Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Generování testovacích dat z anotací

Plzeň 2016

Bc. Michal Dékány

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 11. května 2016

Bc. Michal Dékány

Poděkování

Rád bych poděkoval vedoucímu Ing. Richardu Lipkovi, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování diplomové práce.

Abstrakt

Hlavním cílem této práce je vytvoření nástroje v *Javě*, který bude využívat náhodné generátory a umožní snadné generování realistických testovacích dat, jejichž příprava a údržba je obvykle časově náročná. Náhodné hodnoty budou generovány podle pravidel, které je možné přidat do definic tříd v podobě anotací. V rámci teoretické části práce byly popsány principy a možnosti generování náhodných dat a existující nástroje umožňující generování testovacích dat. Praktická část je věnována popisu návrhu a implementace knihovny *Java Objects Populator*, která je výsledkem této práce. Knihovna je schopna na základě anotací generovat náhodná data pro primitivní i komplexní atributy libovolného *Java* objektu a případně i vytvořit graf objektů, které jsou navzájem propojeny referencemi.

Abstract

The main goal of this thesis is to create a tool in *Java* that uses random generators to prepare simple realistic test data. Such a tool would simplify the preparation and maintanance of testing data which is usually a time-consuming activity. Random values will be created according to rules defined as annotations in Java class definition. The theoretical part of this thesis describes ways and principles how to generate random values and existing tools for generating test data. The practical part covers the design and implementation of the created library *Java Objects Populator* which represents the result of this thesis. The library is able to generate random data for both simple and complex attributes of any *Java* and in some cases also to create graph structures of objects which are connected via references.

Obsah

1 Úvod **1**

2 Generování náhodných dat **3**
2.1 Generování náhodných čísel 3
2.1.1 Tabulkové metody 3
2.1.2 Fyzikální metody 4
2.1.3 Výpočetní metody 5
2.2 Generování textových dat 7
2.2.1 Generátory využívající konečné automaty 7
2.2.2 Generátory využívající trie 8
2.2.3 Generátory využívající Markovovy řetězce 9
2.2.4 Generátory využívající slovníky 11

3 Existující nástroje pro generování testovacích dat **13**
3.1 DataFactory 13
3.2 Java Faker 15
3.3 generatedata 17
3.4 jeneratedata 19
3.5 random-data-generator 20
3.6 Random Beans 22
3.7 Feed4JUnit 24
3.8 PoDaM 28

4 Návrh knihovny **32**
4.1 Struktura knihovny 33
4.2 Použití knihovny 34
4.3 Závislost tříd a jejich atributů 35
4.3.1 Druhy závislostí 36
4.3.2 Závislosti v rámci jedné třídy 37
4.3.3 Anotace pro závislosti tříd 37
4.3.4 Generování závislých tříd 41

4.3.5 Závislost atributů 42
4.4 Generátory a jejich anotace 44
4.4.1 Číselné generátory 44
4.4.2 Textové generátory 46
4.4.3 Ostatní generátory 48
4.5 Populátory a jejich anotace 49
4.5.1 Číselné hodnoty 50
4.5.2 Textové hodnoty 50
4.5.3 Pole a kolekce 50
4.5.4 Zřetězení populátorů 51

5 Implementace knihovny 53
5.1 Struktura knihovny 53
5.1.1 Obecný popis tříd a jejich atributů 54
5.1.2 Továrny 55
5.1.3 Populátor objektů 57
5.1.4 Strategie pro osazení atributů 58
5.1.5 Populátory hodnot 61
5.1.6 Generátory hodnot 62
5.1.7 Poskytovatelé tříd 63
5.1.8 Vyhledávání instancí 64
5.1.9 *Session* pro náhodné generátory 65
5.2 Anotace 66
5.2.1 Označení a rozpoznávání typu anotací 66
5.2.2 Uživatelské anotace 67
5.3 Použité technologie 71
5.3.1 Vkládání závislostí 71
5.3.2 Logování 73
5.3.3 Testy 74

6 Možnosti rozšíření knihovny 75

7 Závěr 80

Literatura 84

A Souhrn informací o nástrojích pro generování *Java* objektů 85

B UML diagram tříd znázorňující strukturu knihovny 86

C UML diagram tříd pro továrny 87

D UML diagram tříd pro strategii vytváření instancí **88**

E UML diagram tříd pro strategii hledání instancí **89**

F UML diagram tříd pro výchozí strategii osazení atributů **90**

1 Úvod

Za účelem zajištění spolehlivosti vyvíjených aplikací je testování velice důležitou součástí procesu vývoje. Jelikož se aplikace stávají čím dál tím složitějšími a jejich vývoj se stává dražším, je velice důležitá automatizace každého procesu, který ji umožňuje. Jedním z těchto procesů je právě testování. Mnoho druhů testů je nyní prováděno automaticky bez potřeby testera (např. jednotkové či integrační testy), avšak příprava testů je převážně manuální proces, při kterém musí být většina kódu vytvořena ručně.

Jedním z hlavních problémů testování aplikací je příprava testovacích dat. V některých typech testů (např. testy uživatelských rozhraní) jsou testovány pouze jednoduché hodnoty jako jsou čísla, data, řetězce aj. Jiné testy jsou však zaměřeny na samotné jádro aplikace (např. doménový model), kde je často potřeba vytvářet netriviální testovací objekty, které se skládají nejenom z již zmiňovaných jednoduchých hodnot, ale také obsahují odkazy na jiné objekty, a tím vytvářejí různě komplexní struktury.

Některé druhy testů, především funkcionální a zátěžové testy, vyžadují velké množství realistických testovacích dat. Jejich příprava je únavný a stále se opakující úkol, a proto si testeři pomáhají využitím náhodných generátorů, které jsou k dispozici ve většině programovacích jazyků. Tyto generátory mohou být snadno použity pro vytvoření testovacích dat, ale obvykle jsou určeny pouze pro generování číselných, a proto si musí testeři připravit vlastní generátory. To ale vyžaduje, aby měli hlubší znalost testovaných datových struktur, aby mohly jednotlivé generátory správně nakonfigurovat. Navíc některé atributy testovaných objektů mohou být závislé na ostatních (například hmotnost a velikost objektu), a tato závislost by měla být respektována při jejich generování.

Tento způsob přípravy testů přináší mnoho kódu, který jsou testeři nuceni manuálně vytvořit, protože je nutné připravit a nastavit všechny objekty před tím, než mohou být použity pro testování. Což často obnáší tvorbu obsáhlých metod, které obsahují pouze volání odpovídajících konstruktorů a *setterů*. Údržba těchto metod může být náročná, protože se struktura jednotlivých objektů může časem změnit, a tím znehodnotit celý test. To narušuje výhody automatizovaného testování a obvykle nutí testery k úpravě testů při každé změně.

Proto vznikla tato práce, jejíž cílem je návrh a implementace knihovny, která bude schopna vygenerovat (náhodná) data pro atributy libovolného *Java* objektu a případně i vytvořit graf objektů, které jsou navzájem propojeny referencemi, v závislosti na pravidlech, která jsou součástí definice tříd v podobě anotací u jednotlivých atributů. Tím by mělo dojít k eliminaci manuálních příprav testovacích dat a problémů s tím souvisejících.

V rámci teoretické části jsou popsány metody generování náhodných čísel a dat, které bude možné využít během návrhu a implementace knihovny. Dále jsou popsány existující nástroje a techniky pro generování náhodných testovacích dat. Praktická část je zaměřena na samotný návrh a implementaci knihovny pro automatické generování testovacích dat na základě anotací. Závěr praktické části bude věnován možnostem dalšího rozšíření implementované knihovny a možnostem jejího praktického využití při testování softwaru.

