# Relazione del Progetto di Linguaggi e Compilatori

Luca Cividini, Fabio Marini, Antonio Riva

giugno 2007

# Indice

# Capitolo 1

# Introduzione

L'acronimo cUml2Svg sta per "Coded UML to SVG"; il progetto consiste in un traduttore da una rappresentazione in codice dei digrammi UML delle classi alla rappresentazione grafica tipica dell'UML in SVG.

SVG è un formato grafico vettoriale basato su XML.

Il progetto include anche un'interfaccia grafica costruita come estensione del popolare ambiente di sviluppo Eclipse [**?**]

# Capitolo 2

# Il linguaggio

Il linguaggio che stiamo proponendo è strutturato in 2 parti, e quindi due tipi di file, che corrispondono al concetto di modello e layout; non a caso l'estensione del primo è u2sm e il secondo è u2sl.

Il file modello sarà il contenitore delle definizione delle classi mentre nel file layout il programmatore indicherà l'aspetto grafico del diagramma che si va a generare.

## 2.1 Model

Il modello conterrà quindi tutte le informazioni riguardo il contenuto/i dati degli elementi che sarà possibile inserire in un diagramma in un secondo momento.

In un linguaggio ad oggetti uno degli strumenti fondamentali è la suddivisione delle classi sotto forma di package. Nella nostra proposta un file di modello può contenere o una lista di classi (non associate ad un package) oppure una lista di package.

La definizione del package (come un po' tutto il nostro linguaggio) è ispirata alla sintassi dei linguaggi java like.

Listing 2.1: Dichiarazione di package

```
package nome.package{
   ...
}
```

Nella definizione della classe è invece dichiarato in modo esplicito la suddivisione tra i componenti base di una classe; sono quindi esplicitati quali sono gli attributi, i metodi e le relazioni. L'aggiunta delle relazioni rispetto ad una classica definizione di classe è necessaria per poter rappresentare nel diagramma le frecce di collegamento.

Listing 2.2: Dichiarazione di classe

```
class NomeClasse{
  relations{
     ...
  }
  attributes{
```

```
    ...
  }
  methods{
    ...
  }
}
```

Analizziamo ora il contenuto delle varie sezioni.

### 2.1.1 Relazioni

Nella sezione delle relazioni viene appunto dichiarata una lista di definizione che contiene il tipo della ralazione, una lista di classi (complete di package) con cui creare una relazione del tipo scelto per ogni classe 3 parametri opzionali (che possono anche essere vuoti) in cui il primo parametro e l'ultimo possono essere utilizzati per la definizione delle cardinalità, mentre il parametro opzionale puù essere utilizzato per definire una descrizione della relazione; queste stringhe andranno a posizionarsi sul disegno delle frecce.

Listing 2.3: Dichiarazione di relazione

```
relations{
  extend nome.package.NomeClasse ("(1,*)" ,
        "descrizione relazione", "(1,1)" );
  use nome.package.NomeClasse1, nome.package.NomeClasse2;
}
```

La lista dei possibili tipi di relzione sono:

- use (uso);
- extends (ereditarietà);
- associate (associazione);
- include (inclusione);
- realize (realizzazione);
- depend (dipendenza);
- composed (composizione);

### 2.1.2 Attributi

Gli attributi vengono dichiarati come visibilità:

- public
- privare
- protected

tipo (opzionale), nome attributo e valore di default (opzionale); la sintassi è quella inidicata nell'esempio sottostante.

Listing 2.4: Dichiarazione di attributi

```
attributes{
  public int attr1=5;
  public attr2;
  private attr3;
}
```

### 2.1.3 Metodi

Anche in questo caso la definizione dei metodi è del tutto simile alla difinizione della segnatura di un metodo in java.

Listing 2.5: Dichiarazione di metodi

```
methods{
  public nomeMetodo1(int arg1=10, String arg2,arg3);
  private nomeMetodo2(arg1);
  nomeMetodo3();
}
```

Nell'esempio di definizione sopra riportato si possono contraddistinguere varie caratteristiche aggiuntive del nostro linguaggio; un metodo è caratterizzato oltre che dal suo nome anche dal tipo di visibilità.

Gli attributi possono essere definiti sia con l'assegnazione di un tipo che con la definizione del solo nome, c'è anche la possibilità di assegnare un valore di default al parametro del metodo (caratteristica disponibile in alcuni linguaggi tipo il PHP). Possiamo notare il fatto che è possibile omettere gran parte degli attributi perchè l'idea è che deve essere obbligatorio definire solo gli elementi minimi per poi poter indicare, opzionalmente, solo ciò che si vuole rappresentare nel diagramma.

