



---

# Guida Utente RECFM

*Generazione di codice per gestire file posizionali*

---

VERSIONE 0.7.0

8 AGOSTO 2023

# Introduzione

In alcune occasioni può capitare di avere a che fare con file (o aree di memoria) posizionali, vedi fig. 1, in questi casi è necessario perdere un sacco di tempo per fare una classe dedicata a ogni stringa-dati con i setter e getter per leggere e scrivere i valori<sup>1</sup>.

S	C	A	R	L	E	T					J	O	H	A	N	S	S	O	N					1	9	8	4	1	1	2	N	E	W			Y	O	R	K					U	S	A
A	N	A									D	E		A	R	M	A	S						1	9	8	8	0	4	3	0	H	A	V	A	N	A					C	U	B		

Figura 1: Esempio di file-dati posizionale

Questo gruppo di programmi si propone di minimizzare il tempo per creare queste classi. In pratica viene definita la struttura della stringa-dati con un file di configurazione, questo viene dato in pasto ad un plugin che genera la classe-dati corrispondente, che può essere utilizzata senza nessun ulteriore intervento utente.

I programmi sono strutturati usando service provider interface, vedi fig. 2, abbiamo un plugin, o un programma utente (*Service*), che vede direttamente le classi definite nella *Service Provider Interface* e recupera la implementazione usando il *ServiceLoader*, in questo modo non ha una dipendenza specifica con una delle implementazioni usate. Il *Service Provider* deve implementare le classi definite nella *Service Provider Interface*.

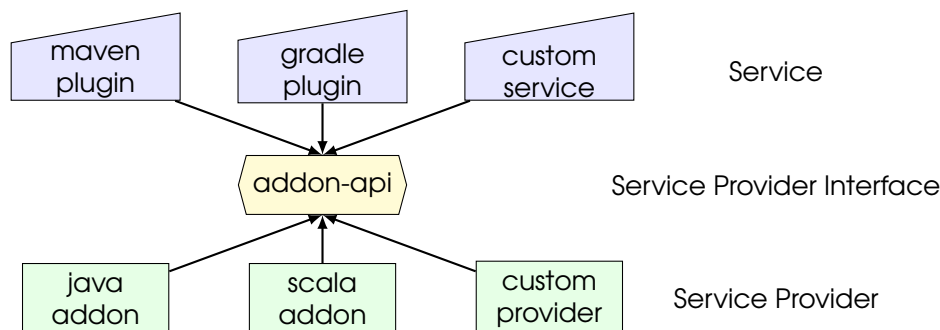


Figura 2: Struttura service, service-provider-interface, service-provider

Se il maven-plugin trova in esecuzione la libreria con l'implementazione java-addon genererà i sorgenti in java, ma se trova l'implementazione scala-addon genererà i sorgenti in scala.

La documentazione è divisa in tre parti. Nella prima, [I](#), viene data una descrizione dettagliata delle classi definite nella *service provider interface*, questa parte è utile per chi volesse sviluppare un *custom service* o un *custom provider*. Se si è interessati solo a come generare il codice partendo dai file di configurazione può essere saltata.

Nella seconda parte, [II](#), viene data una descrizione di due plugin usati generare il codice. In particolare come definire i tracciati con i file di configurazione e come attivare il plugin.

Nella terza parte, [III](#), viene data una descrizione del *service provider* che genera il sorgente java mostrando anche alcune funzionalità aggiuntive delle classi generate oltre ai semplici setter e getter.

<sup>1</sup>esiste com.ancientprogramming.fixedformat4j:fixedformat4j che fornisce alcune funzionalità base, ma in molte situazioni non è sufficientemente flessibile.

# Indice

<b>I</b>	<b>Service Provider Interface</b>	<b>4</b>
<b>1</b>	<b>Service Provider Interface</b>	<b>5</b>
1.1	CodeProvider . . . . .	5
1.2	CodeFactory . . . . .	5
1.3	Classi / Interfacce . . . . .	6
1.3.1	Argomenti globali — GenerateArgs . . . . .	6
1.3.2	Default dei campi — FieldDefault . . . . .	6
<b>2</b>	<b>Definizione di campi singoli</b>	<b>8</b>
2.1	Campo Alfanumerico . . . . .	8
2.2	Campo Numerico . . . . .	9
2.3	Campo Custom (alfanumerico) . . . . .	10
2.4	Campo Numerico nullabile . . . . .	11
2.5	Campo Dominio . . . . .	12
2.6	Campo Filler . . . . .	12
2.7	Campo Valore costante . . . . .	13
<b>3</b>	<b>Definizione di campi multipli</b>	<b>14</b>
3.1	Gruppo di campi . . . . .	14
3.2	Gruppo di campi ripetuto . . . . .	14
3.3	Campi incorporati mediante interfaccia . . . . .	15
3.4	Gruppo di campi definito mediante interfaccia . . . . .	15
3.5	Gruppo di campi ripetuto definito mediante interfaccia . . . . .	15
<b>4</b>	<b>enum utilizzati</b>	<b>17</b>
<b>II</b>	<b>Service</b>	<b>19</b>
<b>5</b>	<b>Maven plugin</b>	<b>21</b>
5.1	Struttura del file di configurazione . . . . .	22
5.2	Campi Singoli . . . . .	24
5.2.1	Campo Alfanumerico . . . . .	24
5.2.2	Campo Numerico . . . . .	25
5.2.3	Campo Custom (alfanumerico) . . . . .	25
5.2.4	Campo Numerico nullabile . . . . .	26
5.2.5	Campo Dominio . . . . .	27
5.2.6	Campo Filler . . . . .	28
5.2.7	Campo Costante . . . . .	29
5.3	Campi multipli . . . . .	30
5.3.1	Campo Gruppo . . . . .	30
5.3.2	Campo Gruppo ripetuto . . . . .	30
5.3.3	Campo gruppo incorporato da interfaccia . . . . .	31
5.3.4	Campo Gruppo da interfaccia . . . . .	31
5.3.5	Campo Gruppo ripetuto da interfaccia . . . . .	32
<b>6</b>	<b>Gradle plugin</b>	<b>34</b>

<b>III</b>	<b>Service Provider</b>	<b>35</b>
<b>7</b>	<b>Service Provider</b>	<b>36</b>
7.1	Generazione sorgente java — java-addon . . . . .	36
7.1.1	Validazione . . . . .	37
7.1.2	Setter e getter . . . . .	38
7.1.3	Campi Singoli . . . . .	38
7.1.4	Campi Multipli . . . . .	39

**Parte I**

**Service Provider Interface**

# Capitolo 1

## Service Provider Interface

L'artefatto `recfm-addon-api` mette a disposizione una serie di interfacce, alcuni enum e java-bean per permettere al modulo client di definire i tracciati. Il codice è compilato in modo da essere compatibile con il java 5, ma fornisce il `module-info` per essere utilizzabile propriamente anche con il java 9 e superiori.

### 1.1 CodeProvider

Il punto di partenza è l'interfaccia *CodeProvider*, recuperata dal *ServiceLoader*, vedi cod. 1, questa interfaccia fornisce l'istanza dell'interfaccia *CodeFactory*.

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

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

Sorgente 1: interfaccia *CodeProvider* e recupero del *CodeProvider* dal *ServiceLoader*

### 1.2 CodeFactory

L'interfaccia *CodeFactory*, vedi cod. 2, fornisce i metodi per definire tutti gli elementi della struttura.

```
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();
}
```

Sorgente 2: interfaccia *CodeFactory*

## 1.3 Classi / Interfacce

Il primo metodo dell'interfaccia *CodeFactory* fornisce la definizione per una classe, vedi cod. 3, e il secondo metodo dell'interfaccia *CodeFactory* fornisce la definizione per una interfaccia, vedi cod. 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);
}
```

Sorgente 3: interfaccia ClassModel

Entrambe le definizioni richiedono il nome della struttura, la sua lunghezza, indicano se generare o meno la documentazione automatica per la classe, l'elenco dei campi che la compongono e mettono a disposizione un metodo per generare il codice sorgente.

```
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);
}
```

Sorgente 4: interfaccia TraitModel

La definizione della classe richiede anche due parametri aggiuntivi per indicare come comportarsi se la dimensione dei dati forniti fosse superiore o inferiore a quella attesa.

Prima di vedere il dettaglio della definizione dei vari campi, vediamo il contenuto degli altri due parametri richiesti per la generazione del codice sorgente.

### 1.3.1 Argomenti globali — GenerateArgs

La classe *GenerateArgs*, vedi cod. 5, permette di definire alcuni parametri generali, comuni per tutte le classi generate. Il parametro *sourceDirectory* indica la directory sorgente radice dove generare i sorgenti, i tre parametri successivi identificano il programma (o plugin) che ha fornito la definizione del tracciato, questi parametri sono mostrati come commento all'inizio dei file generati.

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

Sorgente 5: classe GenerateArgs

### 1.3.2 Default dei campi — FieldDefault

Nella definizione delle classi e dei campi, alcuni parametri disponibili nelle relative definizioni cambiano necessariamente (il nome del campo), altri sono quasi sempre uguali per la stessa tipologia di campo (quali

sono i caratteri validi di un campo alfanumerico). Per semplificare la definizione delle classi, e relativi campi, è possibile omettere nella definizione i parametri “poco variabili”, è però necessario indicare quale valore usare per questi parametri quando vengono omessi. La classe *FieldDefault*, cod. 6, mette a disposizione alcune classi dedicate per impostare il default dei parametri “poco variabili”.

