable numeric field

A nullable numeric field is an extension of an ordinary numeric field. The difference is that the space value (all space characters) is allowed in the data-string, this value corresponds to the `null` value in the data-class field.

```
public interface NuxModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setAccess(AccesMode access);
    void setWordWidth(WordWidth width);
    void setOnOverflow(OverflowAction onOverflow);
    void setOnUnderflow(UnderflowAction onUnderflow);
    void setNormalize(NormalizeNumMode normalize);
    void setInitialize(InitializeNuxMode initialize);
}
```

Listing 14: NuxModel interface (nullable numeric field)

As can be seen from the definition, lst. 14 there are the same parameters of a numeric field (lst. 10), with the same meanings, plus one: the `initialize` parameter. This parameter indicates how to initialize the field when the class is created with an empty constructor, with the value spaces (i.e. `null`) or zero.

To manage the default at a general level for this type of field, a dedicated default class is used, which is the copy of the corresponding one for the ordinary numeric case, plus the initialization default.

2.5 Domain field

A domain-type field is an alphanumeric field that can only assume predefined constant values. The domain type field, lst. 16, has the 4 basic parameters: `offset`, `length`, `name` and `override`, plus the `items` parameter

```
@Data
public class NuxDefault {
    private AccesMode access = AccesMode.String;
    private WordWidth wordWidth = WordWidth.Int;
    private OverflowAction onOverflow = OverflowAction.Trunc;
    private UnderflowAction onUnderflow = UnderflowAction.Pad;
    private NormalizeNumMode normalize = NormalizeNumMode.None;
    private InitializeNuxMode initialize = InitializeNuxMode.Spaces;
}
```

Listing 15: NuxDefault (default nullable numeric field)

which will have to provide the list of allowed constant values.

```
public interface DomModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setItems(String[] items);
}
```

Listing 16: DomModel interface (domain field)

For this type of field, no global default makes sense as seen for alphanumeric and numeric fields. The allowed values are those supplied in the `items` parameter, any other value will cause an exception.

2.6 Filler field

A field of type *Filler* is not a real field, has no associated name, does not generate any setter or getter in the data-class, and no control methods. It is a way to indicate that in the data-string there is an area to which no value is associated, or that we are not interested in that area of the data-string.

```
public interface FilModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);

    void setFill(Character fill);
}
```

Listing 17: FilModel interface (filler field)

The filler field has only the 2 basic parameters: `offset` and `length`. It then has the `fill` parameter to indicate with which character to value the field when the data-class is created with the empty constructor.

```
@Data
public class FilDefault {
    private char fill = 0;
}
```

Listing 18: class FilDefault (default filler field)

The filler field has a default class dedicated to indicate the filler character at a general level.

2.7 Field Constant value

A field of type *Constant* is similar to a filler field. As for the filler field, setters and getters are not generated in the data-class, but in the validation of the data-string it is verified that the area corresponding to the field has the indicated constant value.

```
public interface ValModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);

    void setValue(String value);
}
```

Listing 19: ValModel interface (constant range)

Even the constant field has only the 2 basic parameters: `offset` and `length`. It then has the `value` parameter to indicate the expected constant value. The supplied value is used to set the value when the data-class is instantiated with the empty constructor.

Chapter 3

Definition of a group of fields

In some cases it may be useful to group some fields within a container element. This allows you to have within the definition of the string-data structure two fields with the same name within different container elements.

3.1 Field group

A field of type *Group* is not a real field, it is actually a container of other fields.

```
public interface GrpModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setFields(List<FieldModel> fields);
}
```

Listing 20: interfaccia GrpModel (campo gruppo)

A group field has 4 basic parameters: `offset`, `length`, `name` and `override`. In addition to this, it provides the `fields` parameter to provide the list of child fields. The string-data area selected by `offset` and `length` will need to be completely defined by the child fields. It is possible for a child field to use the same name as a field defined at the same level as the parent field.

3.2 Repeating field group

A field of type *Repeating Group* is similar to a group field, except that there are n occurrences of the group. In addition to the parameters used by the group field, there is the `times` parameter, which indicates the number of times the group is repeated.

```
public interface OccModel extends  FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setFields(List<FieldModel> fields);
    void setTimes(int times);
}
```

Listing 21: OccModel interface (repeating group field)

In this case the area of the data-string defined by `offset` and `length` is that of the first occurrence of the group. The dimension actually used is `length` × `times`.

3.3 Fields embedded via interface

An interface (lst. 4) can be defined in the same way as a class (lst. 3). The *Embedded* field is not a real field, it doesn't have a name. Used to indicate that a certain area of the data-string must be interpreted with the fields defined by the indicated interface.

```
public interface EmbModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);

    void setSource(TraitModel source);
}
```

Listing 22: EmbModel interface (embedded field)

The embedded field has only the 2 basic parameters: `offset` and `length`. It then has the `source` parameter to indicate which interface to use to define the fields. The interface must have the same size (`length`) of the embedded field, the initial position of the interface will be translated to adapt it to the value indicated by the embedded field. The generated class will obviously implement the indicated interface.

3.4 Field group defined via interface

The *Group/Interface* field is similar to the embedded field. In this case the fields defined by the interface are not incorporated at the current level, but a group containing them is defined.

```
public interface GrpTraitModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setTypedef(TraitModel typedef);
}
```

Listing 23: GrpTraitModel interface (group/interface field)

A group/interface field has 4 basic parameters: `offset`, `length`, `name` and `override`. It then has the `typedef` parameter to indicate which interface to use to define the fields of the group. The interface must have the same size (`length`) of the group/interface field, the initial position of the interface will be translated to adapt it to the value indicated by the group/interface field. In this case it is the group that implements the indicated interface.

3.5 Repeating field group defined via interface

The *Repeating Group/Interface* field is similar to the group/interface field, except that there are n occurrences of the group. In addition to the parameters used by the group/interface field, there is the `times` parameter, which indicates the number of times the group is repeated.

