ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Utilizzo di Neverlang per la creazione di Domain Specific Languages

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Viroli Mirko

Candidato
Terenzi Mirco

Correlatore
Prof. Aguzzi Gianluca

II Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Max 2000 characters, strict.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Write your intro here.

Struttura della Tesi

Capitolo 2

Background

2.1 Domain-Specific Languages

In modo del tutto opposto rispetto ai linguaggi general-purpose, progettati per poter essere utilizzati in ogni contesto con un'efficienza e un grado d'espressività relativamente uguali, i Domain Specific Language (DSL) sono ottimizzati per uno specifico ambito e risultano essere, in molti casi, una soluzione molto più naturale rispetto a quella fornita dai primi [Hud97]. Tra gli esempi più comuni di DSL troviamo SQL, LaTeX (utilizzato anche per la scrittura di questo documento) e CSS.

Nonostante la definizione di DSL sia chiara, non è altrettanto immediato definire se un linguaggio sia o meno un DSL. In questo caso, vi sono alcuni principi chiave da osservare [Fow10]:

- Un DSL è un linguaggio di programmazione e, come tale, la sua struttura dovrebbe essere progettata in modo da essere facile da comprendere per gli esseri umani e, al tempo stesso, eseguibile da un compilatore.

- Essendo un linguaggio, deve avere un senso di fluidità e la sua espressività deve essere derivata non solo da un'espressione individuale, ma anche dalla combinazione di più istruzioni.

- Coerentemente alla definizione, un DSL dovrebbe implementare l'insieme minimo di caratteristiche necessarie per poter supportare il dominio applicativo

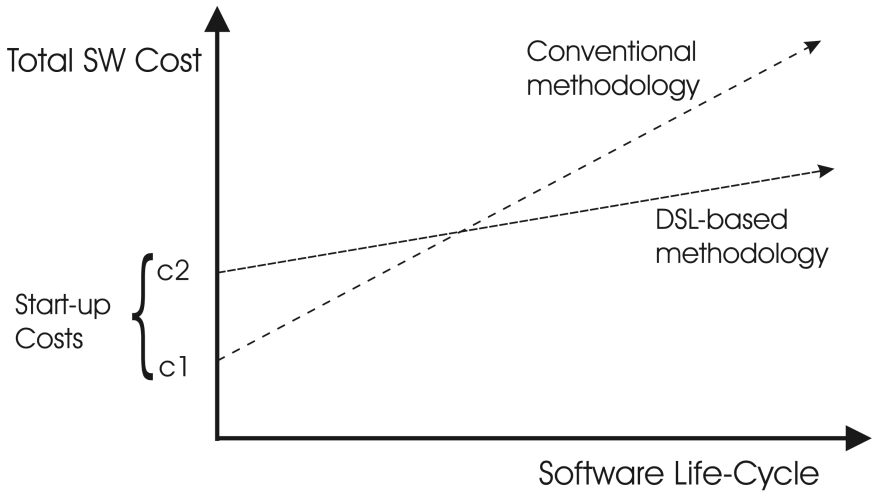

Figura 2.1: Il vantaggio ottenuto dall'utilizzo di DSL.

di interesse ed evitare funzioni non strettamente necessarie che potrebbero rendere il linguaggio più difficile, sia da utilizzare, sia da comprendere.

L'utilizzo di questa tipologia di linguaggi di programmazione comporta una serie di vantaggi:

- **Produttività**: È possibile aumentare il livello d'astrazione e, di conseguenza, aumentare la produttività. Questo perché è possibile utilizzare direttamente i concetti propri del dominio applicativo, non essendo limitati dalla necessità di mantenere una generalità atta ad ottenere un linguaggio applicabile in molteplici contesti [KT08]. Come indicato da Paul Hudak [Hud97] ed illustrato nella fig. 2.1, assumendo che il costo di sviluppo di un programma sia lineare, possiamo ipotizzare che il costo iniziale richiesto sia maggiore nel caso si utilizzasse un DSL rispetto a metodologie più convenzionali (considerando il caso in cui il linguaggio andasse sviluppato, se ne venisse utilizzato uno esistente tale costo sarebbe significativamente minore). Ciò nonostante, la pendenza della curva è considerevolmente più bassa e quindi, da un determinato punto in poi, l'utilizzo di DSL porterebbe un risparmio significativo.

- **Qualità del codice**: L'utilizzo di DSL favorisce anche una migliore qualità del codice. Infatti, il linguaggio può includere regole direttamente trasposte dal dominio all'interno del quale è applicato. In questo modo risulta molto più difficile, talvolta impossibile, ottenere dei risultati non attesi. Ad esempio, Antti Raunio, capo ingegnere del progetto EADS [Met07], afferma che "la qualità del codice generato è chiaramente migliore [...] perché il linguaggio di modellazione è stato progettato per adattarsi all'architettura del nostro terminale"[1]. Inoltre, l'offuscamento della reale complessità del problema, dovuto all'utilizzo di DSL, consente ai nuovi sviluppatori di lavorare ad un alto livello d'astrazione, senza dover conoscere tutti i dettagli inerenti all'implementazione del linguaggio [Met07].

