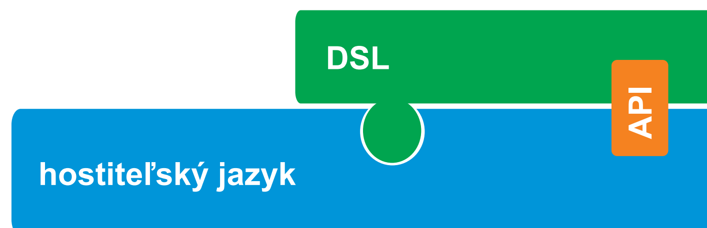
Interný doménovo-špecifický jazyk

Interný doménovo-špecifický jazyk (DSL) je závislý od zvoleného hostiteľského programovacieho jazyka využívajúc jeho syntax. Tvorba *interného DSL* je jednoduchšia oproti externému DSL, pretože sa využíva funkcionalita a syntax hostiteľského programovacieho jazyka (zvyčajne GPL). Autor interného DSL nepotrebuje mať znalosti o bezkontextových gramatikách, spracovaní jazyka a ani nemusí mať špeciálne nástroje ako v prípade jazykových prostredí. Vďaka využitiu hostiteľského jazyka, pri práci s vytvoreným DSL môžeme používať všetky nástroje IDE určeného pre zvolený hostiteľský jazyk. Tým sa značne zjednodušuje práca s DSL pomocou už existujúcich nástrojov, ktoré sú programátorovi dobre známe, a tým sa znižujú náklady potrebné k vytvoreniu nového DSL.

Naviazanosť na hostiteľský jazyk má aj svoje nevýhody. Veľkou nevýhodou je, že vytvorený interný DSL je obmedzený syntaxou hostiteľského jazyka. Program v DSL musí zodpovedať určenej štruktúre programu, pravidlám a obmedzeniam. Tým klesá flexibilita interného DSL a redukujú sa výhody použitia DSL.

Na interný DSL môžeme pozerať ako na akúsi nadstavbu nad hostiteľským jazykom (Obr. 1). Keďže hranica medzi interným DSL a štandardným API knižnice, či programovým rámcom je tenká, pri návrhu a implementácii interného DSL jazyka je možné využiť viacero vzorov. Pomocou týchto vzorov je možné dosiahnuť, aby spôsob tvorby a zápisu programu viac pripomínal prirodzený jazyk odstránením nadbytočnej syntaxe nesúvisiacej s doménou a tak zlepšoval celkovú čitateľnosť zdrojového textu.



Obr. 1: Architektúra interného DSL

Expression Builder

Spomínané vzory ovplyvňujú rozhranie interného DSL, pomocou ktorého je možné tvoriť program (vety) v tomto jazyku. Toto rozhranie sa nazýva konštruktor výrazov (angl. *Expression Builder*). Keďže každý vzor má svoje

špecifiká, na základe zvoleného vzoru sa líši aj rozhranie konštruktora výrazov. Rozdiel medzi jednotlivými vzormi je ukázaný na príkladoch, kde ako jazyk všeobecného použitia bola použitá Java.

Pre porovnanie je najprv uvedený príklad, na ktorom je ilustrovaný zápis použitím normálneho API (Zdroj. text 1). Príklad predstavuje interný DSL pre definíciu študenta a jeho vlastnosti.

```
Address a = new Address("Letná 9", "Košice");
Subject s1 = new Subject("Language-oriented programming", 2, 6);
Subject s2 = new Subject("Domain-specific languages", 3, 0);
return new Student("Ján", "Zelený", a, s1);
```

Listing 1: Použitie normálneho API

Method Chaining

V prípade použitia vzoru *zreťazenia metód* (angl. *Method Chaining*) je spôsob zápisu odlišný (Zdroj. text 2). Metódy rozhrania interného DSL, ktorých funkciou je nastavovať jednotlivé vlastnosti tried, používajú ako návratovú hodnotu samotný objekt. To umožňuje nastaviť viacero vlastností použitím metód rozhrania v rámci jedného výrazu. Problém pri použití zreťazenia metód je chýbajúca informácia, kedy sa má vytváraný objekt `Subject` priradať objektu `Student`. Možným riešením je pridanie metódy `save()` do rozhrania konštruktora výrazov, ktorej úlohou bude realizovať toto priradenie. Avšak pridaním tejto metódy sa naruší návrh rozhrania nadbytočnou syntaxou, ktorá nesúvisí s doménou.

```
student()
  .firstname("Ján")
  .surname("Zelený")
  .address()
    .street("Letná 9")
    .city("Košice")
  .subject()
    .name("Language-oriented programming")
    .grade(2)
  .subject()
    .name("Domain-specific languages")
    .grade(3)
  .end();
```