As in the case of the repeated group, the area of the data-string defined by `offset` and `length` is that of the

```java
public interface OccTraitModel extends FieldModel {
    void setOffset(Integer offset);
    void setLength(int length);
    void setName(String name);

    void setOverride(boolean override);

    void setTypedef(TraitModel typedef);
    void setTimes(int times);
}
```

Listing 24: interface OccTraitModel (repeating group/interface field)

first occurrence of the group. The dimension actually used is `length` × `times`.

Chapter 4

Used enums

Many of the configuration classes or interfaces have fields with values that are limited to some value expressed by enum. Let's see them one by one.

```
public enum LoadOverflowAction { Error, Trunc }
```

Listing 25: enum LoadOverflowAction

The LoadOverflowAction enum is used by the ClassModel class to indicate how to behave when the class is deserialized and the data-string is larger than expected by the data-class.

```
public enum LoadUnderflowAction { Error, Pad }
```

Listing 26: enum LoadUnderflowAction

The LoadUnderflowAction enum is used by the ClassModel class to indicate how to behave when the class is deserialized and the data-string has a smaller size than expected by the data-class.

```
public enum CheckAbc { None, Ascii, Latin1, Valid }
```

Listing 27: enum CheckAbc

The CheckAbc enum is used by the AbcModel class to indicate which characters are considered valid.

```
public enum OverflowAction { Error, Trunc }
```

Listing 28: enum OverflowAction

The OverflowAction enum is used by the class to manage numeric and alphanumeric fields (AbcModel, NumModel, CusModel, NuxModel), to indicate how to behave when the setter proposes a value that has a size greater than the one expected for that field.

```
public enum UnderflowAction { Error, Pad }
```

Listing 29: enum UnderflowAction

The UnderflowAction enum is used by the class to manage numeric and alphanumeric fields (AbcModel, NumModel, CusModel, NuxModel), to indicate how to behave when the setter proposes a value that has a size smaller than the one expected for that field.

```
public enum NormalizeAbcMode { None, Trim, Trim1 }
```

Listing 30: enum NormalizeAbcMode

The NormalizeAbcMode enum is used by the class to manage alphanumeric fields (AbcModel, CusModel), to indicate how to normalize the value returned by the getter.

```
public enum NormalizeNumMode { None, Trim }
```

Listing 31: enum NormalizeNumMode

The NormalizeNumMode enum is used by the class to manage numeric fields (NumModel, NuxModel), to indicate how to normalize the value returned by the getter.

```
public enum WordWidth { Byte, Short, Int, Long }
```

Listing 32: enum WordWidth

The WordWidth enum is used by the class to handle numeric fields (NumModel, NuxModel), to indicate the primitive numeric data type of minimum size to use when creating numeric setters and getters.

```
public enum AccesMode { String, Number, Both }
```

Listing 33: enum AccesMode

The AccesMode enum is used by the class to handle numeric fields (NumModel, NuxModel), to indicate whether to create alphanumeric, numeric, or both setters and getters.

```
public enum InitializeNuxMode { Spaces, Zeroes }
```

Listing 34: enum InitializeNuxMode

The InitializeNuxMode enum is used by the NuxModel class, to indicate how to initialize the field when the data-class is created with the constructor with no arguments.

```
public enum CheckCus { None, Ascii, Latin1, Valid, Digit, DigitOrBlank }
```

Listing 35: enum CheckCus

The CheckCus enum is used by the CusModel class to indicate which characters are considered valid.

```
public enum AlignMode { LFT, RGT }
```

Listing 36: enum AlignMode

The CheckCus enum is used by the CusModel class to indicate how to align the field.

Part II

Service

Service

The *Service Provider Interface* simply fixes the general structure, but contains only interfaces and java-beans. The *Service* is the application that allows the user to provide the input required by the *Service Provider Interface*. To produce the output, the *Service* will use the *ServiceLoader* to search the classpath for a *Service Provider* that implements the *Service Provider Interface*, and the *Service Provider* will generate the output from the input.

The heart of the *Service Provider Interface*, `recfm-addon-api`, is the *CodeProvider* interface. The implementation of the interface is searched with the ServiceLoader mechanism, lst. 37.

```
ServiceLoader<CodeProvider> loader = ServiceLoader.load(CodeProvider.class);
CodeProvider codeProvider = loader.iterator().next();
CodeFactory factory = codeProvider.getInstance();
```

Listing 37: retrieve of the CodeProvider

Once an instance of *CodeFactory* has been retrieved, it is possible to create the definitions of the data-strings and generate the sources of the data-classes.

Two *clients* have been developed, one in the form of a maven plugin (`recfm-maven-plugin`), and the other in the form of a gradle plugin (`recfm-gradle-plugin`). The code is largely identical, only the trigger mechanism changes.

Chapter 5

Maven plugin

The `recfm-maven-plugin` maven plugin allows you to create multiple classes and interfaces using one or more configuration files in yaml format. The external libraries used require java 8, so you need at least java 8 to run this plugin.

The plugin expects configuration parameters as parameters:

```
@Parameter(defaultValue = "${project.build.directory}/generated-sources/recfm",
    property = "maven.recfm.generateDirectory", required = true)
private File generateDirectory;

@Parameter(defaultValue = "${project.build.resources[0].directory}",
    property = "maven.recfm.settingsDirectory", required = true)
private File settingsDirectory;

@Parameter(required = true)
private String[] settings;

@Parameter(defaultValue = "true", property = "maven.recfm.addCompileSourceRoot")
private boolean addCompileSourceRoot = true;

@Parameter(defaultValue = "false", property = "maven.recfm.addTestCompileSourceRoot")
private boolean addTestCompileSourceRoot = false;

@Parameter
private String codeProviderClassName;
```

Listing 38: settable parameters of the maven plugin

- The `generateDirectory` field indicates the root directory to use for generating the sources, it is used to set the value of the `sourceDirectory` field of the `GenerateArgs` class, as can be seen from the definition, if the parameter is omitted the `target/generated-sources/recfm` directory is used, normally the default value can be left. The other three parameters of `GenerateArgs` are an identifier of the *service* program and are set automatically.
- The `settingsDirectory` field indicates the directory containing the configuration files, if the parameter is omitted the value `src/main/resources` is used, normally the default value can be left.
- The `settings` field indicates the list of configuration files to be used to generate the classes/interfaces; the parameter must be supplied to the plugin.
- The `addCompileSourceRoot` field is a technical field, it indicates to maven that the directory where the sources were generated must be included among those used for the main compilation, if the parameter is omitted the value `true` is used; the `true` value is appropriate when a code generation directory other than `src/main/java` is used, otherwise it is necessary to use additional plugins to add the new path to the maven compilation path.