```
@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();
}
```

Sorgente 6: classe FieldDefault

Il primo default riguarda il comportamento di default della classe quando viene creata partendo da una struttura (stringa), e questa ha una dimensione diversa da quella attesa; nel caso che la lunghezza della struttura fornita sia maggiore di quella attesa è possibile lanciare una eccezione e ignorare il contenuto in eccesso, nel caso che la lunghezza della struttura fornita sia minore di quella atteso è possibile lanciare una eccezione o completare la parte mancante con i valori di default della parte mancante.

```
@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;
}
```

Sorgente 7: classe ClsDefault

Gli altri default permettono di impostare i valori di default di alcuni parametri per cinque tipologie di campi. Non avendo mostrato il dettaglio delle definizioni delle varie tipologie di campo, non è opportuno introdurre in questo punto il contenuto delle classi dei default, saranno mostrate insieme al campo corrispondente.



## Capitolo 2

# Definizione di campi singoli

Nella definizione della classe, e dell'interfaccia l'elenco dei campi è impostato come `List<FieldModel>`, ma l'interfaccia `FieldModel` è una scatola vuota, serve solo per collegare a essa tutte le definizioni dei campi. In generale tutti i campi hanno una posizione iniziale (offset) e una dimensione (length); molti campi sono referenziabili tramite un nome, ma non tutti hanno necessariamente il nome; quando i campi hanno un nome possono essere primari o sovra-definire (override) campi primari, in fase di inizializzazione dei campi di una classe vengono considerati solo le definizioni dei campi primari.

Nelle definizioni dei campi la posizione iniziale del campo (offset) è impostata come `Integer`, cioè in generale non è obbligatorio impostarla, può essere calcolata automaticamente dal *Service Provider*.

### 2.1 Campo Alfanumerico

Un campo alfanumerico ha i 4 parametri base: offset, length, name e override.

I parametri `onOverflow` e `onUnderflow` indica come deve comportarsi il setter quando viene fornito un valore con dimensione maggiore o minore di quella prevista per quel campo. Il parametro `onOverflow` può assumere i valori `Error` e `Trunc`, nel primo caso è atteso che il codice generi una eccezione, nel secondo caso il valore viene trocato (a destra) ignorando i caratteri in eccesso rispetto alla dimensione attesa. Il parametro `onUnderflow` può assumere i valori `Error` e `Pad`, nel primo caso è atteso che il codice generi una eccezione, nel secondo caso vengono aggiunti degli spazi (a destra) per raggiungere la dimensione attesa.

Il parametro `check` permette di indicare dei controlli per restringere l'insieme di caratteri permessi per il valore. Questo controllo si attiva in fase di validazione della strina-dati, chiamando i setter e i getter. I possibili valori sono `None`, `Ascii`, `Latin1`, `Valid`, nel primo caso non viene fatto nessun controllo, nel secondo caso vengono accettati solo i caratteri ASCII, nel terzo caso vengono accettati solo i caratteri ISO-8859-1<sup>1</sup>, e nell'ultimo vengono accettati i caratteri UTF-8 validi.

```
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);
}
```

Sorgente 8: interfaccia `AbcModel` (campo alfanumerico)

Il parametro `normalize` permette di indicare come normalizzare il valore del campo in fase di getter. Il parametro `normalize` può assumere 3 valori `None`, `Trim` e `Trim1`. Il primo valore indica di non eseguire nessuna

<sup>1</sup>più precisamente i caratteri unicode da `\u0020` a `\u007e` e da `\u00a0` a `\u00ff`

modifica del dato, il secondo valore indica di rimuovere tutti gli spazi a destra fino a trovare un carattere diverso da spazio, nel caso che il valore sia composto solo da spazi viene prodotta una stringa vuota, l'ultimo valore, analogamente al valore precedente fa rimuovere gli spazi a destra fino a trovare un carattere diverso da spazio, ma nel caso che il valore sia composto solo da spazi restituisce una stringa composta da uno spazio.

Il parametro `checkGetter` indica se attivare o meno il controllo indicato col parametro `check` quando viene chiamato il getter; se la stringa-dati viene validata preventivamente, questo controllo può essere disattivato. Il parametro `checkSetter` indica se attivare o meno il controllo indicato col parametro `check` quando viene chiamato il 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;
}
```

Sorgente 9: class `AbcDefault` (default campo alfanumerico)

La classe `AbcDefault`, vedi cod. 9, imposta i valori di default per i parametri `onOverflow`, `onUnderflow`, `check`, `normalize`, `checkGetter`, `checkSetter`, nel caso non siano impostati dal client.

## 2.2 Campo Numerico

Anche un campo numerico ha i 4 parametri base: `offset`, `length`, `name` e `override`.

Il parametro `access` indica come generare i setter e getter. Nella stringa-dati il campo numerico ha una rappresentazione in formato stringa, nel codice generato è possibile scegliere se i setter e getter gestiscano il valore come stringa (con caratteri numerici) o convertire il frammento di stringa-dati, corrispondente al campo, in una rappresentazione numerica nativa (`byte`, `short`, `int`, `long`) o gestirli entrambi. Il parametro `access` può assumere i valori `String`, `Number` e `Both`. Nel primo caso vengono generati setter e getter che gestiscono il valore come stringa (numerica), nel secondo caso come numerico nativo, e nell'ultimo caso vengono generati entrambi (andrà indicato dal provider come distinguere il getter stringa da quello numerico). Nel caso venga usato un accesso che prevede setter/getter di tipo stringa, viene controllato in fase di setter che la stringa fornita sia numerica, e in fase di getter che la stringa restituita sia numerica.

```
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);
}
```

Sorgente 10: interfaccia `NumModel` (campo numerico)

Il parametro `wordWidth` ha senso solo se si è scelta una modalità di accesso che genera i setter/getter numerici, sostanzialmente indica la dimensione minima da usare nelle rappresentazioni numeriche. Il parametro `wordWidth` può assumere i valori `Byte`, `Short`, `Int` e `Long`, i valori corrispondono all'utilizzo dei tipi nativi corrispondenti. Per fare un esempio, se un campo numerico è rappresentato da una stringa di 4 caratteri, può essere convertito in formato numerico in formato `short`, se il parametro `wordWidth` è impostato a `Int` vengono generati setter/getter di tipo `int`; se il valore del parametro fosse stato `Byte` o `Short` sarebbero stati generati setter/getter di tipo `short`.

I parametri `onOverflow` e `onUnderflow` indica come deve comportarsi il setter quando viene fornito un valore con dimensione maggiore o minore di quella prevista per quel campo. Il parametro `onOverflow` può assumere i valori `Error` e `Trunc`, nel primo caso è atteso che il codice generi una eccezione, nel secondo caso il valore viene troncato (a sinistra) ignorando le cifre in eccesso rispetto alla dimensione attesa. Il parametro `onUnderflow` può assumere i valori `Error` e `Pad`, nel primo caso è atteso che il codice generi una eccezione, nel secondo caso vengono aggiunti degli zero (a sinistra) per raggiungere la dimensione attesa.

Il campo `normalize` ha senso solo se si è scelta una modalità di accesso che genera i setter/getter stringa, permette di indicare come normalizzare il valore del campo in fase di getter. Il parametro `normalize` può assumere 2 valori `None` e `Trim`. Il primo valore indica di non eseguire nessuna modifica del dato, l'altro indica di rimuovere tutti gli zero a sinistra fino a trovare una cifra diversa da zero, nel caso che il valore sia composto solo da zeri viene prodotta una stringa composta da uno zero.

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

Sorgente 11: class NumDefault (default campo numerico)

La classe *NumDefault*, vedi cod. 11, imposta i valori di default per i parametri `access`, `wordWidth`, `onOverflow`, `onUnderflow` e `normalize` nel caso non siano impostati dal client.

## 2.3 Campo Custom (alfanumerico)

Anche un campo custom ha i 4 parametri base: `offset`, `length`, `name` e `override`.

Un campo custom è una generalizzazione di un campo alfanumerico, e può essere configurato per emulare un campo numerico o numerico nullabile. Il primo parametro sensibile da considerare è `align`, il parametro indica come deve essere allineato il campo `onUnderflow = Pad`. Il parametro può assumere 2 valori `LFT` e `RGT`, il primo valore indica che il campo deve essere allineato a sinistra, il secondo valore che deve essere allineato a destra. Il valore di questo parametro non impatta solo sul parametro `onUnderflow` (indicando da quale direzione devono essere aggiunti i caratteri di padding), ma anche su `onOverflow` (indicando da quale direzione devono essere tolti i caratteri in eccesso) e `normalize` (indicando da quale direzione devono essere rimossi i caratteri di padding).