2 Generování náhodných dat

Podle [17] se pokusy o generování náhodných dat počítačem objevily ihned se vznikem prvních číslicových počítačů. Tehdy se jednalo pouze o generování náhodných čísel, nicméně všechny metody pro generování náhodných dat jsou dodnes založeny na náhodných číslech.

Proto bude tato kapitola především zaměřená na popis metod a možností generování náhodných čísel. Dále budou popsány metody generování náhodných textových dat především kvůli požadavkům na jejich strukturu či realističnost. Generování ostatních druhů dat není nutné zdlouhavě popisovat, protože je závislé na jejich struktuře a bude převážně založeno, jak již bylo uvedeno, na generátorech náhodných čísel. V kapitole se mohou objevit pojmy a věty z teorie pravděpodobnosti a matematické statistiky. Jejich popis je možné nalézt v [18].

2.1 Generování náhodných čísel

Generování náhodných čísel je využíváno v různých aplikacích, kde je kladen důraz na proměnlivost systému. Ať už jde o šifrování, počítačové simulace nebo hazardní hry, potřebujeme v aplikacích často používat náhodná čísla nebo alespoň čísla, která se jako náhodná tváří. Tato kapitola tedy bude zaměřena na popis využívaných metod k jejich generování.

2.1.1 Tabulkové metody

Podle [17] byla jedna z prvních metod používaných pro generování náhodných čísel založena na využití rozsáhlých tabulek (posloupností) čísel. [7] uvádí, že v historii byla publikována celá řada takovýchto tabulek, kde některé z nich byly vytvořeny s využitím čísel ze sčítání lidu, z telefonních seznamů apod. Nejrozsáhlejší tabulky (milión dekadických náhodných číslic), které byly pořízeny s využitím tzv *elektronické rulety*, byly publikovány společností *RAND* (přibližně v roce 1955).

V mnohých aplikacích je běžně vyžadováno velké množství náhodných čísel, někdy i v řádech desítek až stovek tisíc. Pro uchování takového množství náhodných čísel spolu s programem a dalšími daty často nestačila operační paměť i velkých počítačů. Proto byly tabulky uloženy na vnějších pamětí (diskety, disky atd.). To však vede k nutnosti komunikace mezi vnitřní a vnější pamětí počítače, která značně zpomaluje generování čísel. Už jenom zavedení těchto tabulek do paměti počítače je zdlouhavá záležitost. Navíc se může stát, že program bude vyžadovat víc náhodných čísel než obsahují tabulky. Proto není využívání náhodných tabulek příliš vhodnou metodou pro generování náhodných čísel.

2.1.2 Fyzikální metody

Podle [7] je nejjednodušším způsobem generování čísel vrhání hracích kostek nebo házení mincí, avšak tyto metody není možné reálně využít pro generování čísel v počítačích. Vhodnější metodou je využití tzv. *fyzikálního generátoru*, který může být součástí počítače nebo může být připojený například jako periferie. Ten generuje v potřebný okamžik náhodné číslo, které je založené na analýze náhodných fyzikálních jevů, jako jsou například radioaktivní rozpad prvků, šum elektronických zařízení atd. I přesto, že je počítač deterministickým zařízením, probíhá v něm mnoho „náhodných“ jevů, které jsou ovlivněny aktivitou jeho uživatele. Příkladem může být pohyb kurzoru po obrazovce, prodleva mezi stisky kláves, pohyb hlaviček disku, výše příkonu, teplota procesu apod. Mnoho z těchto vlastností lze v moderních počítačích sledovat a využít pro generování náhodných hodnot.

Generátory náhodných čísel založené na těchto fyzikálních metodách se často nazývají *Pravé generátory náhodných čísel* (zkratka `TRNG`). Tato metoda je primárně využívána v zařízeních, kde je kladen důraz na skutečnou náhodu (např. herní automaty). Kromě výhody v podobě zcela náhodných čísel přináší tato metoda podle [7] řadu nevýhod. Závažnou nevýhodou je to, že okamžiky, při kterých v zařízení vznikají náhodné hodnoty, se obvykle liší od okamžiků, kdy je skutečně potřebujeme. Proto se může snadno stát, že budeme čekat na vygenerování nového náhodného čísla. Pro některé aplikace může být nevýhodné to, že není možné opakované získání stejné posloupnosti náhodných čísel

2.1.3 Výpočetní metody

Kvůli nedostatkům ostatních metod generování náhodných čísel začaly být podle [17] postupně využívány programově realizované generátory, které z posledního (nebo několika posledních) prvků posloupnosti náhodných čísel počítají podle vhodně zkonstruovaného vzorce další prvek, tzn. že je využíváno speciálních rekurentních vzorců [7].

Už na první pohled je vidět, že se nejedná o žádnou náhodu, a proto se generátory náhodných čísel založené na této metodě často nazývají *Generátory pseudonáhodných čísel* (zkratka `PRNG`). Nicméně pokud má generovaná posloupnost *pseudonáhodných* čísel patřičné statistické vlastnosti – zejména splňuje požadované pravděpodobnostní rozdělení a sousední prvky posloupnosti (dvojice, trojice, ap.) jsou statisticky nezávislé, lze ji použít stejně dobře jako posloupnost skutečně náhodných čísel získávanou například z fyzikálního generátoru (viz kapitola 2.1.2).

Hlavní výhodou této metody je její cenná vlastnost v *reprodukovatelnosti pseudonáhodné posloupnosti*, tzn. že ji lze zcela přesně kdykoliv zopakovat (např. při dalším běhu programu) [17]. Podle [7] je navíc jejich výhodou to, že je lze získat přímo počítačem a to rychle a v dostatečném počtu.

Generování čísel s rovnoměrným rozdělením

Pode [17] jsou nejčastěji používanými metodami pro generování rovnoměrně rozdělených nezáporných celých čísel tzv. *kongruentní metody*, které jsou založeny na využití vzorců typu:

$$y_{j+1} = (C_1 + C_2 y_j) \bmod M,$$

kde y_{j+1} je generovaný prvek náhodné posloupnosti, y_j je předchozí prvek posloupnosti a C_1, C_2, M jsou číselné parametry generátoru. Vhodnou volbou hodnot C_1, C_2, M a také výchozí hodnoty $y0$ (*seed*) lze ovlivnit kvalitu (zejména rovnoměrnost) generované posloupnosti. Největší možné číslo na výstupu generátoru může být $M - 1$. Obvykle se volí $M = 2^n$, kde n je počet bitů, na kterých se v příslušném počítači zobrazuje kladné číslo (bez znaménka).

Z uvedeného vztahu dále vyplývá, že generovaná posloupnost náhodných čísel je *periodická*, tzn. že se generované hodnoty pravidelně opakují. Kvalitní generátor by měl mít co nejdelší *periodu* opakování, která je menší než M

Vygenerované hodnoty mohou být transformovány na *normalizované rovnoměrné pravděpodobnostní rozdělení*, tedy na reálná čísla z intervalu $\langle 0, 1)$.