- The `addTestCompileSourceRoot` field is similar to the previous field, but adds the generation directory to the test compilation path, if omitted the value `false` is used, except in special cases the default value can be left.
- The `codeProviderClassName` field indicates which is the concreated class that implements the *Service Interface*, if omitted the "first" implementation of the *ServiceLoader* retrieved is used; if there is only one implementation in the classpath, it is not necessary to set the parameter.

```xml
<plugin>
    <groupId>io.github.epi155</groupId>
    <artifactId>recfm-maven-plugin</artifactId>
    <version>0.7.0</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <settings>
                    <setting>foo.yaml</setting>
                </settings>
            </configuration>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>io.github.epi155</groupId>
            <artifactId>recfm-java-addon</artifactId>
            <version>0.7.0</version>
        </dependency>
    </dependencies>
</plugin>
```

Listing 39: minimal example of plugin execution

An example of plugin execution is shown in the lst. 39, the plugin to be executed must have as dependency a library that provides the implementation of the interface, otherwise the `ServiceLoader` does not find anything and the plugin terminates in error.

All other parameters are provided in the configuration files.

5.1 Configuration file structure

To manage the layout configuration the plugin defines the *MasterBook* class, see lst. 40, is divided into two components, the first `defaults` is simply the java-bean *FieldDefault* (see 6) made available by the *Service Provider Interface* to provide the default values of the "slightly variable" parameters of the classes and fields.

```java
@Data
public class MasterBook {
    private FieldDefault defaults = new FieldDefault();
    private List<ClassPackage> packages = new ArrayList<>();
}
```

Listing 40: MasterBook configuration class

To simplify the enhancement of the yaml configuration file, a feature of the yaml libraries is used, which allows you to define abbreviated or alternative names of the parameters and values of the enum type fields. The component details of the `defaults` field will be shown along with the field it supplies the default parameter value to.

The second component of *MasterBook*, `packages`, is a list of *ClassPackage* (41), i.e. packages within which a list of interfaces and classes are defined. Expanding an example of this object in yaml format (with relative default) we have:

```java
@Data
public class ClassPackage {
    private String name;     // package name
    private List<TraitModel> interfaces = new ArrayList<>();
    private List<ClassModel> classes = new ArrayList<>();
}
```

Listing 41: classe di configurazione ClassPackage

```yaml
defaults:
  cls:
    onOverflow: Trunc   # :ovf: Error, Trunc
    onUnderflow: Pad    # :unf: Error, Pad
    doc: true
packages:
  - name: com.example.test  # package name
    interfaces:
      - &IFoo         # interface reference
        name: IFoo    # interface name
        length: 12    # :len: interface length
        fields:
          - ...
    classes:
      - name: Foo           # class name
        length: 10          # :len: class length
        onOverflow: Trunc   # :ovf: Error, Trunc
        onUnderflow: Pad    # :unf: Error, Pad
        fields:
          - ...
```

Listing 42: configuration, packages / interfaces / classes areas

The possible alternative names of the fields and the list of allowed *enum* values are shown in the comments. If no interfaces are used, the `interfaces` node can be omitted. For both classes and interfaces, the name and length of the path to be associated must be set by the user, in the definition of the class the behavior can also be set if a structure with a larger size is provided during the de-serialization phase or less than expected.

	interfaces TraitModel				
	attribute	alt	type	O	default
	`name`		`String`	✔	
	`length`	`len`	`int`	✔	
ClsDefault	`doc`		`boolean`		`${defaults.cls.doc:true}`
	`fields`		`array`	✔	

Table 5.1: Attributes that can be set for defining an interface

Although all field types have a starting position and length, the detail of the configuration parameters varies from field to field and it is necessary to enter the configuration parameters field by field.

To explicitly indicate the type of field used, *tag* are introduced to be associated with each field. Table 5.3 shows the tags associated with each type of field.

> **With regard to the offset of the fields, it should be noted that some characteristics do not depend on the *Service*, but on the *Service Provider*: the minimum offset can be zero or one, the offset setting can be mandatory or optional (the offset can be calculated automatically using the offset and length of the previous field), or not allowed.**
>
> **The *Service Provider* described in § 6.1 uses a minimum offset of 1 and setting the offset is optional. If you omit the offset in a field defined with an override, it is assumed that the field overrides the field that precedes it in the structure definition. When defining interfaces the use of the offset is optional, but unlike classes, which require a minimum offset of 1, any initial value can be used for interfaces, the effective offset is corrected when the interface is applied to the class.**

	classes	ClassModel			
	attribute	alt	type	O	default
	name		String	✔	
	length	len	int	✔	
ClsDefault	onOverflow	ovf	enum		${defaults.cls.onOverflow:Trunc}
	onUnderlow	unf	enum		${defaults.cls.onUnderflow:Pad}
	doc		boolean		${defaults.cls.doc:true}
	fields		array	✔	

Table 5.2: Attributes that can be set for the definition of a class

Field definition tag		
tag	class	note
!Abc	AbcModel	alphanumeric field
!Num	NumModel	numeric field
!Cus	CusModel	custom field
!Nux	NuxModel	Nullable numeric field
!Dom	DomModel	domain field
!Fil	FilModel	filler field
!Val	ValModel	constant field
!Grp	GrpModel	field group of fields
!Occ	OccModel	field group of repeating fields
!Emb	EmbModel	fields embedded via interface
!GRP	GrpTraitModel	field group of fields via interface
!OCC	OccTraitModel	field group of repeating fields via interface

Table 5.3: Yaml tag for field identification

5.2 Single fields

5.2.1 Alphanumeric field

The yaml definition of the alphanumeric field reflects the structure imposed by the service provider interface, see 8. An alphanumeric field is specified by indicating the `!Abc` *tag* , an example of defining alphanumeric fields is shown in lst. 43, the example also shows the global default node for alphanumeric fields, the values set are the default ones of the *service provider interface*, so it is not necessary to explicitly set the parameters if you want to set these values.