Il parametro `padChar` indica il carattere di riempimento da aggiungere (in caso di `onUnderflow = Pad`) o togliere (in caso di `normalize = Trim`). Il parametro `initChar` indica il carattere da usare per inizializzare il campo.

Per il parametro `check` valgono le stesse considerazioni del corrispondente parametro nel [caso alfanumerico](#). In questo caso i possibili valori sono `None`, `Ascii`, `Latin1`, `Valid`, `Digit` e `DigitOrBlank`. I primi quattro valori sono identici al caso alfanumerico, il valore `Digit` limita i caratteri accettati a quelli numerici (da 0 a 9), come per un campo numerico; il valore `DigitOrBlank` richiede che i caratteri siano numerici o tutti spazi, come per un campo numerico nullabile.

Il parametro `regex` può essere valorizzato con una espressione regolare che deve essere soddisfatta da valore del campo. Se questo parametro è presente, viene ignorato il parametro `check`.

Per i parametri `onOverflow`, `onUnderflow`, `normalize`, `checkGetter`, `checkSetter` valgono le stesse considerazioni dei corrispondenti campi nel caso alfanumerico. Attenzione perché l'azione dei parametri `onOverflow`, `onUnderflow` e `normalize` dipende anche dal valore dei parametri `align` e `initChar`.

La classe *CusDefault*, vedi cod. 13, imposta i valori di default per i parametri `align`, `padChar`, `initChar`, `check`, `onOverflow`, `onUnderflow`, `normalize`, `checkGetter` e `checkSetter` nel caso non siano impostati dal client.

```

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);
}

```

Sorgente 12: interfaccia CusModel (campo custom)

```

@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;
}

```

Sorgente 13: class CusDefault (default campo custom)

## 2.4 Campo Numerico nullabile

Un campo numerico nullabile è una estensione di un campo numerico ordinario. La differenza è che nella stringa-dati è permesso il valore spazio (tutti i caratteri a spazio), a questo valore corrisponde il valore null nel campo della classe-dati.

```

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);
}

```

Sorgente 14: interfaccia NuxModel (campo numerico nullabile)

Come si vede dalla definizione, cod. 14 ci sono gli stessi parametri di un campo numerico (cod. 10), con gli stessi significati, più uno: il parametro initialize. Questo parametro indica come inizializzare il campo quando viene creata la classe con il costruttore vuoto, a spazio (cioè a null), o a zero.

```

@Data
public class NuxDefault {
    private AccessMode access = AccessMode.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;
}

```

Sorgente 15: class NuxDefault (default campo numerico nullabile)

Per gestire il default a livello generale per questo tipo di campi viene usata una classe default dedicata, che è la copia della corrispondente per il caso numerico ordinario, con in più il default di inizializzazione.

## 2.5 Campo Dominio

Un campo di tipo dominio è un campo alfanumerico che può assumere solo dei valori costanti predefiniti. Il campo di tipo dominio, cod. 16, ha i 4 parametri base: offset, length, name e override, e in più il parametro items che dovrà fornire l'elenco dei valori costanti ammessi.

```

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);
}

```

Sorgente 16: interfaccia DomModel (campo dominio)

Per questo tipo di campo non ha senso nessun default globale come visto per i campi alfanumerici e numerici. Il valori permessi sono quelli forniti nel parametro items, qualunque altro valore causerà una eccezione.

## 2.6 Campo Filler

Un campo di tipo *Filler*, non è un campo vero e proprio, non ha un nome associato, non genera nessun setter o getter nella classe-dati, e nessun metodo di controllo. È un modo per indicare che nella stringa-dati è presente un'area a cui non è associato nessun valore, o non siamo interessati a quella area della stringa-dati.

```

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

    void setFill(Character fill);
}

```

Sorgente 17: interfaccia FilModel (campo filler)

Il campo filler ha soltanto i 2 parametri base: offset e length. Ha poi il parametro fill per indicare con quale carattere valorizzare il campo quando la classe-dati viene creata con il costruttore vuoto.

Il campo filler prevede una classe default dedicata per indicare il carattere di riempimento a livello generale.

```

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

```

Sorgente 18: class FilDefault (default campo filler)

## 2.7 Campo Valore costante

Un campo di tipo *Costante* è simile al campo filler. Come per il campo filler non vengono generati setter e getter nella classe-dati, ma nella validazione della stringa-dati viene verificato che l'area corrispondente al campo abbia il valore costante indicato.

```

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

    void setValue(String value);
}

```

Sorgente 19: interfaccia ValModel (campo costante)

Anche il campo costante ha soltanto i 2 parametri base: offset e length. Ha poi il parametro value per indicare il valore costante atteso. Il valore fornito viene usato per *valorizzare* il campo quando la classe-dati viene istanziata con il costruttore vuoto.

## Capitolo 3

# Definizione di campi multipli

In alcuni casi può essere utile raggruppare alcuni campi all'interno di un elemento contenitore. Questo permette di avere all'interno della definizione della struttura della stringa-dati due campi con lo stesso nome all'interno di elementi contenitori diversi.

### 3.1 Gruppo di campi

Un campo di tipo *Gruppo* non è un campo vero e proprio, è in realtà un contenitore di altri campi.

```
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);  
}
```

Sorgente 20: interfaccia GrpModel (campo gruppo)

Un campo gruppo ha i 4 parametri base: offset, length, name e override. Oltre a questo prevede il parametro fields per fornire l'elenco dei campi figli. L'area di stringa-data selezionata da offset e length dovrà essere completamente definita dai campi figli. È possibile per un campo figlio, usare lo stesso nome di un campo definito allo stesso livello del campo padre.

### 3.2 Gruppo di campi ripetuto

Un campo di tipo *Gruppo Ripetuto* è simile al campo gruppo, con la differenza che sono presenti  $n$  occorrenze del gruppo. Oltre ai parametri usati dal campo gruppo è presente il parametro times, che indica il numero di volte che il gruppo è ripetuto.

```
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);  
}
```

Sorgente 21: interfaccia OccModel (campo gruppo ripetuto)

In questo caso l'area della stringa-dati definita da `offset` e `length` è quella della prima occorrenza del gruppo. La dimensione realmente usata è `length × times`.

### 3.3 Campi incorporati mediante interfaccia

Una interfaccia (cod. 4) può essere definita allo stesso modo di una classe (cod. 3). Il campo *Incorporato* non è un campo vero e proprio, non ha un nome. Permette di indicare che una certa area della stringa-dati deve essere interpretata con i campi definiti dalla interfaccia indicata.

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

    void setSource(TraitModel source);
}
```

Sorgente 22: interfaccia EmbModel (campo incorporato)

Il campo incorporato ha soltanto i 2 parametri base: `offset` e `length`. Ha poi il parametro `source` per indicare quale interfaccia utilizzare per definire i campi. L'interfaccia dovrà avere la stessa dimensione (`length`) del campo incorporato, la posizione iniziale della interfaccia verrà traslata per adattarla al valore indicato dal campo incorporato. La classe generata, ovviamente, implementerà l'interfaccia indicata.

### 3.4 Gruppo di campi definito mediante interfaccia

Il campo *Gruppo/Interfaccia* è simile al campo incorporato. In questo caso i campi definiti dall'interfaccia non vengono incorporati al livello corrente, ma viene definito un gruppo che li contiene.

```
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);
}
```

Sorgente 23: interfaccia GrpTraitModel (campo gruppo/interfaccia)

Un campo gruppo/interfaccia ha i 4 parametri base: `offset`, `length`, `name` e `override`. Ha poi il parametro `typedef` per indicare quale interfaccia utilizzare per definire i campi del gruppo. L'interfaccia dovrà avere la stessa dimensione (`length`) del campo gruppo/interfaccia, la posizione iniziale della interfaccia verrà traslata per adattarla al valore indicato dal campo gruppo/interfaccia. In questo caso è il gruppo che implementa l'interfaccia indicata.

### 3.5 Gruppo di campi ripetuto definito mediante interfaccia

Il campo *Gruppo/Interfaccia Ripetuto* è simile al campo gruppo/interfaccia, con la differenza che sono presenti *n* occorrenze del gruppo. Oltre ai parametri usati dal campo gruppo/interfaccia è presente il parametro `times`, che indica il numero di volte che il gruppo è ripetuto.