Generování čísel různých pravděpodobnostních rozdělení

Výše popsaný generátor produkuje pouze čísla v rovnoměrném rozložení, tzn. že každé číslo z intervalu má stejnou pravděpodobnost výskytu. V některých aplikacích, např. v simulacích systémů hromadné obsluhy [17], potřebujeme generovat náhodné hodnoty s normálním (Gaussovým) rozdělením, exponenciálním či vlastním. Pro přípravu takového generátoru můžeme využít některou z následujících (základních) metod:

- **Metoda inverzní transformace** – umožňuje *transformaci* náhodných čísel Y s normalizovaným rovnoměrným rozdělením na čísla X zadaná distribuční funkcí $F(x)$ jejich pravděpodobnostního rozdělení. Tzn. že generátorem normalizovaného rovnoměrného rozdělení vyrobíme konkrétní hodnotu y a transformujeme ji na odpovídající x podle vzorce $X = F^{-1}(Y)$ [17].
- **Vylučovací metoda** – experimentální metoda, při které je vyžadována znalost funkce hustoty pravděpodobnosti $f(x)$. Během hledání náhodné hodnoty jsou generovány náhodné body v dvourozměrném prostoru (souřadnici x z definičního oboru a y z oboru hodnot funkce), které se testují, zda leží pod křivkou funkce $f(x)$. Pokud ano, je vrácena náhodná hodnota x.
- **Centrální limitní věta pro normální rozdělení** – [17] tvrdí, že součet náhodných čísel s libovolným rozdělením má asymptoticky normální rozdělení se střední hodnotou a rozptylem danými součtem stejných hodnot a rozptylů pravděpodobnostních rozdělení prvků součtu. S využitím této věty lze odvodit například vzorec pro generátor náhodných čísel s normálním (Gaussovým) rozdělení.

2.2 Generování textových dat

Generování řetězců je běžným problémem, který lze jednoduše vyřešit generováním náhodného řetězce o zadané či náhodné délce, při kterém bude použito diskrétní rovnoměrné rozložení pravděpodobnosti výskytu každého znaku ze zadané abecedy. Bohužel takto generované řetězce nejsou příliš použitelné, protože nepředstavují žádná slova ani věty z žádného jazyka (nejsou realistické). Zároveň nemají žádnou strukturu (formát), a proto je není možné použít například pro barevné kódy, IP adresy atd. Proto je vhodnější využití pokročilejších metod pro generování náhodných řetězců, které budou popsány v následujících kapitolách.

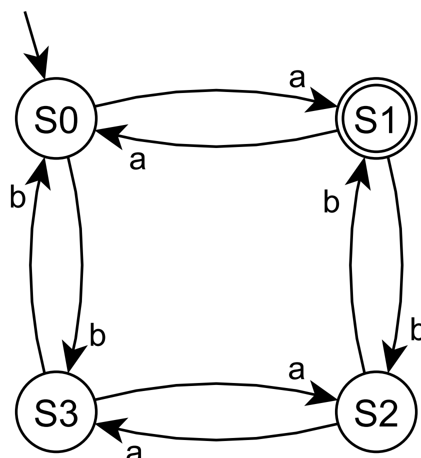
2.2.1 Generátory využívající konečné automaty

Pro možnost generování řetězců můžeme využít *konečných automatů*, které jsou podle [22] tvořeny konečným počtem *stavů*, mezi kterými jsou *přechody*. Automat má jediný *počáteční stav* a alespoň jeden *koncový stav*, který je někdy označován jako *přijímací stav*. Pro přechody mezi jednotlivými stavy slouží *vstupní symboly*, a proto musí každý automat obsahovat definici *přechodové funkce (tabulky)*. Automaty přijímají konečnou množinu vstupních symbolů, která se často nazývá *abeceda*. Na obrázku 2.1 je ukázka znázornění konečného automatu.

Konečné automaty se před začátkem jejich činnosti nacházejí v definovaném počátečním stavu. Pro jejich činnost je nutno zadat vstup, který je čten (zpracováván) po jednom symbolu. Po každém přečteném symbolu přejde do stavu daného hodnotou, která v přechodové tabulce odpovídá aktuálnímu stavu a přečtenému symbolu. Takto pokračuje do té doby, dokud není přečten celý vstup. Poté podle toho, zda skončil v některém z koncových stavů, je rozhodnuto, zda byl zadaný vstup přijat či zamítnut. Množina všech řetězců přijímaných automatem tvoří *regulární jazyk*.

Pro samotné generování řetězců je tedy možné využít přechodovou funkci (tabulku) libovolného konečného automatu. Generátor začne v počátečním stavu automatu a bude náhodně přecházet podle přechodové tabulky do libovolných dosažitelných stavů. Generovaný řetězec je tedy postupně skládán ze symbolů, které by bylo nutné použít k jednotlivým náhodným přechodům. Generování lze náhodně ukončit v případě, kdy se automat nachází ve svém

koncovém stavu. Další možností ukončení generování je zavedením omezující podmínky (např. pro maximální délku generovaného řetězce).



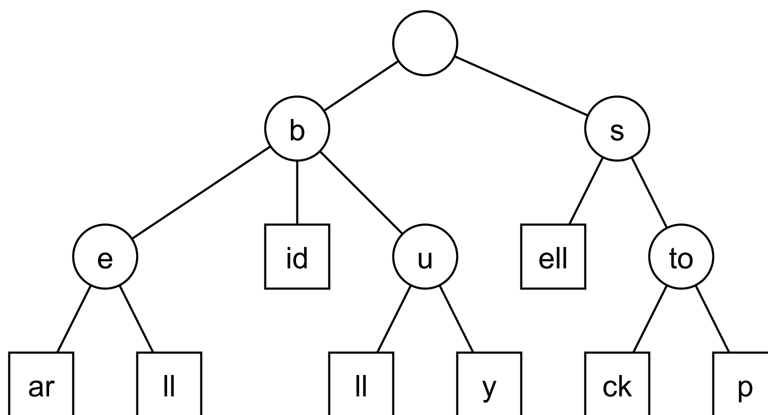
Obrázek 2.1: Ukázka konečného automatu akceptujícího lichý počet „a“ a sudý počet „b“

První možností využití tohoto způsobu generování je ruční sestavení automatu, jehož přechodová funkce bude použita pro generování náhodných řetězců. Další možností je pak využití tzv. *Regulárních výrazů*, které slouží k popisu množin řetězců (např. čísla SPZ, IP adresy atd.), tzn. že tvoří regulární jazyk. Tyto výrazy lze pak převést na konečný automat, jehož přechodová tabulka může být použita výše uvedeným algoritmem pro generování náhodných řetězců. Využití regulárních výrazů má výhodu v tom, že jsou dostupné ve většině programovacích jazyků či textových nástrojů, tzn. že jsou hojně využívány.

2.2.2 Generátory využívající trie

Podle [11] je *trie* kompaktní stromová datová struktura, která je vhodná pro uložení množiny řetězců, kterými mohou být například slova v textu. Jednotlivé uzly trie obsahují části uloženého řetězce, kde kořen je vždy prázdný, protože je asociován s prázdným řetězcem. Pokud je trie prohledávána, potom můžeme říct, že v každém uzlu obsahuje všechny podřetězce, kterými může pokračovat řetězec v dosud prohledané cestě. Všichni následníci uzlu

mají společný *prefix*, který je shodný s řetězcem přiřazeným k danému uzlu. Struktura trie je znázorněna na obrázku 2.2.



Obrázek 2.2: Ukázka struktury trie pro anglická slova *bear*, *sell*, *stock*, *bull*, *buy*, *bid*, *bell*, *stop*

Trie svojí strukturou mohou připomínat konečné automaty (viz kapitola 2.2.1), ale na rozdíl od automatů neobsahují cykly. Díky tomu bude generování náhodných řetězců z trie o mnoho jednodušší. Jedinou úlohou generátoru je náhodné procházení stromu od kořene k jednotlivým listům, tzn. že v každém uzlu bude jeho hodnota uložena do generovaného řetězce a bude vybrán náhodný uzel, do kterého bude generátor pokračovat. Jako zastavovací podmínka může sloužit maximální délka řetězce, maximální zanoření v trii apod. Algoritmus generování může být ukončen v libovolném uzlu trie nebo při dosažení listového uzlu. Trie také může obsahovat v jednotlivých uzlech informaci, zda je v nich ukončeno uložené slovo. Potom algoritmus generování může být ukončen pouze v takto označených uzlech.