In the example, the default node of alphanumeric fields is set using the canonical names of the parameters. The *plugin* uses an available functionality of the library to read the yaml file, and defines short parameter names, which can be used as an alternative to the canonical names.

Table 5.4 shows all the attributes expected for an alphanumeric field, the relative abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

5.2.2 Numeric field

The yaml definition of the numeric field reflects the structure imposed by the service interface, see 10. A numeric field is specified by indicating the `!Num` *tag*, an example of defining numeric fields is shown in the lst. 44, the example also shows the global default node for numeric fields, the values set are the default ones of the *service provider interface*, so it is not necessary to explicitly set the parameters if you want to set these values.

Table 5.5 shows all the attributes expected for a numeric field, the relative abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value. Even if the `acccess` and `wordWidth` parameters were introduced in § 2.2, I remind you that a “numeric” field can be managed as a

```yaml
defaults:
  abc:
    check: Ascii        # :chk: None, Ascii, Latin1, Valid
    onOverflow: Trunc   # :ovf: Error, Trunc
    onUnderflow: Pad    # :unf: Error, Pad
    normalize: None     # :nrm: None, Trim, Trim1
    checkGetter: true   # :get:
    checkSetter: true   # :set:
packages:
  - name: com.example.test
    classes:
      - name: Foo3111
        length: 55
        fields:
          - !Abc { name: firstName   , at:  1, len: 15 }
          - !Abc { name: lastName    , at: 16, len: 15 }
          - !Num { name: birthDate   , at: 31, len:  8 }
          - !Abc { name: birthPlace  , at: 39, len: 14 }
          - !Abc { name: birthCountry, at: 53, len:  3 }
```

Listing 43: example of definition of alphanumeric fields

	!Abc: AbcModel				
	attribute	alt	type	O	default
	offset	at	int	✔	self-calculated
	length	len	int	✔	
	name		String	✔	
	override	ovr	boolean		false
AbcDefault	onOverflow	ovf	enum		${defaults.abc.onOverflow:Trunc}
	onUnderlow	unf	enum		${defaults.abc.onUnderflow:Pad}
	check	chk	enum		${defaults.abc.check:Ascii}
	normalize	nrm	enum		${defaults.abc.normalize:None}
	checkGetter	get	boolean		${defaults.abc.checkGetter:true}
	checkSetter	set	boolean		${defaults.abc.checkSetter:true}

Table 5.4: Attributes that can be set for an alphanumeric field

```yaml
defaults:
  num:
    onOverflow: Trunc   # :ovf: Error, Trunc
    onUnderflow: Pad    # :unf: Error, Pad
    normalize: None     # :nrm: None, Trim
    wordWidth: Int      # :wid: Byte(1,byte), Short(2,short), Int(4,int), Long(8,long)
    access: String      # :acc: String(Str), Numeric(Num), Both(All)
packages:
  - name: com.example.test
    classes:
      - name: Foo3112
        length: 8
        doc: No
        fields:
          - !Num { name: year , at: 1, len: 4 }
          - !Num { name: month, at: 5, len: 2 }
          - !Num { name: mday , at: 7, len: 2 }
```

Listing 44: example of definition of numeric fields

string (where only numeric characters are allowed), or converted into a native numeric format, or both. The `access` parameter indicates whether to create string setters/getters only, create numeric setters/getters only, or both. In case a native number representation is used, the `wordWidth` parameter indicates the minimum size

	!Num: NumModel				
	attribute	alt	type	O	default
	offset	at	int	✔	self-calculated
	length	len	int	✔	
	name		String	✔	
	override	ovr	boolean		false
NumDefault	onOverflow	ovf	enum		${defaults.num.onOverflow:Trunc}
NumDefault	onUnderlow	unf	enum		${defaults.num.onUnderflow:Pad}
NumDefault	access	acc	enum		${defaults.num.access:String}
NumDefault	wordWidth	wid	enum		${defaults.num.wordWidth:Int}
NumDefault	normalize	nrm	enum		${defaults.num.normalize:None}

Table 5.5: Attributes that can be set for a numeric field

native representation to use. In general the *Service Provider* will select the size of the native representation based on the size of the data-string that will end up r epresenting the value of the field.

5.2.3 Custom field (alphanumeric)

The yaml definition of the custom field reflects the structure imposed by the service provider interface, see 12. A custom field is specified by indicating the `!Cus` *tag*, an example of defining custom fields is shown in the lst. 45, in the example the global default node for the custom fields is also shown, the values set are the default ones of the *service provider interface*, therefore it is not necessary to explicitly set the parameters if you want to set these values.