```

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);
}

```

Sorgente 24: interfaccia OccTraitModel (campo gruppo/interfaccia ripetuto)

Come per il caso del gruppo ripetuto l'area della stringa-dati definita da offset e length è quella della prima occorrenza del gruppo. La dimensione realmente usata è  $\text{length} \times \text{times}$ .

# Capitolo 4

## enum utilizzati

Molte delle classi o interfacce di configurazione hanno dei campi con valori limitati ad alcuni valori espresse mediante enum. Vediamoli uno per uno.

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

Sorgente 25: enum LoadOverflowAction

L'enum `LoadOverflowAction` è usato dalla classe `ClassModel` per indicare come comportarsi quando la classe viene deserializzata e la stringa-dati ha una dimensione maggiore di quella attesa dalla classe-dati.

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

Sorgente 26: enum LoadUnderflowAction

L'enum `LoadUnderflowAction` è usato dalla classe `ClassModel` per indicare come comportarsi quando la classe viene deserializzata e la stringa-dati ha una dimensione inferiore a quella attesa dalla classe-dati.

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

Sorgente 27: enum CheckAbc

L'enum `CheckAbc` è usato dalla classe `AbcModel` per indicare quali caratteri sono considerati validi.

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

Sorgente 28: enum OverflowAction

L'enum `OverflowAction` è usato dalla classi per gestire i campi numerici e alfanumerici (`AbcModel`, `NumModel`, `CusModel`, `NuxModel`), per indicare come comportarsi quando il setter propone un valore che ha una dimensione maggiore di quella attesa per quel campo.

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

Sorgente 29: enum UnderflowAction

L'enum [UnderflowAction](#) è usato dalla classi per gestire i campi numerici e alfanumerici ([AbcModel](#), [NumModel](#), [CusModel](#), [NuxModel](#)), per indicare come comportarsi quando il setter propone un valore che ha una dimensione inferiore a quella attesa per quel campo.

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

Sorgente 30: enum NormalizeAbcMode

L'enum [NormalizeAbcMode](#) è usato dalla classi per gestire i campi alfanumerici ([AbcModel](#), [CusModel](#)), per indicare come normalizzare il valore restituito dal getter.

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

Sorgente 31: enum NormalizeNumMode

L'enum [NormalizeNumMode](#) è usato dalla classi per gestire i campi numerici ([NumModel](#), [NuxModel](#)), per indicare come normalizzare il valore restituito dal getter.

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

Sorgente 32: enum WordWidth

L'enum [WordWidth](#) è usato dalla classi per gestire i campi numerici ([NumModel](#), [NuxModel](#)), per indicare il tipo dato numerico primitivo di dimensione minima da usare quando vengono creati setter e getter numerici.

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

Sorgente 33: enum AccesMode

L'enum [AccesMode](#) è usato dalla classi per gestire i campi numerici ([NumModel](#), [NuxModel](#)), per indicare se creare setter e getter alfanumerici, numerici o entrambi.

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

Sorgente 34: enum InitializeNuxMode

L'enum [InitializeNuxMode](#) è usato dalla classe [NuxModel](#), per indicare come inizializzare il campo quando la classe-dati viene creata col costruttore senza argomenti.

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

Sorgente 35: enum CheckCus

L'enum [CheckCus](#) è usato dalla classe [CusModel](#) per indicare quali caratteri sono considerati validi.

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

Sorgente 36: enum AlignMode

L'enum [CheckCus](#) è usato dalla classe [CusModel](#) per indicare come allineare il campo.

# **Parte II**

# **Service**

# Service

La *Service Provider Interface* fissa semplicemente la struttura generale, ma contiene solo interfacce e java-bean. Il *Service* è l'applicazione che permette all'utente di fornire l'input previsto dalla *Service Provider Interface*. Per produrre l'output, il *Service* utilizzerà il *ServiceLoader* per cercare nel classpath un *Service Provider* che implementi la *Service Provider Interface*, e sarà il *Service Provider* a generare l'output dall'input.

Il cuore del *Service Provider Interface* recfm-addon-api è la interfaccia *CodeProvider*. L'implementazione della interfaccia viene cercata col meccanismo del *ServiceLoader*, cod.37.

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

Sorgente 37: recupero del *CodeProvider*

Una volta recuperata una istanza di *CodeFactory* è possibile creare le definizioni delle stringhe-dati e generare i sorgenti delle classi-dati.

Sono stati sviluppati due *client*, uno sotto forma di maven plugin recfm-maven-plugin, e l'altro sotto forma di gradle plugin recfm-gradle-plugin. Il codice in gran parte è identico, cambia solo il meccanismo di innesco.

## Capitolo 5

# Maven plugin

Il maven plugin `recfm-maven-plugin` permette di creare più classi e interfacce usando uno più file di configurazione in formato `yaml`. Le librerie esterne utilizzate richiedono il java 8, quindi per eseguire questo plugin è necessario almeno il java 8.

Il plugin si aspetta come parametri di configurazione

```
@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;
```

Sorgente 38: parametri impostabili del maven plugin

- Il campo `generateDirectory` indica la directory root da utilizzare per la generazione dei sorgenti, viene usato per valorizzare il campo `sourceDirectory` della classe `GenerateArgs`, come si vede dalla definizione, se il parametro è omesso viene utilizzata la directory `target/generated-sources/recfm`, normalmente può essere lasciato il valore di default. Gli altri tre parametri di `GenerateArgs` sono un identificativo del programma `service` e vengono valorizzati automaticamente.
- Il campo `settingsDirectory` indica la directory che contiene i file di configurazione, se il parametro è omesso viene usato il valore `src/main/resources`, normalmente può essere lasciato il valore di default.
- Il campo `settings` indica l'elenco dei file di configurazione da utilizzare per generare le classi/interfacce; il parametro deve essere fornito al plugin.
- Il campo `addCompileSourceRoot` è un campo tecnico, indica a maven che la directory dove sono stati generati i sorgenti deve essere inclusa tra quelle utilizzate per la compilazione principale, se il parametro è omesso viene utilizzato il valore `true`; il valore `true` è opportuno quando viene usata una directory di generazione del codice diversa da `src/main/java`, altrimenti è necessario usare plugin aggiuntivi per aggiungere il nuovo path a quello di compilazione di maven.

- Il campo `addTestCompileSourceRoot` è analogo al campo precedente, ma aggiunge la directory di generazione al path di compilazione dei test, se omesso viene utilizzato il valore `false`, tranne in casi particolari può essere lasciato il valore di default.
- Il campo `codeProviderClassName` indica quale è la classe concreta che implementa la *Service Interface*, se omesso viene utilizzata la “prima” implementazione recuperata del *ServiceLoader*; se in classpath è presente una sola implementazione, non è necessario valorizzare il parametro.

```
<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>
          <setting>bar.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>
```

Sorgente 39: esempio minimale di esecuzione del plugin

Un esempio di esecuzione del plugin è mostrato nel cod. 39, il plugin per essere eseguito deve avere come dipendenza una libreria che fornisca l’implementazione dell’interfaccia, altrimenti il *ServiceLoader* non trova nulla ed il plugin termina in errore.

Tutti gli altri parametri sono forniti nei file di configurazione.

## 5.1 Struttura del file di configurazione

Per gestire la configurazione dei tracciati il plugin definisce la classe *MasterBook*, vedi cod. 40, è divisa in due componenti, la prima `defaults` è semplicemente il java-bean *FieldDefault* (vedi 6) messo a disposizione dalla *Service Provider Interface* per fornire i valori di default dei parametri “poco variabili” delle classi e dei campi.

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

Sorgente 40: classe di configurazione MasterBook

Per semplificare la valorizzazione del file di configurazione `yaml`, viene usata una funzionalità delle librerie `yaml`, che permette di definire nomi abbreviati o alternativi dei parametri e dei valori dei campi di tipo `enum`. I dettagli della componente del campo `defaults` sarà mostrato insieme al campo a cui fornisce il default del valore dei parametri.

```

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

```

Sorgente 41: classe di configurazione ClassPackage

La seconda componente di *MasterBook*, packages, è una lista di *ClassPackage* (41), cioè di package all'interno dei quali vengono definiti un elenco di interfacce e classi. Espandendo un esempio di questo oggetto in formato yaml (con il default relativo) abbiamo:

```

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:
          - ...

```

Sorgente 42: configurazione, area packages / interfaces / classes

Nei commenti vengono mostrati gli eventuali nomi alternativi dei campi e l'elenco dei valori *enum* permessi. Se non vengono usate interfacce, il nodo interfaces può essere omesso. Sia per le classi che le interfacce il nome e la lunghezza del tracciato da associare devono essere impostate dall'utente, nella definizione della classe può anche essere impostato il comportamento nel caso che venga fornita in fase di de-serializzazione una struttura con una dimensione maggiore o minore di quella attesa.

interfaces <a href="#">TraitModel</a>				
	attributo	alt	tipo	O
	name		String	✓
	length	len	int	✓
<a href="#">ClsDefault</a>	doc		boolean	
	fields		array	✓

Tabella 5.1: Attributi impostabili per la definizione di una interfaccia

Anche se tutti i tipi di campo hanno una posizione di inizio e una lunghezza, il dettaglio dei parametri di configurazione varia da campo a campo ed è necessario introdurre i parametri di configurazione campo per campo.

Per indicare esplicitamente il tipo di campo utilizzato vengono introdotti dei *tag* da associare ad ogni campo, nella tabella 5.3 sono mostrati i *tag* associati a ogni tipo di campo.

Riguardo all'offset dei campi, va segnalato, che alcune caratteristiche non dipendono dal *Service*, ma dal *Service Provider*: l'offset minimo può essere zero o uno, l'impostazione dell'offset può es-



classes <a href="#">ClassModel</a>				
attributo	alt	tipo	O	default
name		String	✓	
length	len	int	✓	
onOverflow	ovf	enum		<code>\${defaults.cls.onOverflow:Trunc}</code>
onUnderflow	unf	enum		<code>\${defaults.cls.onUnderflow:Pad}</code>
doc		boolean		<code>\${defaults.cls.doc:true}</code>
fields		array	✓	

Tabella 5.2: Attributi impostabili per la definizione di una classe

Tag definizione campo		
tag	classe	note
<a href="#">!Abc</a>	<a href="#">AbcModel</a>	campo alfanumerico
<a href="#">!Num</a>	<a href="#">NumModel</a>	campo numerico
<a href="#">!Cus</a>	<a href="#">CusModel</a>	campo custom
<a href="#">!Nux</a>	<a href="#">NuxModel</a>	campo numerico nullabile
<a href="#">!Dom</a>	<a href="#">DomModel</a>	campo dominio
<a href="#">!Fil</a>	<a href="#">FilModel</a>	campo filler
<a href="#">!Val</a>	<a href="#">ValModel</a>	campo costante
<a href="#">!Grp</a>	<a href="#">GrpModel</a>	gruppo di campi
<a href="#">!Occ</a>	<a href="#">OccModel</a>	gruppo di campi ripetuti
<a href="#">!Emb</a>	<a href="#">EmbModel</a>	campi inclusi da interfaccia
<a href="#">!GRP</a>	<a href="#">GrpTraitModel</a>	gruppo di campi inclusi da interfaccia
<a href="#">!OCC</a>	<a href="#">OccTraitModel</a>	gruppo di campi ripetuti inclusi da interfaccia

Tabella 5.3: Tag yaml per la identificazione del campo

sere obbligatoria, o facoltativa (l'offset può essere calcolato automaticamente usando l'offset e la lunghezza del campo precedente), o non permessa.

Il *Service Provider* descritto nella sezione 7.1 utilizza un offset minimo 1 e l'impostazione dell'offset è facoltativa. Se si omette l'offset in un campo definito con *override*, si assume che il campo ridefinisce il campo che lo precede nella definizione della struttura. Nella definizione delle interfacce l'uso dell'offset è opzionale, ma a differenza delle classi, che richiedono un offset minimo 1, per le interfacce può essere usato qualunque valore iniziale, l'offset effettivo viene corretto quando l'interfaccia viene applicata alla classe.

## 5.2 Campi Singoli

### 5.2.1 Campo Alfanumerico

La definizione yaml del campo alfanumerico riflette la struttura imposta dalla service provider interface, vedi 8. Un campo alfanumerico è specificato indicando il tag [!Abc](#), un esempio di definizione di campi alfanumerici è mostrato nel cod. 43, nell'esempio è mostrato anche il nodo del default globale per i campi alfanumerici, i valori impostati sono quelli di default della *service provider interface*, quindi non è necessario impostare esplicitamente i parametri se si vuole impostare questi valori.

Nell'esempio, il nodo di default dei campi alfanumerici, è impostato usando i nomi canonici dei parametri. Il *plugin* usa una funzionalità disponibile della libreria per leggere il file yaml, e definisce dei nomi abbreviati dei parametri, che possono essere utilizzati come alternativa ai nomi canonici.

Nella tabella 5.4 sono mostrati tutti gli attributi previsti per un campo alfanumerico, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

```

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 }

```

Sorgente 43: esempio definizione campi alfanumerici

!Abc: <a href="#">AbcModel</a>				
	attributo	alt	tipo	O
	offset	at	int	✓
	length	len	int	✓
	name		String	✓
	override	ovr	boolean	
<a href="#">AbcDefault</a>	onOverflow	ovf	<a href="#">enum</a>	
	onUnderflow	unf	<a href="#">enum</a>	
	check	chk	<a href="#">enum</a>	
	normalize	nrm	<a href="#">enum</a>	
	checkGetter	get	boolean	
	checkSetter	set	boolean	
	default			
	auto-calcolato			
	false			
	<code>\${defaults.abc.onOverflow:Trunc}</code>			
	<code>\${defaults.abc.onUnderflow:Pad}</code>			
	<code>\${defaults.abc.check:Ascii}</code>			
	<code>\${defaults.abc.normalize:None}</code>			
	<code>\${defaults.abc.checkGetter:true}</code>			
	<code>\${defaults.abc.checkSetter:true}</code>			

Tabella 5.4: Attributi impostabili per un campo alfanumerico

## 5.2.2 Campo Numerico

La definizione yaml del campo numerico riflette la struttura imposta dalla service interface, vedi [10](#) Un campo numerico è specificato indicando il tag `!Num`, un esempio di definizione di campi numerici è mostrato nel cod. [44](#), nell'esempio è mostrato anche il nodo del default globale per i campi numerici, i valori impostati sono quelli di default della *service provider interface*, quindi non è necessario impostare esplicitamente i parametri se si vuole impostare questi valori.

Nella tabella [5.5](#) sono mostrati tutti gli attributi previsti per un campo numerico, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default. Anche se i parametri access e wordWidth sono stati introdotti nella [§ 2.2](#), ricordo che un campo "numerico" può essere gestito come una stringa (dove sono ammessi solo caratteri numerici), o convertito in un formato numerico nativo, o entrambi. Il parametro access indica se creare soltanto i setter/getter stringa, creare soltanto i setter/getter numerici o entrambi. Nel caso che venga utilizzata una rappresentazione numerica nativa, il parametro wordWidth indica la rappresentazione nativa di dimensione minima da usare. In generale il *Service Provider* selezionerà la dimensione della rappresentazione nativa in base alla dimensione della stringa-dati che finirà con rappresentare il valore del campo.

## 5.2.3 Campo Custom (alfanumerico)

La definizione yaml del campo custom riflette la struttura imposta dalla service provider interface, vedi [12](#). Un campo alfanumerico è specificato indicando il tag `!Cus`, un esempio di definizione di campi custom è mostrato

```

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 }

```

Sorgente 44: esempio definizione campi numerici

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

Tabella 5.5: Attributi impostabili per un campo numerico

nel cod. 45, nell'esempio è mostrato anche il nodo del default globale per i campi custom, i valori impostati sono quelli di default della *service provider interface*, quindi non è necessario impostare esplicitamente i parametri se si vuole impostare questi valori.

Un campo custom è una estensione di un campo alfanumerico. Un campo alfanumerico è necessariamente allineato a sinistra, troncato, trim-ato a destra, pad-dato a destra con spazi, inizializzato a spazi. In un campo custom è possibile scegliere l'allineamento del campo, il carattere di pad-ding e di inizializzazione; ha un check esteso rispetto a quello alfanumerico, infine, l'attributo regex può essere usato per validare i valori ammessi per il campo (al posto di quello definito con check).

Nella tabella 5.6 sono mostrati tutti gli attributi previsti per un campo custom, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

## 5.2.4 Campo Numerico nullabile

La definizione yaml del campo numerico nullabile riflette la struttura imposta dalla service interface, vedi 14. Un campo numerico nullabile è specificato indicando il tag **!Nux**, un esempio di definizione di campi numerici nullabili è mostrato nel cod. 46, nell'esempio è mostrato anche il nodo del default globale per i campi numerici nullabili, i valori impostati sono quelli di default della *service provider interface*, quindi non è necessario impostare esplicitamente i parametri se si vuole impostare questi valori.

Un campo numerico nullabile è una estensione di un campo numerico ordinario, la differenza è che nella rappresentazione stringa-dati può assumere il valore spazio (tutti spazi), che corrisponde nella classe dati al valore null. Conseguentemente nella definizione del campo è presente un parametro aggiuntivo per indicare se il campo deve essere inizializzato a null o a zero quando la classe-dati viene creata col costruttore vuoto.