2.2.3 Generátory využívající Markovovy řetězce

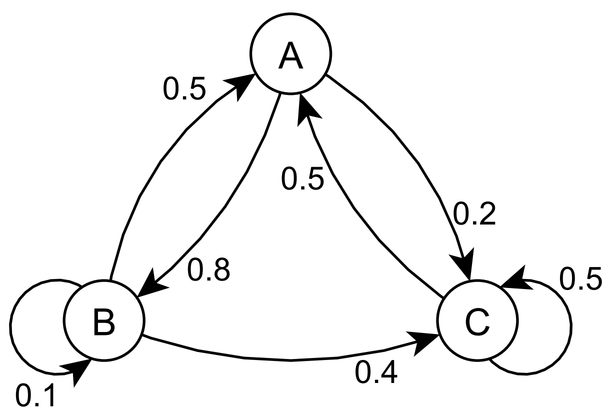
Další možností je využití *Markovových řetězců* pro modelování přirozeného jazyka na úrovni písmen [19] nebo slov [23]. Také mohou být využity pro modelování formálních jazyků [16]. I přesto, že je většina řešení zaměřena především na problém rozpoznávání řeči, mohou být stejné modely použity

pro generování náhodných textů či řetězců. Několik příkladů takových generátorů lze snadno najít na internetu[1].

Generátory založené na Markovových řetězcích obvykle využívají *korpus*, který slouží k určení pravděpodobnosti výskytu písmene následujícího jiné písmeno nebo sekvenci písmen, která jej předchází [4]. Počet písmen, podle kterých je určována pravděpodobnost výskytu dalšího písmene, je označován jako *hloubka* generování.

Nejjednodušším příkladem je generování s hloubkou 1, při kterém je využita tabulka udávající pravděpodobnost pro kombinaci dvou po sobě jdoucích písmen (např. písmeno „h“ následující „t“ s pravděpodobností 0.0253, „x“ následující „t“ s pravděpodobností 0.0001 atd.).

Generátory pracují podobným způsobem jako konečné automaty (viz kapitola 2.2.1), kde je každý stav určen posledním písmenem vygenerovaného slova a pravděpodobnost přechodu do dalšího stavu (resp. pravděpodobnost dalšího vygenerovaného písmene) je určena z tabulky pravděpodobností pro kombinace písmen (viz obrázek 2.3). Při použití vyšších hloubek je stav určen posledními n písmeny vygenerovaného slova, kde n je hloubka generování. To ale velice rychle vede k větší velikosti pravděpodobnostní tabulky, proto není možné zvyšovat hloubku do nekonečna.



Obrázek 2.3: Graf pro Markovův řetězec s pravděpodobnostmi mezi přechody mezi stavy A, B a C

[1]Například: `http://ben.akrin.com/?p=779`

Pro ukázku významu hloubky byly připraveny příklady, které obsahují vygenerovaná slova z anglického korpusu s využitím různých hloubek:

Hloubka 1 *williine w angusat leseakefry rilfrarean cykly icet st ce drns*

Hloubka 2 *deas corty compappowl eappanite bent drat thead fly me*

Hloubka 3 *lip broken fearly earinter yest cruel seasure inst collow femal*

Hloubka 4 *hange push cause healthy sign housers stop hollow little circle*

Z této ukázky je vidět, že s vyšší hloubkou se začínají generovat skutečná anglická slova, protože tabulka pravděpodobností neobsahuje pouze páry písmen, ale obsahuje skutečná slova nebo části slov, které mají délku podle zvolené hloubky.

2.2.4 Generátory využívající slovníky

Nejvíce používanou metodou pro „generování" náhodných řetězů je využití slovníků, které obvykle mají podobu tabulky. Výběr náhodného řetězce pak může být zcela náhodný nebo podle zadaných pravděpodobností výskytu. Pokud jsou využity pravděpodobnosti výskytu hodnot, potom je princip generování založen na kategorickém rozložení pravděpodobnosti.

Pro kategorické generování hodnot je nejprve nutné, aby součet pravděpodobností výskytu byl vždy roven 1, a proto musí být tyto hodnoty normalizovány. Poté mohou být všechny řetězce s nenulovou pravděpodobností rozloženy na interval $\langle 0, 1)$, jak je vidět z obrázku 2.4. Pro výběr náhodného řetězce je pak nutné vygenerování náhodného čísla právě z intervalu $\langle 0, 1)$, které bude použito pro výběr řetězce, jehož intervalu náleží vygenerované číslo.

Obrázek 2.4: Ukázka rozložení hodnot na intervalu $\langle 0, 1\rangle$

Na obrázku 2.4 je znázorněno rozložení tří hodnot, které mají pravděpodobnosti výskytu: „První“ (0.25), „Druhá“ (0.5) a „Třetí“ (0.25). Hodnoty byly rozloženy na následující intervaly: „První“ $\langle 0, 0.25)$, „Druhá“ $\langle 0.25, 0.75)$ a „Třetí“ $\langle 0.75, 1.0)$. Pokud generátor náhodných čísel vrátí například hodnotu 0.4, potom bude vybrána hodnota „Druhá“.

Jak je vidět z výše uvedeného příkladu, je tento princip „generování“ náhodných řetězců velice jednoduchý a může být použit pro libovolný způsob uložení slovníků, které mohou být uloženy v *hašovací tabulce* [11], *trii* (viz kapitola 2.2.2) nebo *seznamu* [11].

3 Existující nástroje pro generování testovacích dat

Před samotným návrhem a implementací této práce bylo nejprve nutné najít a prozkoumat již existující řešení pro generování náhodných testovacích dat, která by mohla být použita jako inspirace během návrhu nebo využita či rozšířena během implementace.

V současné době existuje mnoho nástrojů umožňujících generování komplexních testovacích dat, ale většinou jsou primárně určeny pro testování webových služeb a umožňují pouze vytváření datových souborů ve formátech `JSON`, `CSV`, `XML`, aj. Tyto soubory je poté nutné načíst, zpracovat a výsledná data poté nastavit do atributů *Java* objektů.

Tato kapitola bude především zaměřena na řešení, která umožňují automatické nebo alespoň poloautomatické generování náhodných *Java* objektů a jejich atributů. Pro každé řešení bude uveden obecný popis funkcionality a možností využití, základní informace o projektu, jeho klady a zápory a případně bude uvedena krátká ukázka. V příloze A je uvedena souhrnná tabulka s obecnými informacemi o jednotlivých řešení.

Předem je důležité uvést, že ani jedno z nalezených řešení plně nevyhovovalo požadavkům, především kvůli jejich nárokům či chybějící funkcionalitě, a proto nebylo využito ani rozšířeno. Avšak mnoho z nich posloužilo jako vhodná inspirace běhen návrhu a implementace.

3.1 DataFactory

Knihovna *DataFactory*[1] umožňuje jednoduché generování testovacích dat a byla primárně vytvořena pro plnění databází vývojových či testovacích prostředí náhodnými daty jako jsou jména, adresy, e-maily, telefonní čísla, texty a data.