A custom field is an extension of an alphanumeric field. An alphanumeric field is necessarily left-aligned, right-truncated/trimmed, right-padded with spaces, initialized to spaces. In a custom field it is possible to choose the alignment of the field, the padding and initialization character; it has an extended `check` with respect to the alphanumeric one, finally, the `regex` attribute can be used to validate the values allowed for the field (instead of the one defined with `check`).

```yaml
defaults:
  cus:
    padChar: ' '        # :pad:
    initChar: ' '       # :ini:
    check: Ascii        # :chk: None, Ascii, Latin1, Valid, Digit, DigitOrBlank
    align: LFT          # LFT, RGT
    onOverflow: Trunc   # :ovf: Error, Trunc
    onUnderflow: Pad    # :unf: Error, Pad
    normalize: None     # :nrm: None, Trim, Trim1
    checkGetter: true   # :get:
    checkSetter: true   # :set:
packages:
  - name: com.example.test
    classes:
      - name: Foo3113
        length: 8
        doc: No
        fields:
          - !Cus { name: year , at: 1, len: 4 }
          - !Cus { name: month, at: 5, len: 2 }
          - !Cus { name: mday , at: 7, len: 2 }
```

Listing 45: example of definition of custom fields

Table 5.6 shows all the attributes expected for a custom field, the related abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

	!Cus: CusModel				
	attribute	alt	type	O	default
	offset	at	int	✔	self-calculated
	length	len	int	✔	
	name		String	✔	
	override	ovr	boolean		false
CusDefault	onOverflow	ovf	enum		${defaults.cus.onOverflow:Trunc}
CusDefault	onUnderlow	unf	enum		${defaults.cus.onUnderflow:Pad}
CusDefault	padChar	pad	char		${defaults.cus.pad:' '}
CusDefault	initChar	ini	char		${defaults.cus.ini:' '}
CusDefault	check	chk	enum		${defaults.cus.check:Ascii}
CusDefault	align		enum		${defaults.cus.align:LFT}
CusDefault	normalize	nrm	enum		${defaults.cus.normalize:None}
CusDefault	checkGetter	get	boolean		${defaults.cus.checkGetter:true}
CusDefault	checkSetter	set	boolean		${defaults.cus.checkSetter:true}
	regex		String		null

Table 5.6: Attributes that can be set for a custom field

5.2.4 Nullable numeric field

The yaml definition of the nullable numeric field reflects the structure imposed by the service interface, see 14. A nullable numeric field is specified by indicating the `!Nux` *tag*, an example of the definition of nullable numeric fields is shown in the lst. 46, the example also shows the global default node for nullable numeric fields, the values set are the default ones of the *service provider interface*, so it is not necessary to explicitly set the parameters if you want to set these values.

```yaml
defaults:
  nux:
    onOverflow: Trunc   # :ovf: Error, Trunc
    onUnderflow: Pad    # :unf: Error, Pad
    normalize: None     # :nrm: None, Trim
    wordWidth: Int      # :wid: Byte(1,byte), Short(2,short), Int(4,int), Long(8,long)
    access: String      # :acc: String(Str), Numeric(Num), Both(All)
    initialize: Spaces  # :ini: Spaces(Space), Zeroes(Zero)
packages:
  - name: com.example.test
    classes:
      - name: Foo3114
        length: 8
        doc: No
        fields:
          - !Nux { name: year , at: 1, len: 4 }
          - !Nux { name: month, at: 5, len: 2 }
          - !Nux { name: mday , at: 7, len: 2 }
```

Listing 46: example of definition of nullable numeric fields

A nullable numeric field is an extension of an ordinary numeric field, the difference is that in the string-data representation it can assume the value space (all spaces), which corresponds to the `null` value in the data class. Consequently, in the definition of the field there is an additional parameter to indicate whether the field must be initialized to `null` or to zero when the data-class is created with the empty constructor.

Table 5.7 shows all the attributes expected for a numeric field, the relative abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

	!Nux: NuxModel				
	attribute	alt	type	O	default
	offset	at	int	✔	self-calculated
	length	len	int	✔	
	name		String	✔	
	override	ovr	boolean		false
NuxDefault	onOverflow	ovf	enum		${defaults.nux.onOverflow:Trunc}
NuxDefault	onUnderlow	unf	enum		${defaults.nux.onUnderflow:Pad}
NuxDefault	access	acc	enum		${defaults.nux.access:String}
NuxDefault	wordWidth	wid	enum		${defaults.nux.wordWidth:Int}
NuxDefault	normalize	nrm	enum		${defaults.nux.normalize:None}
NuxDefault	initialize	ini	enum		${defaults.nux.initialize:Space}

Table 5.7: Attributes that can be set for a nullable numeric field

5.2.5 Domain field

The yaml definition of the numeric field reflects the structure imposed by the service interface, see 16. A domain field is specified by indicating the `!Dom` *tag*, an example of domain field definition is shown in the lst. 47, this field type has no global default.

```yaml
packages:
  - name: com.example.test
    classes:
      - name: Foo3115
        length: 12
        doc: No
        fields:
          - !Num { name: year , at: 1, len: 4 }
          - !Dom { name: month, at: 5, len: 3,
                   items: [ Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec ] }
          - !Num { name: mday , at: 8, len: 2 }
          - !Dom { name: wday , at: 10, len: 3,
                   items: [ Sun, Mon, Tue, Wed, Thu, Fri, Sat ] }
```

Listing 47: example of definition of domain fields

A domain field is substantially an alphanumeric field, which can assume only a limited number of values.

!Dom: DomModel				
attribute	alt	type	O	default
offset	at	int	✔	self-calculated
length	len	int	✔	
name		String	✔	
override	ovr	boolean		false
items		array	✔	

Table 5.8: Attributes that can be set for a domain field

Table 5.8 shows all the attributes expected for a domain field, the related abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value. When the data-class is created with the empty constructor, the field is initialized with the first value among those provided in the list of possible values.

5.2.6 Filler Field

The yaml definition of the filler field reflects the structure imposed by the service interface, see 17. A filler field is specified by indicating the `!Fil` *tag*, an example of definition of filler fields is shown in the lst. 48, in the example the global default node for the filler fields is also shown, the set value is the default one of the service provider interface, therefore it is not necessary to explicitly set the parameter if you want to set this value.

```yaml
defaults:
  fil:
    fill: 0             # \u0000
packages:
  - name: com.example.test
    classes:
      - name: Foo3116
        length: 10
        doc: No
        fields:
          - !Num { at: 1, len: 4, name: year }
          - !Fil { at: 5, len: 1, fill: '-' }
          - !Num { at: 6, len: 2, name: month }
          - !Fil { at: 8, len: 1, fill: '-' }
          - !Num { at: 9, len: 2, name: mday }
```

Listing 48: example of definition of filler fields

A filler field is not a real field, the setters/getters are not generated, no check is made on the value of the corresponding data-string. It simply indicates an area of the data-string that we are not interested in, but which must be present in the definition of the structure in order not to leave undefined areas.