Listing 2: Použitie vzoru zreťazenia metód

Function Sequence

Ďalším vzorom je *postupnosť funkcií* (angl. *Function Sequence*). Na rozdiel od zreťazenia metód, kde sú metódy navrhnuté tak, aby bolo možné použiť špecifický zápis programu, je postupnosť funkcií tradičným volaním jednotlivých metód zaradom nezávisle na sebe (Zdroj. text 3). Ako je vidno na príklade, pomocou odsadenia textu je možné dosiahnuť podobnú čitateľnosť ako pri zreťazení metód. Hlavnou nevýhodou tohto vzoru je, že pri volaniach metód nie je jasný kontext volania. V prípade volania metódy `grade()` nie je implicitne jasné, vlastnosť ktorého objektu `Subject` sa má nastaviť. Kontext môže byť vyjadrený použitím členskej premennej `currentSubject`. Kvôli nutnosti použitia kontextových premenných je použitím tohto vzoru zdrojový text menej prehľadný.

```
student();
  firstname("Ján");
  surname("Zelený");
  address();
    street("Letná 9");
    city("Košice");
  subject();
    name("Language-oriented programming");
    grade(2);
    credits(6);
  subject();
    name("Domain-specific languages");
    grade(3);
```

Listing 3: Použitie vzoru postupnosť funkcií

Nested Function

Ďalším vzorom je *vnorenie funkcií* (angl. *Nested Function*), pri ktorom sú volania metód vnorené ako parametre volania ďalšej metódy (Zdroj. text 4). Zásadným rozdielom oproti predchádzajúcim vzorom je poradie vyhodnocovania jednotlivých metód. Pri zreťazení metód a postupnosti funkcií sa metódy vyhodnocujú „zľava doprava“, zatiaľčo pri vnorených volaniach funkcií sa najprv vyhodnotia parametre a až následne samotná volaná metóda. Dôsledkom poradia vyhodnocovania jednotlivých metód je to, že sú prístupné všetky hodnoty pri vyhodnocovaní poslednej metódy. Tým je vyriešený problém zreťazenia metód a chýbajúcej informácie o ukončení vytvárania objektu. Vnorenie funkcií zároveň zabezpečuje aj kontext volania metód.

Uvedené vzory majú jeden spoločný nedostatok. Aby bolo možné pri písaní programu v internom DSL použiť zápis uvedený v 2,3,4, je potrebné aby metódy rozhrania konštruktora výrazov boli globálne (resp. statické). Z hľadiska návrhu softvérových systémov je použitie globálnych funkcií častokrát nežiadúce. Riešením je použitie *obmedzenia viditeľnosti objektu* (angl. *Object Scoping*) v kombinácii s iným vzorom. Riešenie spočíva vo vložení DSL programu do podtriedy, ktorá dedí od triedy predstavujúcej konštruktor výrazov. Keďže konštruktor výrazov obsahuje metódy používané pri tvorbe DSL programu, pri písaní DSL programu v podtriede bude možné použiť uvedený zápis programu. Navyše v prípade potreby je možné pridať ďalšie metódy, prípadne prekryť existujúce metódy. Nevýhodou je, že takéto riešenie vyžaduje princíp dedenia vo zvolenom hostiteľskom GPL. Príklad použitia je uvedený spolu so vzorom vnorenia funkcií (Zdroj. text 5).

```
student(
  firstname("Ján"),
  surname("Zelený"),
  address(
    street("Letná 9"),
    city("Košice")
  ),
  subject(
    name("Language-oriented programming"),
    grade(2),
    credits(6)
  ),
  subject(
    name("Domain-specific languages"),
    grade(3)
  )
);
```

Listing 4: Použitie vzoru vnorenia funkcií

```
class MyStudent : StudentBuilder {
  protected override void build() {
    student(
      firstname("Ján"),
      surname("Zelený"),
      address(
        street("Letná 9"),
        city("Košice")
      ),
      subject(
        name("Language-oriented programming"),
        grade(2),
        credits(6)
      ),
      subject(
        name("Domain-specific languages"),
        grade(3)
      )
    );
  };
}
```

Listing 5: Použitie vzoru vnorenia funkcií s obmedzením viditeľnosti