Projekt pro *DataFactory* byl založen 4. února 2011 vývojářem Andym Gibsonem na stránkách *GitHubu* pod licencí *LGPL Version 3*. Knihovna je

[1] `https://github.com/andygibson/datafactory`

aktuálně volně dostupná ve verzi 0.8 v podobě *Maven* závislosti, která byla vydána 8. září 2015.

```
<dependency>
    <groupId>org.fluttercode.datafactory</groupId>
    <artifactId>datafactory</artifactId>
    <version>0.8</version>
</dependency>
```

Zdrojový kód 3.1: *Maven* závislost pro knihovnu *DataFactory*

DataFactory nepodporuje generování *Java* objektů a jejich atributů, ale podporuje pouze generování náhodných hodnot, které musí být do atributů nastaveny ručně.

Jak už bylo uvedeno, *DataFactory* umožňuje generování náhodných jmen, adres, slov nebo vět obsahujících náhodná slova. K tomu využívá připravené slovníky, které jsou umístěny v kódu, a proto není možné je snadno změnit. Navíc tyto slovníky jsou dostupné pouze v anglickém jazyce. To ovšem není zásadní problém, protože slovníky mají připravené rozhraní, které je možné implementovat a nastavit generátoru. Přesto je například formát generování adres a e-mailů napevno v kódu generátoru a nelze jej nijak jednoduše změnit.

DataFactory dále umožňuje generování náhodné hodnoty ze seznamu hodnot, náhodných čísel s rovnoměrným rozdělením, náhodných znaků a řetězců a náhodných dat.

```
DataFactory df = new DataFactory();
Person person = new Person();
person.setFirstName(df.getFirstName());
person.setLastName(df.getLastName());
person.setGender(df.getItem(Gender.values()));
person.setBirthDate(df.getDateBetween(
    df.getDate(1916, 1, 1), new Date()));
```

Zdrojový kód 3.2: Ukázka generování objektu osoby s využitím knihovny *DataFactory*

Mezi přední výhody *DataFactory* patří její jednoduchost, požadavek minimálně *Javy* verze 5 a to, že není závislá na jiných knihovnách. Na druhou stranu je velikou nevýhodou chybějící možnost generování *Java* objektů a jejich atributů a chybějící podpora vícejazyčnosti nebo snadné úpravy slovníků. Avšak přes tyto nevýhody může být knihovna užitečná v mnoha aplikacích, kde jsou výše uvedené způsoby generování dat dostačující.

3.2 Java Faker

Java Faker[2] vznikl na základě existující knihovny *Faker*[3] pro jazyk *Ruby*, která generuje testovací data. Primárně je určena ke generování ukázkových dat pro vyvíjené aplikace.

Projekt pro *Java Faker* byl založen 6. června 2011 vývojářem Renem Shao na stránkách *GitHubu* pod licencí *Apache License 2.0*. Aktuálně je knihovna volně dostupná v podobě *Maven* závislosti ve verzi 0.8, která byla vydána 13. dubna 2016. Z historie commitů je zřejmé, že *Java Faker* je aktivně vyvíjen, proto lze očekávat podporu v případě chybné či chybějící funkcionality.

```
<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
    <version>0.8</version>
</dependency>
```

Zdrojový kód 3.3: *Maven* závislost pro knihovnu *Java Faker*

Způsob a možnosti generování dat jsou velice podobné jako u knihovny *DataFactory* (viz kapitola 3.1). Stejně jako *DataFactory* nepodporuje generování *Java* objektů a jejich atributů, ale podporuje pouze generování náhodných hodnot, které musí být do atributů nastaveny ručně.

Podle [6] *Java Faker* umožňuje generování náhodných jmen, adres, telefonních čísel, e-mailů, barev, dat a časů, aj. Všechny generované hodnoty, kromě dat a časů, jsou dostupné pouze v podobě řetězců.

```
Faker faker = new Faker(new Locale("cs", "CZ"));

Person person = new Person();
person.setFirstName(faker.name().firstName());
person.setLastName(faker.name().lastName());

Date past = faker.date().past(TimeUnit.DAYS, 365 * 1000);
person.setBirthDate(faker.date().between(past, new Date()));
```

Zdrojový kód 3.4: Ukázka generování objektu osoby s využitím knihovny *Java Faker*

[2] https://github.com/DiUS/java-faker
[3] https://github.com/stympy/faker

Pro generování náhodných textových hodnot jsou využity slovníky, které podporují vícejazyčnost, kde každý jazyk má svůj vlastní slovník. Na rozdíl od *DataFactory* jsou tyto slovníky uloženy v `YAML`[4] souborech nacházejících se uvnitř `JAR` souboru knihovny. Aktuálně je podporováno 40 jazyků, ale bohužel český jazyk mezi ně nepatří i přesto, že například slovenský jazyk podporován je. Je zde však možnost vytvoření vlastního slovníku pro podporovaný či nepodporovaný jazyk. Tento slovník (v `YAML` souboru) musí být uložen do kořenové složky `classpath`.

```
cs:
  faker:
...
    name:
      man_first_name: [Adam, Adolf, Albert, Aleš, ...]
      woman_first_name: [Adéla, Agáta, Alena, Amálie, ...]
      man_last_name: [Adamovský, Apetický, Aulický, ...]
      woman_last_name: [Adamovská, Apetická, Aulická, ...]
      prefix: [Bc., DiS., Ing., Mgr., JUDr., MUDr.]
      suffix: [Ph.D.]
...
```

Zdrojový kód 3.5: Ukázka definice českého slovníku pro knihovnu *Java Faker*

Java Faker vyžaduje minimálně *Javu* verze 6 a má dvě závislosti. První z nich je knihovna `snakeyaml` umožňující práci se soubory `YAML`, ve kterých jsou uloženy slovníky. Druhou závislostí je knihovna `commons-lang` poskytující užitečné nástroje pro práci s řetězci, reflexí [8], náhodnými generátory atd.

Vícejazyčnost a možnost úpravy slovníků pro generování textových hodnot patří mezi hlavní výhody *Java Fakeru*. Mezi nevýhody patří chybějící možnost generování objektů a jejich atributů a to, že neumožňuje generování jiných hodnot než těch slovníkových. Přesto může být knihovna *Java Faker* užitečná v mnoha aplikacích, kde jsou výše uvedené způsoby generování dat dostačující. Příkladem může být knihovna *Random Beans* (viz kapitola 3.6), která obsahuje rozšíření využívající právě možnosti *Java Fakeru*.

[4] Snadno čitelný formát pro serializaci strukturovaných dat.

3.3 generatedata

Volně dostupný nástroj *generatedata*[5] s otevřeným zdrojovým kódem slouží ke snadnému generování velkého množství náhodných dat za účelem testování softwaru nebo naplnění databází. Tato náhodná data mohou být generována v různých formátech či jazycích. Stejně jako většina ostatních nástrojů *generatedata* nepodporuje český ani slovenský jazyk.

Tento nástroj má připravené přehledné a interaktivní GUI (viz obrázek 3.1), které umožňuje snadné nastavení množství a formátu generovaných dat včetně cílového formátu pro export. Pokud uživatel tohoto nástroje nechce využívat připravené GUI, je zde možnost generování dat programově přes připravené `REST API`.