## 2.2 Grammatica

- s ->layout /
- layout ->import_definition+ main_group
- import_definition ->IMPORT FILE_NAME ;
- main_group ->groups
- groups ->group_preference*
  (groups |group_definition)+
- group_preference ->layout_preference

- group_preference ->margin_preference
- group_preference ->collapse_preference
- group_preference ->args_preference
- group_definition ->“(” (CLASS_TYPE
  |CLASS_NAME_WITH_PACKAGE |CLASS_NAME
  |CLASS_RANGE_WITH_PACKAGE) “)”
- layout_preference ->LAYOUT LAYOUT_CARD
- collapse_preference ->COLLAPSE COLLAPSE_TYPE
- args_preference HIDE_ARGS
- margin_preference ->MARGIN MARGIN_SIZE_TOP
  MARGIN_SIZE_RIGHT MARGIN_SIZE_BOTTOM MARGIN_SIZE_LEFT
- model ->(class_definition+ |package_definition+) /
- package_definition ->PACKAGE PACKAGE_NAME { class_definition+ }
- class_definition ->COMMENT? VISIBILITY?
  CLASS_TYPE CLASS_NAME { relations? attributes? methods? }
- relations ->RELATIONS { relation }
- relation ->COMMENT? RELATION_TYPE
  (RELATION_CLASS_NAME
  |RELATION_CLASS_NAME_WITH_PACKAGE) relation_cardinality?
  (RELATION_COMMA (RELATION_CLASS_NAME
  |RELATION_CLASS_NAME_WITH_PACKAGE)
  relation_cardinality?)* RELATION_END
- relation_cardinality ->CARDINALITY_START CARDINALITY
  RELATION_CARDINALITY_COMMA CARDINALITY
  RELATION_CARDINALITY_COMMA CARDINALITY CARDINALITY_STOP
- attributes ->ATTRIBUTES { attribute+ }
- attribute ->COMMENT? VISIBILITY?
  (typed_attribute_name |attribute_name) default_value?
- attribute_name ->VARIABLE
- typed_attribute_name ->VARIABLE VARIABLE

- default_value ->EQUAL equal_to
- equal_to ->(NUMBER |STRING)
- methods ->METHOD { method }
- method ->COMMENT? VISIBILITY?
  (typed_method |method_name)
  "(" (method_arg (, method_arg)*)? ")" ;
- typed_method ->VARIABLE VARIABLE
- method_name ->VARIABLE
- method_arg ->COMMENT? (typed_argument
  |argument) default_value?
- argument ->VARIABLE
- typed_argument ->VARIABLE VARIABLE
- attribute_type ->type
- type ->VARIABLE

## 2.3 Controlli semantici

I controlli semantici effettuati sono quelli elencati sotto:

- non è possibile re-dichiarare una classe
- non è possibile re-dichiarare una metodo (con stessa segnatura) in una classe
- i nomi degli attributi di un metodo non si possono ripetere nella stessa dichiarazione
- non è possibile re-dichiarare una relazione in una classe (ugual tipo, sorgente, destinazione)
- non è possibile re-dichiarare un attributo in una classe (ugual tipo e nome)
- non è possibile inserire la stessa classe in un diagramma

## 2.4 Implementazione generatore SVG

La funzione principale del linguaggio cUml è quella di permettere la creazione di diagrammi UML partendo da un modello dati e uno schema di layout. Come dice il nome del programma il formato prescelto per l'esportazione è SVG ovvero un formato vettoriale basato su XML.

## 2.5 Plugin Eclipse

Per il progetto è stato sviluppato un plugin per Eclipse che si occupa di:

- text-highlight
- controllo degli errori
- compilazione

### 2.5.1 text-hilight

Il plugin gestisce i file *model* e *layout* in modi differenti; i file, infatti, presentano caratteristiche e strutture diverse tra loro. la classe che esegue la scansione del file e gestisce l'editor di *model* è la seguente:

Listing 2.6: ModelScanner

```
public class ModelScanner extends RuleBasedScanner {
  //TODO controllare tutte le keyword
  private static String[] keywords= { "class","package", "public",
      "private","interface", "methods","extend", "relations",
      "attributes"  };
  public ModelScanner(ColorManager manager) {
          manager.getColor(ColorConstants.DEFAULT)));
    ArrayList<IRule> rules= new ArrayList<IRule>();
    Token tokencomment = new Token(new TextAttribute(
            manager.getColor(ColorConstants.COMMENT)));
    //Add rule for processing instructions
    rules.add( new SingleLineRule("//", "\n", tokencomment));
    rules.add( new SingleLineRule("#", "\n", tokencomment));
    rules.add( new MultiLineRule("/*", "*/", tokencomment));

    Token token= new Token(new TextAttribute(
        manager.getColor(ColorConstants.KEYWORD),    //parola
        null,    //sfondo
        SWT.BOLD));
    WordRule wordRule = new WordRule(new WordDetector());
    for (int i = 0; i < keywords.length; i++) {
      wordRule.addWord(keywords[i], token);
    }
    rules.add(wordRule);

    token = new Token(new TextAttribute(
              manager.getColor(ColorConstants.BRACET)));
    RuleBrace braceRule = new RuleBrace(token);
    rules.add(braceRule);

    IRule[] r= new IRule[rules.size()];
```

```
    setRules(rules.toArray(r));
  }
}
```

la classe che esegue la scansione del file e gestisce l'editor di *layout*, invece, è la seguente:

Listing 2.7: LayoutScanner

```
prendere quella aggiornata
```

In entrambe le classi le parole chiave sono specificate nella stringa *Keywords*, vengono definite le regole per i commenti, mentre per la gestione delle parentesi sono state definite delle regole comuni per entrambi i modelli.