	`!Fil`: FilModel				
	attribute	alt	type	O	default
	`offset`	`at`	`int`	✔	self-calculated
	`length`	`len`	`int`	✔	
FilDefault	`fill`		`char`		`${defaults.fil.fill:0}`

Table 5.9: Attributes that can be set for a filler field

Table 5.9 shows all the attributes expected for a filler field, the relative abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

5.2.7 Constant field

The yaml definition of the constant field reflects the structure imposed by the service interface, see 19. A constant field is specified by indicating the `!Val` *tag*, an example of definition of cotant fields is shown in the lst. 49, this type of field has no global defaults.

```yaml
packages:
  - name: com.example.test
    classes:
      - name: Foo3117
        length: 10
        fields:
          - !Num { at: 1, len: 4, name: year }
          - !Val { at: 5, len: 1, val: "-" }
          - !Num { at: 6, len: 2, name: month }
          - !Val { at: 8, len: 1, val: "-" }
          - !Num { at: 9, len: 2, name: mday }
```

Listing 49: example of definition of constant fields

A constant field can be thought of as a variant of a filler field, or as a domain field with only one value. Setters/getters are not generated for this type of field, but the field is checked to verify that the data-string corresponding to the field has the expected value.

!Val: ValModel				
attribute	alt	type	O	default
offset	at	int	✔	self-calculated
length	len	int	✔	
value	val	string	✔	

Table 5.10: Attributes that can be set for a constant range

Table 5.10 shows all the attributes required for a constant field, the relative abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

5.3 Manifold fields

In some cases it is useful to group some fields within a context containing element. This way you can use the same field name in different contexts. A manifold field has no global defaults.

5.3.1 Field group of fields

The yaml definition of the field group field reflects the structure imposed by the service interface, see 20. A field group field is specified by indicating the `!Grp` *tag*, an example of this field definition is shown in the lst. 50.

```
packages:
  - name: com.example.test
    classes:
      - name: Foo3118
        length: 12
        fields:
          - !Grp { name: startTime, at: 1, len: 6, fields: [
            !Num { name: hours  , at: 1, len: 2 },
            !Num { name: minutes, at: 3, len: 2 },
            !Num { name: seconds, at: 5, len: 2 }
            ] }
          - !Grp { name: stopTime , at: 7, len: 6, fields: [
            !Num { name: hours  , at:  7, len: 2 },
            !Num { name: minutes, at:  9, len: 2 },
            !Num { name: seconds, at: 11, len: 2 }
            ] }
```

Listing 50: example definition group of fields

!Grp: GrpModel				
attribute	alt	type	O	default
offset	at	int	✔	self-calculated
length	len	int	✔	
name		String	✔	
override	ovr	boolean		false
fields		array	✔	

Table 5.11: Attributes that can be set for a group of fields

Table 5.11 shows all the attributes expected for a group of fields, their abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

5.3.2 Field group of repeating fields

The yaml definition of the repeating field group field reflects the structure imposed by the service interface, see 21. A repeating field group field is specified by indicating the `!Occ` *tag*, an example of this field definition is shown in the lst. 51.

```yaml
packages:
  - name: com.example.test
    classes:
      - name: Foo3119
        length: 590
        fields:
          - !Num { name: nmErrors, at: 1, len: 2 }
          - !Occ { name: tabError, at: 3, len: 49, x: 12, fields: [
            !Abc { name: status  , at:  3, len:  5 },
            !Num { name: code    , at:  8, len:  4 },
            !Abc { name: message , at: 12, len: 40 }
          ] }
```

Listing 51: example definition group of repeated fields

!Occ: OccModel				
attribute	alt	type	O	default
offset	at	int	✔	self-calculated
length	len	int	✔	
name		String	✔	
override	ovr	boolean		false
times	x	int	✔	
fields		array	✔	

Table 5.12: Attributes that can be set for a repeating field group

Table 5.12 shows all the attributes expected for a field group of repeated fields, the related abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

5.3.3 Fields embedded via interface

The yaml definition of embedded fields via interface reflects the structure imposed by the service interface, see 22. An embedded field is specified by indicating the `!Emb` *tag*, an example of this field definition is shown in the lst. 52.

Interfacing embedded fields unlike other manifold fields does not create an explicit context item. Fields are children of the current structure, not of a context item. But the current element implements the interface and this creates an implicit context.

!Emb: EmbModel				
attribute	alt	type	O	default
offset	at	int	✔	self-calculated
length	len	int	✔	
source	src	interface	✔	

Table 5.13: Attributes that can be set for embedded fields via interface

Table 5.13 shows all the attributes expected for an embedded fields via interface, the relative abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

```yaml
packages:
  - name: com.example.test
    interfaces:
      - &Time
        name: ITime
        len: 6
        fields:
          - !Num { name: hours  , len: 2 }
          - !Num { name: minutes, len: 2 }
          - !Num { name: seconds, len: 2 }
    classes:
      - name: Foo311a
        length: 14
        fields:
          - !Num { name: year , at: 1, len: 4 }
          - !Num { name: month, at: 5, len: 2 }
          - !Num { name: mday , at: 7, len: 2 }
          - !Emb { src: *Time , at: 9, len: 6 }
```

Listing 52: example definition of embedded fields via interface

5.3.4 Field group of fields via interface

The yaml definition of the interface-group field reflects the structure imposed by the service interface, see 23. A group/interface field is specified by indicating the `!GRP` *tag*, an example of a definition of this field is shown in the lst. 53.

```yaml
packages:
  - name: com.example.test
    interfaces:
      - &Time
        name: ITime
        len: 6
        fields:
          - !Num { name: hours  , len: 2 }
          - !Num { name: minutes, len: 2 }
          - !Num { name: seconds, len: 2 }
    classes:
      - name: Foo311b
        length: 12
        fields:
          - !GRP { name: startTime, at: 1, len: 6, as: *Time }
          - !GRP { name: stopTime , at: 7, len: 6, as: *Time }
```

Listing 53: example definition of interface-group fields

An interface-group field is similar to a group of fields, the difference is that the fields of the group are not defined individually, but all together by importing them from the interface. The group will implement the interface.