Obrázek 3.1: GUI pro nástroj *generatedata*

Podle [10] je nástroj *generatedata* implementovaný v *JavaScriptu*, *PHP* a *MySQL*. Umožňuje generování přibližně 30 druhů dat jako jsou jména, příjmení, adresy, e-maily, data atd. Vygenerovaná data je možno exportovat

[5] `http://www.generatedata.com`

do formátů: `CSV`, `Excel`, `HTML`, `JSON`, `LDIF`, `SQL` a `XML`. Dále umožňuje export do konstrukcí programovacích jazyků: *JavaScript*, *Perl*, *PHP* a *Ruby*.

```
var data = [
  {
    "firstName": "Macon", "lastName": "Figueroa",
    "email": "ut.nisi.a@diamloremauctor.org",
    "birthDate": "2015-04-26"
  },
  {
    "firstName": "Phyllis", "lastName": "Sanford",
    "email": "mollis.non@egetipsum.co.uk",
    "birthDate": "2005-03-22"
  }
];
```

Zdrojový kód 3.6: Ukázka exportu vygenerovaných dat do *JavaScriptu*

Projekt pro *generatedata* byl založen 26. ledna 2012 vývojářem Benjaminem Keenem na stránkách *GitHubu*[6] pod licencí *GNU GPL 3.0*. Dne 21. května 2013 byla uvolněna „první“ verze 3.0.0. Nyní je dostupná verze 3.2.5, která byla vydána 16. dubna 2016. Z historie verzí je zřejmé, že tento nástroj je aktivně vyvíjen mnoha vývojáři.

Tento nástroj je uveden hlavně z důvodu, že má veliký potenciál být rozšířen tak, aby dokázal exportovat vygenerovaná data do konstrukcí jazyka *Java*. Z ukázkového zdrojového kódu 3.7 je zřejmé, že by bylo ovšem nutné použít návrhový vzor *Fluent interface*, tzn. že by musely *settery* atributů vracet instanci aktuálního objektu (zde `Person`) tak, aby bylo možné jejich zřetězení.

```
Person[] data = new Person[] {
  new Person()
    .setFirstName("Macon").setLastName("Figueroa")
    .setEmail("ut.nisi.a@diamloremauctor.org")
    .setBirthDate("2015-04-26"),
  new Person()
    .setFirstName("Phyllis").setLastName("Sanford")
    .setEmail("mollis.non@egetipsum.co.uk")
    .setBirthDate("2015-04-26"),
};
```

Zdrojový kód 3.7: Ukázka možného rozšíření exportu vygenerovaných dat pro objektový jazyk *Java*

[6] `https://github.com/benkeen/generatedata`

Generování realistických dat nástrojem *generatedata* je dále možné integrovat do *Javy* (nebo jiného programovacího jazyka) využitím připraveného `REST API`. Tímto způsobem by mohla být některá z existujících knihoven rozšířena o funkce generování, které nabízí právě *generatedata*. Další možností je implementace zcela nové knihovny.

Nástroj *generatedata* má spoustu výhod a možností využití v praxi především díky přehlednému `GUI` a připravenému `REST API`, ale pro využití v uživatelských aplikacích je nutná jeho instalace a konfigurace na vlastním webovém serveru. Nebo je možné využívat veřejně dostupnou verzi[7], která v základu umožňuje generování pouze 100 záznamů zároveň. Pro možnost generování libovolných počtů je nutné přispět částkou 20 dolarů.

3.4 jeneratedata

Knihovna *jeneratedata*[8] je pouhým souborem nástrojů pro generování náhodných hodnot. Hlavní motivací pro její vznik byla náhrada online nástroje *generatedata* (viz kapitola 3.3), který slouží ke snadnému generování velkého množství náhodných dat za účelem testování softwaru nebo naplnění databází.

Projekt pro *jeneratedata* byl založen 24. dubna 2010 vývojářem Agustinem Barto *Google Code* pod licencí *Apache License 2.0*. Knihovna je aktuálně dostupná ve verzi 0.2, která byla vydána 25. července 2010. I přesto, že se jedná o *Maven* projekt, knihovna není dostupná v centrálním úložišti *Maven* projektů, ale je dostupná pouze na stránkách projektu. Od vydání poslední verze nedošlo k žádným dalším změnám, a proto můžeme tento projekt považovat za uzavřený.

Podle [2] je základem všech generátorů generické rozhraní `Generator`, které je implementováno generátory náhodných jmen, dat, čísel, textů atd. Některé generátory, např. generátory polí a kolekcí, jsou tvořeny kombinací jednoho či více generátorů. Přístup skládání a využívání připravených generátorů je základem celé knihovny (viz zdrojový kód 3.8). Dále je zde připravené generické rozhraní `Transformer`, které slouží k transformaci dat ze vstupního do cílového datového typu. Transformátory mohou být vhodné pro převod

[7] `http://www.generatedata.com`

[8] `https://code.google.com/archive/p/jeneratedata`

mezi číselnými formáty, hodnot na řetězce a nebo naopak pro převod z řetězců na hodnoty (např. číselné hodnoty, data atd.).

```
public class PersonGenerator implements Generator<Person> {
  Random random = new Random();
  Generator<String> maleNameGen = new MaleNameGenerator();
  Generator<String> femNameGen = new FemaleNameGenerator();
  Generator<String> lastNameGen = new LastNameGenerator();

  public Person generate() {
    Person person = new Person();
    // random gender
    if (random.nextDouble() < 0.5) {
      person.setFirstName(maleNameGen.generate());
      person.setGender(Gender.MALE);
    }
    else {
      person.setFirstName(femNameGen.generate());
      person.setGender(Gender.FEMALE);
    }
    person.setLastName(lastNameGen.generate());

    return person;
  }
}
```

Zdrojový kód 3.8: Ukázka implementace generátoru osob s využitím knihovny *jeneratedata*

Knihovna *jeneratedata* vyžaduje minimálně *Javu* verze 6 a je závislá pouze na knihovně `commons-lang`, ze které jsou využity pouze nástroje pro generování náhodných řetězců. Její hlavní výhodou je jednoduchost a také velké množství připravených generátorů. Bohužel má ale mnoho nevýhod. Jednou z nich je to, že už není déle vyvíjená. Další nevýhodou je chybějící vícejazyčnost generovaných dat a to, že slovníky pro generování náhodných jmen jsou součástí zdrojového kódu.

3.5 random-data-generator

Knihovna *random-data-generator*[9] je velmi jednoduchá a umožňuje vytváření datových struktur (doménového modelu) s náhodnými daty. Slouží především

[9] `https://code.google.com/archive/p/random-data-generator`

ke generování testovacích dat pro databáze bez nutnosti použití dat uložených v externích souborech (např. `XML` používané v *DBUnit*[10]).

Projekt pro *random-data-generator* byl založen 16. září 2011 vývojářem Ramonem Bucklandem na stránkách *Google Code* pod licencí *Apache License 2.0*. Knihovna je aktuálně dostupná ve verzi 0.01, která byla vydána v den založení projektu, a která je dostupná pouze na stránkách projektu. Od vydání první verze nedošlo k žádným dalším změnám, a proto můžeme tento projekt považovat za uzavřený. Avšak před více jak rokem se objevila na *GitHubu* „druhá" verze[11], která je dostupná pouze pro programovací jazyk *Scala*. Bohužel ani u této verze nedošlo od 4. února 2015 k žádné změně.

Podle [5] by mělo být možné generovat *Java* objekty s atributy naplněnými náhodnými daty. K tomu je možné využít připravené generátory pro křestní jména, příjmení, data atd. Další možností je využití možnosti výběru náhodné hodnoty ze seznamu hodnot. Dále knihovna umožňuje generování „vnořených" objektů, tzn. generování atributů obsahujících reference na jiné objekty. Generování lze konfigurovat pouze programově přiřazováním generátorů k jednotlivých atributům podle jejich jména.

```
RandomDataGenerator rdg = new RandomDataGenerator();
Person person = (Person)rdg.generate(
                  new GenConfig()
                    .name(Name.Firstname, "firstName")
                    .name(Name.Lastname, "lastName")
                    .randomObject(
                      new GenConfig()
                        .randomFromStringList("age",
                            "10, 20, 30, 40, 50, 60, 70")
                      , "myAge"
                      , Age.class
                    )
                  ,Person.class);
```

Zdrojový kód 3.9: Ukázka generování objektu osoby s využitím knihovny *random-data-generator*

Výhodou knihovny je její jednoduchost a to, že vyžaduje minimálně *Javu* verze 5. Hlavní nevýhodou je to, že projekt není udržovaný a dále se nevyvíjí. Další nevýhodou je omezená a nedokončená funkcionalita generování a také

[10]Rozšíření *jUnit* testů, které umožňuje snadné testování databázové vrstvy aplikace.
[11]`https://github.com/rbuckland/random-data-generator`

nutnost programové konfigurace generování, která může přinést problémy v případě, že dojde ke změně názvu některého z atributů.