- **Migliore manutenibilità**: Sebbene l'uso di DSL non renda l'implementazione necessariamente meno complessa di quanto si possa ottenere utilizzando un linguaggio *general-purpose*, la manutenibilità del codice risulta essere accentuata [KvdSV10]. Infatti, considerando il volume del codice, l'utilizzo di DSL comporta una minor quantità di codice da comprendere, facilitandone la modifica. Inoltre, è possibile ignorare il problema di mantenere coerente ciò che è definito dalla grammatica con la struttura gerarchica definita dall'Abstract Syntax Tree (AST) in quanto quest'ultimo si evolve con la prima [BM10].

2.2 Neverlang

Neverlang Language Workbench è un framework sviluppato presso l'Università di Milano dal professor Cazzola e dai suoi collaboratori, il cui scopo è favorire lo sviluppo di linguaggi di programmazione, in particolare secondo il paradigma di programmazione feature-oriented.

È basato sull'idea che i linguaggi di programmazione abbiano un'intrinseca divisione modulare in più caratteristiche, o *features*, ciascuna delle quali è implementata da un componente specifico. In accordo con tale visione, l'obiettivo

[1] Di seguito riportata l'affermazione citata, in lingua originale: "the quality of the generated code is clearly better, simply because the modelling language was designed to fit our terminal architecture"

del framework è definire i linguaggi tramite una divisione in frammenti, chiamati moduli, ognuno dei quali si occupa di implementare una specifica caratteristica e, infine, tramite la combinazione dei diversi moduli, ottenere un linguaggio di programmazione specifico per il contesto applicativo richiesto, ossia un DSL [Caz].

In particolare, all'interno di ogni modulo vengono definite due parti principali:

- la **sintassi**, utilizzando una grammatica formale;
- la **semantica**, in funzione della sintassi e sfruttando i vari elementi non-terminali e i loro attributi. Inoltre, il comportamento del componente può essere suddiviso in diverse fasi, ciascuna identificata da un ruolo specifico del componente.

Successivamente, i componenti del linguaggio vengono definiti combinando definizioni di sintassi e semantica provenienti da diversi moduli, all'interno di elementi detti *slice* [VC15].

Tra i vantaggi principali di Neverlang troviamo [Caz12]:

- **Modularità**: Ognuno dei moduli che compongono il linguaggio viene compilato separatamente, permettendo di utilizzarne uno o più di uno (in tal caso aggregandoli in uno *slice*) all'interno di altri linguaggi.
- **Riutilizzo**: Neverlang offre la possibilità di riutilizzare frammenti di linguaggio in più di un contesto. Ad esempio, un frammento può utilizzare la sintassi di un altro frammento definito in precedenza e ridefinire la semantica, o viceversa. Inoltre, è possibile ridefinire l'ordine dei simboli non-terminali utilizzati nella sintassi o nella semantica importata.
- **Estensibilità**: L'architettura modulare utilizzata all'interno di Neverlang facilita l'estensione di linguaggi esistenti. Per aggiungere nuove funzionalità non è necessario modificare il codice, ma è sufficiente integrare un nuovo *slice*.

2.3 Java

In aggiunta a Neverlang, per la realizzazione del progetto è stato utilizzato il linguaggio di programmazione Java. Java è un linguaggio di programmazione

ad alto livello, orientato agli oggetti e a tipizzazione statica, sviluppato da Sun Microsystems nel 1991. È molto diffuso e ben supportato, con una vasta comunità di sviluppatori e una grande quantità di librerie. Uno degli obiettivi principali di Java è quello di essere il più possibile indipendente dalla piattaforma di esecuzione, permettendo di scrivere una volta il codice e farlo eseguire su qualsiasi Java Virtual Machine (JVM), indipendentemente dall'architettura del computer [IBM].

Java è stato utilizzato per la realizzazione del progetto in quanto Neverlang è sviluppato per essere completamente integrato con esso. Il suo compilatore, nlgc, è stato sviluppato per poter convertire il codice scritto utilizzando il DSL di Neverlang, generando un nuovo codice supportato dalla JVM. Inoltre, Neverlang permette di utilizzare Java (ma non solo; anche Scala, ad esempio, è supportato) come linguaggio per la definizione della semantica all'interno dei moduli del DSL. Ciò è possibile in quanto gli accessi a variabili non-terminali, definiti all'interno della sintassi, sono sostituiti dallo specifico plug-in con accessi alla reale rappresentazione interna del linguaggio. In particolare, l'accesso alle variabili viene effettuato tramite una chiamata all'n-esimo figlio dell'AST [CV13].

Bibliografia

[BM10] Claus Brabrand and Pierre-Etienne Moreau. Preliminary proceedings of the tenth workshop on language descriptions tools and applications ldta 2010. 2010.

[Caz] Walter Cazzola. Neverlang 2: Language workbench. Ultimo accesso: 23 Agosto 2024.

[Caz12] Walter Cazzola. Domain-specific languages in few steps. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, pages 162–177, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CV13] Walter Cazzola and Edoardo Vacchi. Neverlang 2 – componentised language development for the jvm. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Software Composition*, pages 17–32, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Fow10] Martin Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

[Hud97] Paul Hudak. Domain specific languages. *Handbook of programming languages*, 3(39-60):21, 1997.

[IBM] IBM. What is java? Ultimo accesso: 23 Agosto 2024.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Pr, 2008.

[KvdSV10] Paul Klint, Tijs van der Storm, and Jurgen Vinju. On the impact of dsl tools on the maintainability of language implementations. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, New York, NY, USA, 2010. Association for Computing Machinery.

[Met07] MetaCase. Eads case study, 2007.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages Systems & Structures*, 43:1–40, 2015.