!GRP: GrpTraitModel				
attribute	alt	type	O	default
`offset`	`at`	`int`	✔	self-calculated
`length`	`len`	`int`	✔	
`name`		`String`	✔	
`override`	`ovr`	`boolean`		`false`
`typedef`	`as`	`interface`	✔	

Table 5.14: Attributi impostabili per un gruppo da interfaccia

Table 5.14 shows all the attributes expected for an interface-group field, the relative abbreviated names,

the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

5.3.5 Field group of repeating fields via interface

The yaml definition of the repeating interface-group field reflects the structure imposed by the service interface, see 24. A repeating interface-group field is specified by indicating the `!OCC` *tag*, an example definition of this field is shown in the lst. 54.

```yaml
packages:
  - name: com.example.test
    interfaces:
      - &Error
        name: IError
        len: 49
        fileds:
          - !Abc { name: status  , at:  1, len:  5}
          - !Num { name: code    , at:  6, len:  4}
          - !Abc { name: message , at: 10, len: 40}
    classes:
      - name: Foo311c
        length: 590
        fields:
          - !Num { name: nmErrors, at: 1, len: 2}
          - !OCC { name: tabError, at: 3, len: 49, x: 12, as: *Error }
```

Listing 54: example definition of the repeated interface-group field

A repeating interface-group field is similar to a repeating field group field, the difference is that the fields of the group are not defined individually, but all together by importing them from the interface. The group will implement the interface.

!OCC: OccTraitModel				
attribute	alt	type	O	default
offset	at	int	✔	self-calculated
length	len	int	✔	
name		String	✔	
override	ovr	boolean		false
times	x	int	✔	
typedef	as	interface	✔	

Table 5.15: Attributes that can be set for a repeating interface-group

Table 5.15 shows all the attributes expected for a group field repeated by the interface, the relative abbreviated names, the corresponding data-type, whether the attribute is mandatory or optional, and any default value.

Part III

Service Provider

Chapter 6

Service Provider

In the previous chapters we have seen the *Service Provider Interface*, which defines interfaces and classes that allow you to define the layouts, and indicate some behaviors that must be used when using the layouts; and some examples of *Service*, which simply values the objects made available to the *Service Provider Interface*, but the real work of generating the code is done by the *Service Provider*.

The SPI structure allows you to have code generated differently, implemented differently, or even generate source in a different language.

Whatever the language generated and the detail of the implementation, the *Service Provider* will have to provide some general functions.

- **decode**: starting from the data-string, it must instantiate the data-class;
- **setter, getter**: the generated data-class must provide access methods to the individual fields;
- **empty constructor**: the data-class can be instantiated with the default values of the fields;
- **encode**: data-class can be serialized into data-string.

Some additional features would also be welcome:

- **validate**: validate the data-string before de-serialization, in order to signal all the areas that cannot be assigned to the relative fields, typically non-numerical characters in numeric-type fields;
- **cast**: if two data-strings have the same length, being able to pass from one data-class that represents them to another;
- **toString**: provide a method that displays all the values of the fields that make up the data-class (in a human readable way, not the encode method);
- (deep) **copy**: generates a copy of the data-class;.

6.1 Java source code generation — `java-addon`

The classes generated by the java *CodeProvider* in addition to the setters and getters have a series of auxiliary methods, see lst. 55.

- a no-argument constructor is provided, which creates the class with default values;
- a cast-like constructor is provided, which takes any other class representing a data-class as an argument;
- a constructor is provided from string-data (deserializer);
- a *deep-copy* method is provided to duplicate the data-class;
- a validation method is provided;
- a `toString` method is provided;
- a method is provided to generate the data-string (serializer);
- a method is provided to retrieve the length of the data-string.

Classes generated from configuration files inherit general classes with common methods for handling control setter/getter and validations. These classes are provided as an external library, see 56.

These libraries are compiled in java-5 compatibility, and contain the *module-info* in order to be correctly managed also with java-9 and higher.

```java
public class Foo312 extends FixRecord {
    public Foo312() { /* ... */ }
    public static Foo312 of(FixRecord r) { /* ... */ }
    public static Foo312 decode(String s) { /* ... */ }
    public Foo312 copy() { /* ... */ }
    // setter and getter ...
    public boolean validateFails(FieldValidateHandler handler) { /* ... */ }
    public boolean validateAllFails(FieldValidateHandler handler) { /* ... */ }
    public String toString() { /* ... */ }
    public String encode() { /* ... */ }    // from super class
    public int length() { return LRECL; }   // string-data length
}
```

Listing 55: example of generated class (Foo312)

```xml
<dependencies>
  <dependency>
    <groupId>io.github.epi155</groupId>
    <artifactId>recfm-java-lib</artifactId>
    <version>0.7.0</version>
  </dependency>
</dependencies>
```

Listing 56: addon-java runtime dependencies

6.1.1 Validation

As seen in the lst. 55 a validation method is generated for each class. The argument is a dedicated interface, but this is for pre-java-8 compatibility. The argument will be a *closure*, implemented with an anonymous or internal class or a λ-function.

```java
public interface FieldValidateHandler {
    void error(FieldValidateError fieldValidateError);
}
```