3.6 Random Beans

Random Beans[12] je knihovna pro generování náhodných *Java* objektů. Hlavní motivací jejího vzniku bylo usnadnění generování náhodných dat, která jsou běžně používána k testování softwaru. Jejich generování se totiž může rychle stát únavným, obzvlášť když doménový model obsahuje příliš mnoho souvisejících tříd.

Podle [9] existuje mnoho případů využití knihovny tam, kde člověk může potřebovat generování náhodných dat během testování, jako jsou například:

- Plnění testovacích databází náhodnými daty.
- Generování souborů s náhodnými daty pro aplikace s dávkovým zpracováním.
- Plnění modelu pro testování zobrazení v MVC webových aplikací.
- Generování náhodných formulářových dat k testování jejich validace.

Projekt pro *Random Beans* byl založen 18. května 2013 vývojářem Mahmoudem Benem Hassinem na stránkách *GitHub* pod licencí *MIT*. Původním názvem byl *jPopulator*, ale 20. prosince 2015 byl změněn na *Random Beans*. Knihovna je aktuálně volně dostupná v podobě *Maven* závislosti ve verzi 2.0.0 vydané 19. února 2016. Nyní je aktivně vyvíjená nová verze 3.0.0, tzn. že lze očekávat novou funkcionalitu a podporu v případě chybné či chybějící funkcionality.

```
<dependency>
    <groupId>io.github.benas</groupId>
    <artifactId>random-beans</artifactId>
    <version>2.0.0</version>
</dependency>
```

Zdrojový kód 3.10: *Maven* závislost pro knihovnu *Random Beans*

[12] https://github.com/benas/random-beans

Podle [9] *Random Beans* umožňuje generování objektů a jejich atributů na základě anotací, kterými jsou atributy anotovány. V případě, že atribut není anotován, je použit výchozí generátor pro daný typ (pokud je podporován). Pro vyřazení atributu z generování slouží anotace `@Exclude`. Pro určení vlastního generátoru anotovaného atributu slouží anotace `@Randomizer`, která má jediný parametr pro určení implementace generátoru hodnot (generické rozhraní `Randomizer`).

```
class Person {
  @Randomizer(NameRandomizer.class)
  private String name;
  // implicit generator
  private Gender gender;
  @Exclude
  private int age;
}
...
EnhancedRandom enhancedRandom =
  EnhancedRandomBuilder.aNewEnhancedRandomBuilder().build();
Person person = enhancedRandom.nextObject(Person.class);
```

Zdrojový kód 3.11: Ukázka generování objektu osoby s využitím anotací knihovny *RandomBeans*

Dále knihovna umožňuje programovou konfiguraci generování objektů, která může být libovolně kombinovatelná s výše popsanými anotacemi. Pro programovou konfiguraci je využit návrhový vzor *Fluent interface*, tzn. že konfigurační metody vracejí instanci konfigurace tak, že možné jejich zřetězení. Avšak ze zdrojového kódu 3.12 je vidět, že programová konfigurace není příliš efektivní z důvodu nutnosti používat jména atributů. To může přinést problémy v případě, že dojde k přejmenování některého z atributů.

```
EnhancedRandom enhancedRandom =
  EnhancedRandomBuilder.aNewEnhancedRandomBuilder()
    .randomize(FieldDefinitionBuilder.field()
      .named("name").ofType(String.class)
      .inClass(Person.class).get(), new NameRandomizer())
    .exclude(FieldDefinitionBuilder.field()
      .named("age").ofType(Integer.class)
      .inClass(Person.class).get())
    .build();
Person person = enhancedRandom.nextObject(Person.class);
```

Zdrojový kód 3.12: Ukázka generování objektu osoby s využitím programové konfigurace knihovny *RandomBeans*

Pro knihovnu *Random Beans* je připraveno několik rozšíření, která rozšiřují možnosti generování dat nebo integrace knihovny. Jedním z nich je například integrace již zmiňované knihovny *Java Faker* (viz kapitola 3.2).

Jak už bylo v předchozích odstavcích uvedeno, knihovna *Random Beans* může být velice užitečná, protože nabízí mnoho možností využití. Bohužel má veliké nároky, protože aktuálně vyvíjená verze 3.0.0 vyžaduje minimálně *Javu* verze 8, jejíž využívání je teprve na vzestupu [20]. Oproti tomu verze 2.0.0 vyžaduje minimální verzi 7, která už běžně používána je. Předchozí verze (pod názvem *jPopulator*) vyžadovaly pouze minimální verzi 6. Dalším nárokem je to, že mimo požadavky na minimální verzi *Javy* obsahuje také mnoho závislostí na jiné knihovny. Seznam závislostí je například snadno dostupný v informacích o knihovně na centrálním úložišti *Maven* projektů[13]. Pokud ale uvedené nároky nejsou žádnou překážkou, může tato knihovna sloužit jako užitečný nástroj pro vývoj a testování softwaru.

3.7 Feed4JUnit

Knihovna *Feed4JUnit*[14] je, jak už název napovídá, rozšířením knihovny *jUnit* o možnost vytváření parametrických testů, kterým jsou předávána predefinovaná či náhodná testovací data. Toto rozšíření umožňuje:

- Čtení testovacích dat z `CSV` nebo `Excel` souborů.
- Získávání testovacích dat z databází nebo z vlastních datových zdrojů.
- Provádění testů s náhodnými, ale platnými daty, která mohou výrazně vylepšit pokrytí kódu, což může vést k odhalení chyb způsobených speciálními kombinacemi vstupních dat.

Konfigurace parametrických testů je velice jednoduchá a je založena na *Java* anotacích. S jejich využitím, jak už bylo uvedeno, je možné definovat (a konfigurovat) import dat ze souboru, databáze nebo datového zdroje. Generování náhodných dat je možné definovat anotacemi, které odpovídají (až na pár výjimek) anotacím z *API for JavaBean validation*[15].

[13] `http://mvnrepository.com/artifact/io.github.benas/random-beans/2.0.0`
[14] `http://sourceforge.net/projects/feed4junit`
[15] `http://beanvalidation.org/1.1/spec/`

Projekt pro *Feed4JUnit* byl založen 5. května 2010 vývojářem Volkerem Bergmannem na stránkách *Source Forge* pod licencí *GNU GPL 3.0*. Knihovna je aktuálně dostupná ve verzi 1.2.0, která byla vydána 25. září 2014 v podobě *Maven* závislosti. Od té doby nedošlo k žádným změnám, a proto nejspíš můžeme považovat projekt za dále nevyvíjený.

```
<dependency>
    <groupId>org.databene</groupId>
    <artifactId>feed4junit</artifactId>
    <version>1.2.0</version>
</dependency>
```

Zdrojový kód 3.13: *Maven* závislost pro knihovnu *Feed4JUnit*

Podle [3] je pro možnost využívat parametrické testovací metody v rámci *jUnit* testů nutné použít spouštěč testů `Feeder`, který se přiřazuje testovací třídě anotací `@RunWith(Feeder.class)`. Takto anotované testovací třídy mohou mít testovací metody s parametry. Pokud parametrické testovací metody nejsou anotovány žádnými anotacemi, jsou spouštěny s různými kombinacemi parametrů, které určí spouštěč testu.