```

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 }

```

Sorgente 45: esempio definizione campi custom

!Cus: <a href="#">CusModel</a>				
	attributo	alt	tipo	O
	offset	at	int	✓
	length	len	int	✓
	name		String	✓
	override	ovr	boolean	
CusDefault	onOverflow	ovf	enum	
	onUnderflow	unf	enum	
	padChar	pad	char	
	initChar	ini	char	
	check	chk	enum	
	align		enum	
	normalize	nrm	enum	
	checkGetter	get	boolean	
	checkSetter	set	boolean	
	regex		String	
				default
				auto-calcolato
				false
				\${defaults.cus.onOverflow:Trunc}
				\${defaults.cus.onUnderflow:Pad}
				\${defaults.cus.pad:' '}
				\${defaults.cus.ini:' '}
				\${defaults.cus.check:Ascii}
				\${defaults.cus.align:LFT}
				\${defaults.cus.normalize:None}
				\${defaults.cus.checkGetter:true}
				\${defaults.cus.checkSetter:true}
				null

Tabella 5.6: Attributi impostabili per un campo custom

Nella tabella 5.7 sono mostrati tutti gli attributi previsti per un campo numerico, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

## 5.2.5 Campo Dominio

La definizione yaml del campo dominio riflette la struttura imposta dalla service interface, vedi 16. Un campo dominio è specificato indicando il tag **!Dom**, un esempio di definizione di campi dominio è mostrato nel cod. 47, questo tipo di campo non ha nessun default globale.

Un campo dominio è sostanzialmente un campo alfanumerico, che può assumere solo un limitato numero di valori.

Nella tabella 5.8 sono mostrati tutti gli attributi previsti per un campo dominio, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default. Quando la classe-dati è creata col costruttore vuoto il campo viene inizializzato con il primo valore tra quelli forniti della lista dei possibili valori.

```

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 }

```

Sorgente 46: esempio definizione campi numerici nullabili

!Nux: <a href="#">NuxModel</a>				
	attributo	alt	tipo	O
	offset	at	int	✓
	length	len	int	✓
	name		String	✓
	override	ovr	boolean	
NuxDefault	onOverflow	ovf	enum	
	onUnderlow	unf	enum	
	access	acc	enum	
	wordWidth	wid	enum	
	normalize	nrm	enum	
	initialize	ini	enum	
				default
				auto-calcolato
				false
				\${defaults.nux.onOverflow:Trunc}
				\${defaults.nux.onUnderflow:Pad}
				\${defaults.nux.access:String}
				\${defaults.nux.wordWidth:Int}
				\${defaults.nux.normalize:None}
				\${defaults.nux.initialize:Space}

Tabella 5.7: Attributi impostabili per un campo numerico nullabile

```

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 ] }

```

Sorgente 47: esempio definizione campi dominio

## 5.2.6 Campo Filler

La definizione yaml del campo filler riflette la struttura imposta dalla service interface, vedi 17. Un campo filler è specificato indicando il tag `!Fil`, un esempio di definizione di campi filler è mostrato nel cod. 48, nell'esempio è mostrato anche il nodo del default globale per i campi filler, il valore impostato è quello di default della *service provider interface*, quindi non è necessario impostare esplicitamente il parametro se si vuole impostare questo valore.

!Dom: DomModel				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
name		String	✓	
override	ovr	boolean		false
items		array	✓	

Tabella 5.8: Attributi impostabili per un campo dominio

```
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 }
```

Sorgente 48: esempio definizione campi filler

Un campo filler non è un campo vero e proprio, non vengono generati i setter/getter, non viene fatto nessun controllo sul valore della stringa-dati corrispondente. Indica semplicemente un'area della stringa-dati a cui non siamo interessati, ma che deve essere presente nella definizione della struttura per non lasciare aree non definite.

!Fil: FilModel				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
FilDefault	fill		char	\${defaults.fil.fill:0}

Tabella 5.9: Attributi impostabili per un campo filler

Nella tabella 5.9 sono mostrati tutti gli attributi previsti per un campo filler, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

## 5.2.7 Campo Costante

La definizione yaml del campo costante riflette la struttura imposta dalla service interface, vedi 19. Un campo costante è specificato indicando il tag **!Val**, un esempio di definizione di campi costanti è mostrato nel cod. 49, questo tipo di campo non prevede default globali.

Un campo costante può essere visto come una variante di un campo filler, o come un campo dominio con un solo valore. Per questo tipo di campo non vengono generati i setter/getter, ma il campo viene controllato per verificare che la stringa-dati corrispondente al campo abbia il valore atteso.

Nella tabella 5.10 sono mostrati tutti gli attributi previsti per un campo costante, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

```

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 }

```

Sorgente 49: esempio definizione campi costanti

!Val: ValModel				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
value	val	string	✓	

Tabella 5.10: Attributi impostabili per un campo costante

## 5.3 Campi multipli

In alcuni casi è utile raggruppare alcuni campi all'interno di un elemento contenitore di contesto. In questo modo è possibile usare lo stesso nome campo in contesti diversi. Un campo multiplo non ha default globali.

### 5.3.1 Campo Gruppo

La definizione yaml del campo gruppo riflette la struttura imposta dalla service interface, vedi 20. Un campo gruppo è specificato indicando il tag **!Grp**, un esempio di definizione di campi gruppo è mostrato nel cod. 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 }
    ] }

```

Sorgente 50: esempio definizione gruppo di campi

Nella tabella 5.11 sono mostrati tutti gli attributi previsti per un campo gruppo, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

### 5.3.2 Campo Gruppo ripetuto

La definizione yaml del campo gruppo ripetuto riflette la struttura imposta dalla service interface, vedi 21. Un campo gruppo ripetuto è specificato indicando il tag **!Occ**, un esempio di definizione di campo gruppo ripetuto è mostrato nel cod. 51.

!Grp: GrpModel				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
name		String	✓	
override	ovr	boolean		false
fields		array	✓	

Tabella 5.11: Attributi impostabili per un gruppo

```

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 }
    ] }

```

Sorgente 51: esempio definizione gruppo di campi ripetuto

!Occ: OccModel				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
name		String	✓	
override	ovr	boolean		false
times	x	int	✓	
fields		array	✓	

Tabella 5.12: Attributi impostabili per un gruppo ripetuto

Nella tabella 5.12 sono mostrati tutti gli attributi previsti per un campo gruppo ripetuto, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

### 5.3.3 Campo gruppo incorporato da interfaccia

La definizione yaml del campo incorporato riflette la struttura imposta dalla service interface, vedi 22. Un campo incorporato è specificato indicando il tag `!Emb`, un esempio di definizione di campo incorporato è mostrato nel cod. 52.

Un campo gruppo da interfaccia a differenza degli altri campi multipli non crea un elemento di contesto esplicito. I campi sono figli della struttura corrente, non di un elemento di contesto. Ma, l'elemento corrente implementa l'interfaccia, e questo crea un contesto implicito.

Nella tabella 5.13 sono mostrati tutti gli attributi previsti per un campo gruppo da interfaccia, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

### 5.3.4 Campo Gruppo da interfaccia

La definizione yaml del campo gruppo/interfaccia riflette la struttura imposta dalla service interface, vedi 23. Un campo gruppo/interfaccia è specificato indicando il tag `!GRP`, un esempio di definizione di campi



```

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 }

```

Sorgente 52: esempio definizione gruppo di campi inclusi da interfaccia

!Emb: <a href="#">EmbModel</a>				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
source	src	interface	✓	

Tabella 5.13: Attributi impostabili per un elenco di campi importato da una interfaccia

gruppo/interfaccia è mostrato nel cod. 53.

```

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 }

```

Sorgente 53: esempio definizione gruppo di campi da interfaccia

Un campo gruppo/interfaccia è simile al campo gruppo, la differenza è che i campi del gruppo non sono definiti singolarmente, ma tutti insieme importandoli dalla interfaccia. Il gruppo implementerà l'interfaccia.

Nella tabella 5.14 sono mostrati tutti gli attributi previsti per un campo gruppo da interfaccia, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

### 5.3.5 Campo Gruppo ripetuto da interfaccia

La definizione yaml del campo gruppo/interfaccia ripetuto riflette la struttura imposta dalla service interface, vedi 24. Un campo gruppo/interfaccia ripetuto è specificato indicando il tag `!OCC`, un esempio di definizione di campi gruppo/interfaccia ripetuto è mostrato nel cod. 54.

!GRP: <a href="#">GrpTraitModel</a>				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
name		String	✓	
override	ovr	boolean		false
typedef	as	interface	✓	

Tabella 5.14: Attributi impostabili per un gruppo da interfaccia

```

packages:
- name: com.example.test
  interfaces:
  - &Error
    name: IError
    len: 49
    fields:
    - !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 }

```

Sorgente 54: esempio definizione gruppo di campi ripetuto da interfaccia

Un campo gruppo/interfaccia ripetuto è simile al campo gruppo ripetuto, la differenza è che i campi del gruppo non sono definiti singolarmente, ma tutti insieme importandoli dalla interfaccia. Il gruppo implementerà l'interfaccia.

!OCC: <a href="#">OccTraitModel</a>				
attributo	alt	tipo	O	default
offset	at	int	✓	auto-calcolato
length	len	int	✓	
name		String	✓	
override	ovr	boolean		false
times	x	int	✓	
typedef	as	interface	✓	

Tabella 5.15: Attributi impostabili per un gruppo ripetuto da interfaccia

Nella tabella 5.15 sono mostrati tutti gli attributi previsti per un campo gruppo ripetuto da interfaccia, i relativi nomi abbreviati, il corrispondente tipo-dati, se l'attributo è obbligatorio o facoltativo, e l'eventuale valore di default.