### 2.5.2 controllo degli errori

il controllo degli errori viene eseguito ogni volta che il file viene salvato; a livello di editor sono stati usati i marker laterali che indicano il tipo di errore commesso (figura 2.1)

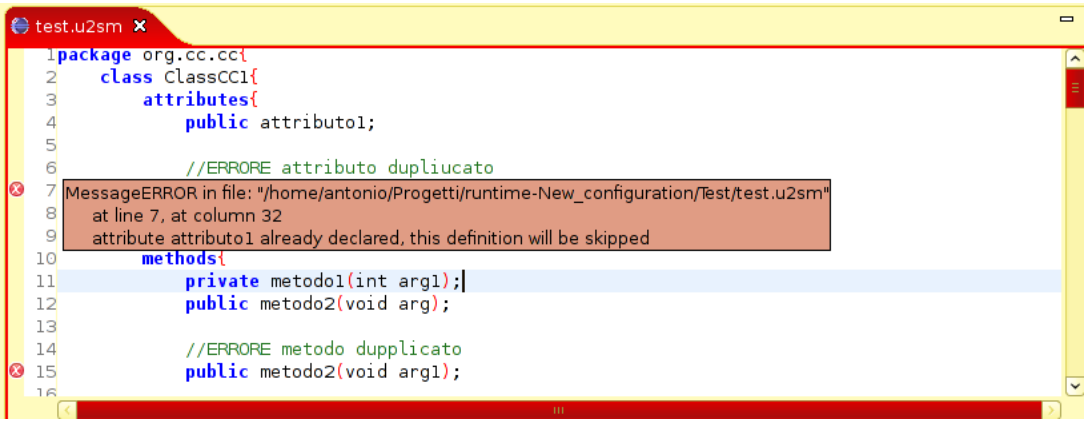


Figura 2.1: Gestione degli errori

# Capitolo 3

# Esempi d'uso

In questo capitolo saranno presentati alcuni esempi di utilizzo del linguaggio cUml sia per la realizzazione del modello dati che per la specifica del layout di impaginazione.

## 3.1 Esempo di modello dati

Il modello dei dati è rappresentato all'interno di files con estensione u2sm.

### 3.1.1 Primo esempio semplice

Il seguente listato definisce una classe di nome Class1 con un attributo pubblico di tipo intero e con valore iniziale pari a 0. La classe ha inoltre un metodo pubblico print.

Listing 3.1: Semplice esempio di modello

```
class Classe1 {
  attributes {
    public int x = 0;
  }

  methods {
    public void print();
  }
}
```

### 3.1.2 Un esempio un po' più complesso

In questo esempio viene mostrato l'uso delle relazioni tra due classi.

Listing 3.2: Un esempio un po' più complesso

```
package org.example {
  class Classe1 {
    attributes {
      public int x = 0;
```

```
    }

    methods {
      public void print();
    }
  }

  class Classe2 {
    relations {
      extend Class1 ("(1,*)","estende","(1,*)");
    }

    attributes {
      private int value = 0;
    }

    methods {
      public void setValue(int newValue);
      public int getValue();
    }
  }
}
```

## 3.2 Esempio di layout

Quando si è conclusa la realizzazione del modello dati è possibile procedere alla definizione del layout che indicherà al programma come disporre gli oggetti all'interno del diagramma SVG.

### 3.2.1 Il primo semplice layout

In questo esempio viene mostrato come includere il file di modello con gli oggetti da rappresentare e come includerli nel diagramma in output. Si considera come input il modello riportato nel listato 3.2

Listing 3.3: Un semplice esempio

```
import example3.2.u2sm;
[
  (class org.example.*)
]
```

Questo comando indica di includere tutte le classi del package org.example.

### 3.2.2 Un esempio un po' più complesso

In questo secondo esempio viene mostrato come includere il file di modello con la definizione degli oggetti e si utilizzano alcuni parametri avanzati come:

- @hide−args che permette di nascondere i parametri dei metodi
- @margin che permette specificare i margini per un gruppo nell'ordine top right bottom left come da standard W3C.

Listing 3.4: Un semplice esempio

```
import example3.2.u2sm;
[
  @hide-args
  (class org.example.Class1)
  [
    @margin 0 0 0 50
    (class org.example.Class1)
  ]
]
```

# Bibliografia