Listing 57: error handler `FieldValidateHandler`

The validation methods indicate whether the data-string acquired with the `decode` static constructor has passed the validation required by the definition of the fields or not, but every time a validation error is detected the `error` method of the interface supplied as an argument is called with the details of the error. In this way it is possible to accumulate all the validation errors detected. The difference between the two methods is that the first `validateFails` in case of multiple wrong characters on the same field, signals only the first, while the second `validateAllFails` signals all the characters in error.

```java
public interface FieldValidateError {
    String name();          // field name in error
    int offset();           // field offset in error
    int length();           // field length in error
    String value();         // field value in error
    Integer column();       // column of the record with the wrong character
    ValidateError code();   // error category
    Character wrong();      // wrong character
    String message();       // field message error
}
```

Listing 58: error detail `FieldValidateError`

The argument to the `error` method is the `FieldValidateError` interface, which is basically a java-bean that only exposes getters in *fluent* format. Please note that some values may be `null`. A field of type constant

has no name. A field of type constant or domain or custom with a control set with a regular expression does not have a wrong character in a precise column. In the error message, if it is possible to identify the character in error, the position of the character relative to the field (not to the data-string) is shown, the character (if it is a control character the unicode encoding is shown), the *name* of the character, and type of error; otherwise the value of the field and the type of error are shown.

```
public enum ValidateError {
    NotNumber, NotAscii, NotLatin, NotValid, NotDomain, NotBlank, NotEqual, NotMatch, NotDigitBlank
}
```

Listing 59: error category `ValidateError`

Possible error types are shown in lst. 59, the meaning is evident from the name.

6.1.2 Setters and getters

The data-class implementation used by this library does not actually generate deserialized fields. When the class is created from the data-string, the static constructor just internally saves the data-string as an array of characters. The getter of a field accesses the range of characters corresponding to the field and deserializes them on the fly. Similarly the setter serializes the supplied value and copies it in the range of characters corresponding to the field. In this way it is trivial to *override* a field, the *cast-like* constructor and *deep-copy* are almost free of cost. The *encode* and *decode* methods are also basically free of cost because the serialization/deserialization operations are actually performed by the setters/getters.

6.1.3 Single fields

`null` value handling

A null value cannot be represented in a data-string, unless conventionally this value is assigned to a particular string, as in fields of the numeric-nullable type. When a setter formally sets the `null` value, in the data-string representation it will actually be assigned the default value of the field: space for an alphanumeric type, zero for a numeric type, the `initChar` for a custom type, and the first value between those defined as possible for a domain field.

```
String getValue() { /* ... */ }   // string getter
int intValue() { /* ... */ }      // int getter
```

Listing 60: Access to numeric values such as strings and primitive numbers

`Both` access for numeric fields

Numeric fields can be handled as strings of numeric characters or as primitive numeric objects. Fields can be configured to have getters/setters of type string or primitive numeric or both. If “both” is chosen, it is not possible to define the getter with the canonical name for both types. In this case the canonical name is used for the getter of type string, the getter with the primitive type has as its name the primitive type and the field name, see 60.

Controls on setters and getters

If the checks on the setters are active and an illegal value is set, an exception is thrown which signals the violation of the check. The exception places the stacktrace on the setter instruction.

Similarly on getters. If the data-string contains an invalid value for the field in the area corresponding to a field, the validation of the structure, which would have signaled the problem, is not performed, and the code continues until the getter, an exception is thrown. The exception places the stacktrace on the getter instruction.

```
200     @Test
201     void testDomain() {
202         BarDom dom = new BarDom();
203         dom.setCur("AAA");
204     }
```

```
io.github.epi155.recfm.java.NotDomainException: com.example.BarDom.setCur, offending value "AAA"
    at com.example.test.TestBar.testDomain(TestBar.java:203)
    ...
```

Listing 61: Exception on the setter

```
300     @Test
301     void testDomain() {
302         BarDom d1 = BarDom.decode("AAA");
303         String cur = d1.getCur();
304     }
```

```
io.github.epi155.recfm.java.NotDomainException: com.example.BarDom.getCur, offending value "AAA" @1+3
    at com.example.test.TestBar.testDomain(TestBar.java:303)
    ...
```

Listing 62: Exception on the getter

6.1.4 Manifold Fields

n this context we will consider repeating fields only those fields of type group or repeated group, defined directly or through an interface. This type of fields generates an intermediate element. As seen from lst. 63, generated for a group defined via an interface, an internal class is created to manage the intermediate element, a private field with an instance of the intermediate element, a fluent getter of the field, and a *Consumer* of the field.

```
public class StopTime implements Validable, ITime {/* ... */}
private final StopTime stopTime = this.new StopTime();
public StopTime stopTime() { return this.stopTime; }
public void withStopTime(WithAction<StopTime> action) { action.accept(this.stopTime); }
```

Listing 63: Implementation of a group inside the data-class

The inner class will implement the validation interface, and, if defined via interface, the interface with the detail definition of the fields of the inner class. Each group can be validated individually as if it were a data-class. The validation interface, `Validable`, requires the `validateFails` and `validateAllFails` method that we have already encountered in data-class validation. All data-classes also implement the `Validable` interface.

```
public interface Validable {
    boolean validateFails(FieldValidateHandler handler);
    boolean validateAllFails(FieldValidateHandler handler);
}
```

Listing 64: Validation interface, at class-data and group level

Defining a repeating group is similar to defining a group. An inner class is created to handle the repeating intermediate element, a private field with n instances of the intermediate element, a fluent getter with an index of the field, and a *Consumer* with index of the field.

Also in this case the inner class that defines the intermediate element implements the validation interface,

```
public class TabError implements Validable, IError {/* ... */}
private final TabError[] tabError = new TabError[] {
    this.new TabError(0),
    /* ... */
};
public TabError tabError(int k) { return this.tabError[k-1]; }
public void withTabError(int k, WithAction<TabError> action) { action.accept(this.tabError[k-1]); }
```

Listing 65: Implementation of a repeating group inside the data-class

and, if defined through an interface, the interface with the definition of the detail of the fields of the inner class.

List of Figures

List of Tables

List of Listings

Index