## Capitolo 6

# Gradle plugin

Il gradle plugin `recfm-gradle-plugin` permette di creare più classi e interfacce usando uno più file di configurazione in formato `yaml`. Le librerie esterne utilizzate richiedono il java 8, quindi per eseguire questo plugin è necessario almeno il java 8.

Il gradle plugin `recfm-gradle-plugin` è semplicemente l'adattamento del `recfm-maven-plugin` per essere usato in un progetto *gradle*. Se si esclude il modo in cui vengono passati i parametri di input, e come viene attivato il plugin, il codice è la copia della versione per maven.

```
@Data
public class RecordFormatExtension {
    private String generateDirectory; // default: "${project.buildDir}/generated-sources/recfm"
    private String settingsDirectory; // default: "${project.projectDir}/src/main/resources"
    private String[] settings;
    private boolean addCompileSourceRoot = true;
    private boolean addTestCompileSourceRoot = false;
    private String codeProviderClassName;
}
```

Sorgente 55: parametri impostabili del gradle plugin

Come si vede col confronto del cod. 38 i parametri utilizzati sono gli stessi del maven-plugin, per la descrizione dei parametri rimando al § 5.

```
buildscript {
    dependencies {
        // plugin
        classpath 'io.github.epil55:recfm-gradle-plugin:0.7.0'
        // addon for java code generation
        classpath 'io.github.epil55:recfm-java-addon:0.7.0'
    }
}
apply plugin: 'RecordFormatPlugin'
recfm {
    settings 'recfm-foo.yaml'
}
```

Sorgente 56: esempio minimale di esecuzione del plugin

**Parte III**

**Service Provider**

# Capitolo 7

## Service Provider

Nei capitoli precedenti abbiamo visto la *Service Provider Interface*, che definisce delle interfacce e delle classi che permettono di definire i tracciati, e indicare alcuni comportamenti che dovranno essere usati in fase di utilizzazione dei tracciati; e alcuni esempi di *Service*, che semplicemente valorizza gli oggetti messi a disposizione della *Service Provider Interface*, ma il vero lavoro di generazione del codice è fatto dal *Service Provider*.

La struttura SPI consente di avere codice generato diverso, implementato in modo diverso, o addirittura generare sorgente in un linguaggio diverso.

Qualunque sia il linguaggio generato e il dettaglio della implementazione il *Service Provider* dovrà fornire alcune funzionalità generali.

- **decode**: partendo dalla stringa-dati, deve istanziare la classe-dati;
- **setter, getter**: la classe-dati generata deve fornire i metodi di accesso ai singoli campi;
- **costruttore vuoto**: la classe-dati può essere istanziata con i valori di default dei campi;
- **encode**: la classe-dati può essere serializzata nella stringa-dati.

Sarebbe gradita anche qualche funzionalità accessoria:

- **validate**: validare la stringa-dati prima della de-serializzazione, in modo da segnalare tutte le aree che non possono essere assegnate ai relativi campi, tipicamente caratteri non numerici in campi di tipo numerico;
- **cast**: se due stringhe-dati hanno la stessa lunghezza, poter passare da una classe-dati che le rappresenta all'altra;
- **toString**: fornire un metodo che mostra tutti i valori dei campi che compongono la classe-dati.
- (deep) **copy**: genera una copia della classe-dati;

### 7.1 Generazione sorgente java — java-addon

Le classi generate dal *CodeProvider* java oltre ai setter e getter hanno una serie di metodi ausiliari, vedi cod. 57.

- viene fornito un costruttore senza argomenti, che crea la classe con i valori a default
- viene fornito un costruttore *cast-like*, che prende come argomento qualunque altra classe che rappresenta una classe-dati
- viene fornito un costruttore da stringa-dati (de-serializzatore)
- viene fornito un metodo *deep-copy* per duplicare la classe-dati
- vengono forniti due metodi di validazione
- viene fornito un metodo di `toString`

```

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
}

```

Sorgente 57: esempio di classe generata (Foo312)

- viene fornito un metodo per generare la stringa-dati (serializzatore)
- viene fornito un metodo per recuperare la lunghezza della stringa-dati

Le classi generate dai file di configurazione ereditano delle classi generali con metodi comuni per la gestione dei setter/getter dei controlli e le validazioni. Queste classi sono fornite come libreria esterna, vedi [58](#).

```

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

```

Sorgente 58: dipendenze runtime dell'addon-java

Queste librerie sono compilate in compatibilità java-5, e contengono il *module-info* per poter essere correttamente gestite anche con java-9 e superiori.

### 7.1.1 Validazione

Come visto nel sorgente [57](#) per ogni classe vengono generati due metodi di validazione. Per entrambi l'argomento è una interfaccia dedicata, ma questo è per compatibilità pre-java-8. L'argomento sarà una *closure*, implementata con una classe anonima o interna o una  $\lambda$ -function.

```

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

```

Sorgente 59: gestore errori FieldValidateHandler

I metodi di validazione indicano se la stringa-dati acquisita con il costruttore statico decode ha superato la validazione richiesta dalla definizione dei campi oppure no, ma ogni volta che viene rilevato un errore di validazione viene chiamato il metodo error dell'interfaccia fornita come argomento con i dettagli dell'errore. In questo modo è possibile accumulare tutti gli errori di validazione rilevati. La differenza tra i due metodi è che il primo validateFails in caso di più caratteri errati sullo stesso campo, segnala solo il primo, mentre il secondo validateAllFails segnala tutti i caratteri in errore.

L'argomento del metodo error è l'interfaccia FieldValidateError, che sostanzialmente è una java-bean che espone solo i getter in formato *fluente*. Segnaliamo che alcuni valori possono essere null. Un campo di tipo costante non ha un nome (name). Un campo di tipo costante o dominio o custom con un controllo impostato con espressione regolare non ha un carattere sbagliato (wrong) ad una ben precisa colonna (column). Nel messaggio di errore, se è possibile identificare il carattere in errore, viene mostrata la posizione del carattere relativa al

```

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
}

```

Sorgente 60: dettaglio errore FieldValidateError

campo (non alla stringa-dati), il carattere (se è un carattere di controllo viene mostrata la codifica unicode), il *nome* del carattere, e il tipo di errore; altrimenti viene mostrato il valore del campo e il tipo di errore.

```

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

```

Sorgente 61: categoria errore ValidateError

I possibili tipi di errore sono mostrati nel sorgente [61](#), il significato è evidente dal nome.

## 7.1.2 Setter e getter

L'implementazione della classe-dati usata da questa libreria, in realtà non genera dei campi de-serializzati. Quando la classe viene creata dalla stringa-dati, il costruttore statico si limita a salvare internamente la stringa-dati come vettore di caratteri. Il getter di un campo accede all'intervallo di caratteri corrispondenti al campo e li de-serializza. Analogamente il setter serializza il valore fornito e lo copia nell'intervallo di caratteri corrispondenti al campo. In questo modo è banale fare un *override* di un campo, e il costruttore *cast-like* e *deep-copy* sono quasi a costo zero. Anche i metodi *encode* e *decode* sono sostanzialmente a costo zero perché le operazioni di serializzazione/deserializzazione sono in realtà eseguite dai setter/getter.

## 7.1.3 Campi Singoli

### Gestione valore null

In una stringa-dati non è rappresentabile un valore nullo, a meno che convenzionalmente si assegni ad una particolare stringa tale valore, come nei campi di tipo numerico-nullabile. Quando un setter formalmente imposta il valore null, nella rappresentazione della stringa-dati in realtà verrà assegnato il valore di default del campo: spazio per un tipo alfanumerico, zero per un tipo numerico, lo *initChar* per un tipo custom, e il primo valore tra quelli definiti come possibili per un campo dominio.

```

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

```

Sorgente 62: Accesso a valori numerici come stringhe e numeri primitivi

### Accesso Both per campi numerici

I campi numerici possono essere gestiti come stringhe di caratteri numerici o come oggetti numerici primitivi. È possibile configurare i campi per avere getter/setter di tipo stringa o numerico primitivo o entrambi. Nel caso venga scelto "entrambi" non è possibile definire il getter con il nome canonico per entrambi i tipi. In questi casi il nome canonico viene usato per il getter di tipo stringa, il getter con il tipo primitivo ha come nome il tipo primitivo e il nome del campo, vedi [62](#).

```

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

```

```

io.github.epil55.recfm.java.NotDomainException: com.example.BarDom.setCur, offending value "AAA"
at com.example.test.TestBar.testDomain(TestBar.java:203)
...

```

Sorgente 63: Eccezione sul setter

## Controlli sui setter e getter

Se sono attivi i controlli sui setter e viene impostato un valore non permesso viene lanciata una eccezione che segnala la violazione del controllo. L'eccezione posiziona lo stacktrace sulla istruzione del setter.

```

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

```

```

io.github.epil55.recfm.java.NotDomainException: com.example.BarDom.getCur, offending value "AAA" @1+3
at com.example.test.TestBar.testDomain(TestBar.java:303)
...

```

Sorgente 64: Eccezione sul getter

Analogamente sui getter. Se la stringa-dati contiene nella zona corrispondente a un campo un valore non valido per il campo, non viene fatta la validazione della struttura, che avrebbe segnalato il problema, e il codice continua fino al getter, viene lanciata una eccezione. L'eccezione posiziona lo stacktrace sulla istruzione del getter.

### 7.1.4 Campi Multipli

In questo contesto considereremo campi multipli solo i campi di tipo gruppo o gruppo ripetuto, definiti direttamente o tramite interfaccia. Questo tipo di campi genera un elemento intermedio. Come si vede dal sorgente 65, generato per un gruppo definito da interfaccia, viene creata una classe interna per gestire l'elemento intermedio, un campo privato con una istanza dell'elemento intermedio, un *getter fluente* del campo, e un *Consumer* del campo.

```

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); }

```

Sorgente 65: Definizione di un gruppo interno alla classe-dati

La classe interna implementerà l'interfaccia di validazione, e, se definito tramite interfaccia, l'interfaccia con la definizione del dettaglio dei campi della classe interna. Ogni gruppo è validabile singolarmente come se fosse una classe-dati. L'interfaccia di validazione, *Validable*, richiede il metodo *validateFails* che abbiamo già incontrato nella validazione della classe-dati. Anche tutte le classi-dati implementano l'interfaccia *Validable*.



```

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

```

Sorgente 66: Interfaccia di validazione, a livello classe-dati e gruppo

La definizione di un gruppo ripetuto è simile a quello di un gruppo. Viene creata una classe interna per gestire l'elemento intermedio ripetuto, un campo privato con  $n$  istanze dell'elemento intermedio, un *getter* *fluente* con un indice del campo, e un *Consumer* con indice del campo.

```

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]); }
}

```

Sorgente 67: Definizione di un gruppo ripetuto interno alla classe-dati

Anche in questo caso la classe interna che definisce l'elemento intermedio implementa l'interfaccia di validazione, e, se definito tramite interfaccia, l'interfaccia con la definizione del dettaglio dei campi della classe interna.

# Elenco delle figure

1	Esempio di file-dati posizionale . . . . .	1
2	Struttura service, service-provider-interface, service-provider . . . . .	1

# Elenco delle tabelle

5.1	Attributi impostabili per la definizione di una interfaccia . . . . .	23
5.2	Attributi impostabili per la definizione di una classe . . . . .	24
5.3	Tag yaml per la identificazione del campo . . . . .	24
5.4	Attributi impostabili per un campo alfanumerico . . . . .	25
5.5	Attributi impostabili per un campo numerico . . . . .	26
5.6	Attributi impostabili per un campo custom . . . . .	27
5.7	Attributi impostabili per un campo numerico nullabile . . . . .	28
5.8	Attributi impostabili per un campo dominio . . . . .	29
5.9	Attributi impostabili per un campo filler . . . . .	29
5.10	Attributi impostabili per un campo costante . . . . .	30
5.11	Attributi impostabili per un gruppo . . . . .	31
5.12	Attributi impostabili per un gruppo ripetuto . . . . .	31
5.13	Attributi impostabili per un elenco di campi importato da una interfaccia . . . . .	32
5.14	Attributi impostabili per un gruppo da interfaccia . . . . .	33
5.15	Attributi impostabili per un gruppo ripetuto da interfaccia . . . . .	33

# Elenco dei sorgenti

1	interfaccia CodeProvider e recupero del CodeProvider dal ServiceLoader . . . . .	5
2	interfaccia CodeFactory . . . . .	5
3	interfaccia ClassModel . . . . .	6
4	interfaccia TraitModel . . . . .	6
5	classe GenerateArgs . . . . .	6
6	classe FieldDefault . . . . .	7
7	classe ClsDefault . . . . .	7
8	interfaccia AbcModel (campo alfanumerico) . . . . .	8
9	class AbcDefault (default campo alfanumerico) . . . . .	9
10	interfaccia NumModel (campo numerico) . . . . .	9
11	class NumDefault (default campo numerico) . . . . .	10
12	interfaccia CusModel (campo custom) . . . . .	11
13	class CusDefault (default campo custom) . . . . .	11
14	interfaccia NuxModel (campo numerico nullabile) . . . . .	11
15	class NuxDefault (default campo numerico nullabile) . . . . .	12
16	interfaccia DomModel (campo dominio) . . . . .	12
17	interfaccia FilModel (campo filler) . . . . .	12
18	class FilDefault (default campo filler) . . . . .	13
19	interfaccia ValModel (campo costante) . . . . .	13
20	interfaccia GrpModel (campo gruppo) . . . . .	14
21	interfaccia OccModel (campo gruppo ripetuto) . . . . .	14
22	interfaccia EmbModel (campo incorporato) . . . . .	15
23	interfaccia GrpTraitModel (campo gruppo/interfaccia) . . . . .	15
24	interfaccia OccTraitModel (campo gruppo/interfaccia ripetuto) . . . . .	16
25	enum LoadOverflowAction . . . . .	17
26	enum LoadUnderflowAction . . . . .	17
27	enum CheckAbc . . . . .	17
28	enum OverflowAction . . . . .	17
29	enum UnderflowAction . . . . .	17
30	enum NormalizeAbcMode . . . . .	18
31	enum NormalizeNumMode . . . . .	18
32	enum WordWidth . . . . .	18
33	enum AccesMode . . . . .	18
34	enum InitializeNuxMode . . . . .	18
35	enum CheckCus . . . . .	18
36	enum AlignMode . . . . .	18
37	recupero del CodeProvider . . . . .	20
38	parametri impostabili del maven plugin . . . . .	21
39	esempio minimale di esecuzione del plugin . . . . .	22
40	classe di configurazione MasterBook . . . . .	22
41	classe di configurazione ClassPackage . . . . .	23
42	configurazione, area packages / interfaces / classes . . . . .	23
43	esempio definizione campi alfanumerici . . . . .	25
44	esempio definizione campi numerici . . . . .	26
45	esempio definizione campi custom . . . . .	27
46	esempio definizione campi numerici nullabili . . . . .	28
47	esempio definizione campi dominio . . . . .	28

48	esempio definizione campi filler . . . . .	29
49	esempio definizione campi costanti . . . . .	30
50	esempio definizione gruppo di campi . . . . .	30
51	esempio definizione gruppo di campi ripetuto . . . . .	31
52	esempio definizione gruppo di campi inclusi da interfaccia . . . . .	32
53	esempio definizione gruppo di campi da interfaccia . . . . .	32
54	esempio definizione gruppo di campi ripetuto da interfaccia . . . . .	33
55	parametri impostabili del gradle plugin . . . . .	34
56	esempio minimale di esecuzione del plugin . . . . .	34
57	esempio di classe generata (Foo312) . . . . .	37
58	dipendenze runtime dell'addon-java . . . . .	37
59	gestore errori FieldValidateHandler . . . . .	37
60	dettaglio errore FieldValidateError . . . . .	38
61	categoria errore ValidateError . . . . .	38
62	Accesso a valori numerici come stringe e numeri primitivi . . . . .	38
63	Eccezione sul setter . . . . .	39
64	Eccezione sul getter . . . . .	39
65	Definizione di un gruppo interno alla classe-dati . . . . .	39
66	Interfaccia di validazione, a livello classe-dati e gruppo . . . . .	40
67	Definizione di un gruppo ripetuto interno alla classe-dati . . . . .	40

# Indice analitico

AbcDefault, [9](#)  
AbcModel, [8](#)  
AccesMode, [18](#)  
AlignMode, [18](#)  
  
CheckAbc, [17](#)  
CheckCus, [18](#)  
ClassModel, [6](#)  
ClsDefault, [7](#)  
CodeFactory, [5](#)  
CodeProvider, [5](#)  
CusDefault, [11](#)  
CusModel, [11](#)  
  
DomModel, [12](#)  
  
EmbModel, [15](#)  
  
FieldDefault, [7](#)  
FilDefault, [13](#)  
FilModel, [12](#)  
  
GenerateArgs, [6](#)  
GrpModel, [14](#)  
GrpTraitModel, [15](#)  
  
InitializeNuxMode, [18](#)  
  
LoadOverflowAction, [17](#)  
LoadUnderflowAction, [17](#)  
  
NormalizeAbcMode, [18](#)  
NormalizeNumMode, [18](#)  
NumDefault, [10](#)  
NumModel, [9](#)  
NuxDefault, [12](#)  
NuxModel, [11](#)  
  
OccModel, [14](#)  
OccTraitModel, [16](#)  
OverflowAction, [17](#)  
  
plugin  
    addCompileSourceRoot, [21](#)  
    addTestCompileSourceRoot, [22](#)  
    codeProviderClassName, [22](#)  
    generateDirectory, [21](#)  
    settings, [21](#)  
    settingsDirectory, [21](#)  
  
TraitModel, [6](#)  
  
UnderflowAction, [17](#)  
  
ValModel, [13](#)  
  
WordWidth, [18](#)