Pokud mají být parametry načteny ze souboru, je možné anotovat testovací metodu anotací `@Source("source.csv")`. Výše popsaný spouštěč testů načte a zpracuje uvedený soubor a spustí testovací metodu pro každou načtenou řádku. Počet spuštění lze ovlivnit anotací `@InvocationCount`.

```
@RunWith(Feeder.class)
public class LoginTest {
  @Test
  @Source("userlogin.csv")
  @InvocationCount(5)
  public void testLogin(String name, String password) {
    // test something
  }
}
```

Zdrojový kód 3.14: Ukázka parametrického testu s načtenými daty ze souboru

Pro načtení parametrů z databáze je nutné anotovat celou třídu testu anotací `@Database` obsahující konfiguraci databázového připojení včetně unikátního identifikátoru tohoto připojení. Poté je možné libovolnou testovací metodu anotovat anotací `@Source`, ve které je uveden unikátní identifikátor

databázového připojení a `SQL` dotaz pro získání testovacích dat. Stejně jako u čtení dat ze souboru i zde je spuštěna testovací metoda pro každý vrácený řádek pro uvedený dotaz.

```
@RunWith(Feeder.class)
@Database(id = "db", driver = "com.mysql.jdbc.Driver",
  url = "jdbc:mysql://localhost:3306/f4jdb",
  user = "root", password = "secret")
public class DatabaseTest {
  @Test
  @Source(id = "db", selector = "select name from person")
  public void testName(String name) {
    // test person's name
  }

  @Test
  @Source(id = "db", selector = "select * from person")
  public void testPerson(Person person) {
    // test person
  }
}
```

Zdrojový kód 3.15: Ukázka parametrického testu s načtenými daty z databáze

Další možností je využití připraveného datového zdroje (generátoru dat). Nejprve je nutné anotovat třídu testu anotací `@Bean`, ve které se určí unikátní identifikátor zdroje společně s dalšími parametry (třída zdroje, parametry konstruktoru, způsob konstrukce). Poté může být libovolná testovací metoda anotována anotací `@Source`, ve které musí být uveden identifikátor datového zdroje. Testovací metoda je poté spuštěna pro všechna získaná data z uvedeného datového zdroje.

Ve výše uvedených typech parametrických testů může být parametrem libovolný objekt, jak je vidět z ukázkového zdrojového kódu 3.15, do jehož atributů jsou nastavena načtená data ze souboru, databáze či datového zdroje. Tyto objekty by měli mít bezparametrický konstruktor a *gettery* a *settery* pro všechny atributy. Dále tyto objekty mohou obsahovat „vnořené“ objekty, které mohou být vytvořeny a naplněny načtenými daty.

Pro generování náhodných hodnot jednotlivých parametrů je možné využít některé z připravených anotací, které například umožňují určit minimální a maximální hodnotu parametru, výčet akceptovaných hodnot atd. Také je například možné určit anotací `@Generator` třídu generátoru, který bude po-

užit pro generování anotovaného parametru. Díky této možnosti mohou být pro generování parametrů testů použity i další knihovny umožňující automatické či poloautomatické generování *Java* objektů. Zdrojový kód 3.16 obsahuje ukázku generátoru náhodných osob s využitím knihovny *Random Beans* (viz kapitola 3.6).

```
class PersonGenerator extends ThreadSafeGenerator<Person> {
  EnhancedRandom enhancedRandom =
   EnhancedRandomBuilder.aNewEnhancedRandomBuilder().build();

  public Class<Person> getGeneratedType() {
    return Person.class;
  }

  public ProductWrapper<Person>
     generate(ProductWrapper<Person> wrapper) {
    Person person = enhancedRandom.nextObject(Person.class);
    return wrapper.wrap(person);
  }
}

@RunWith(Feeder.class)
public class GeneratorTest {
  @Test
  @InvocationCount(100)
  public void test(@Generator("PersonGenerator") Person p) {
    // test person
  }
}
```

Zdrojový kód 3.16: Ukázka parametrického testu s generátorem osob založeného na knihovně *Random Beans*

Feed4JUnit nabízí velice užitečná rozšíření *jUnit* testů, která přináší zcela odlišný způsob generování náhodných dat oproti ostatním knihovnám. Anotace jsou přímou součástí testů, a proto není nutné jimi „zatěžovat“ jednotlivé datové objekty aplikace. Tím je zcela zajištěno oddělení testů od samotné aplikace. Zároveň umožňuje snadnou přípravu testů pro připravená testovací data v souborech, databázi nebo jiných datových zdrojích. Dále umožňuje přípravu testů pro náhodně generovaná data, která mohou být generována knihovnou podporující automatické či poloautomatické generování *Java* objektů.

Za jedinou nevýhodu můžeme považovat zvolenou licenci *GNU GPL 3.0*, která striktně vynucuje použití stejné či kompatibilní licence v aplikacích

odvozených či využívajících takto licencovaný software. Avšak autor dává možnost žádosti o udělení komerční licence pro aplikace, které využívají či rozšiřují tuto knihovnu a jsou distribuovány s nekompatibilní licencí.

3.8 PoDaM

Knihovna *PoDaM*[16] (*POJO Data Mocker*) je užitečný nástroj pro testování, který slouží ke generování a naplnění `POJO` objektů „falešnými“ daty. Umožňuje naplnění libovolného stromu objektů náhodnými daty na základě anotací. Podporuje generování generických tříd a atributů, kolekcí, polí a základních typů jazyka *Java*.

Projekt pro knihovnu *PoDaM* byl založen 1. května 2011 vývojářem Marcem Tedonem na stránkách *GitHub* pod licencí *MIT*. Aktuálně je dostupná v podobě *Maven* závislosti ve verzi 6.0.2, která byla vydána 27. prosince 2015. Z historie commitů a vydaných verzí je vidět, že knihovna je aktivně vyvíjená, a proto lze očekávat podporu v případě chybné či chybějící funkcionality.

```
<dependency>
    <groupId>uk.co.jemos.podam</groupId>
    <artifactId>podam</artifactId>
    <version>6.0.2.RELEASE</version>
</dependency>
```

Zdrojový kód 3.17: *Maven* závislost pro knihovnu *PoDaM*

Podle [21] mezi hlavní vlastnosti *PoDaMu* patří:

- Automatické prozkoumání zadaných `POJO` objektů a naplnění všech podporovaných atributů náhodnými daty s tím, že chování generování je možné upravit.
- Podpora uživatelských anotací pro poskytování náhodných dat či vyloučení atributů z generování.
- Možnost volby, které konstruktory budou využívány pro konstrukci objektů. Zda budou využívány konstruktory s nejmenším počtem parametrů (tzv. odlehčené `POJO` objekty) nebo konstruktory s největším počtem parametrů.

[16] `https://github.com/mtedone/podam`

- Podpora základních typů jazyka *Java*, výčtových typů, polí, kolekcí (seznamy, sady, mapy).
- Podpora generických typů (generických tříd a atributů).
- Podpora uživatelských továren objektů, které není *PoDaM* schopen vygenerovat a naplnit (např. kvůli chybějícím anotacím).
- Podpora pro dodatečné vykonávání libovolných metod objektu po dokončení jeho generování a plnění.

Jak již bylo uvedeno, *PoDaM* umožňuje vytvoření objektu s použitím konstruktoru s nejmenším či největším počtem parametrů. Zároveň podporuje naplnění již vytvořeného objektu náhodnými daty. Zdrojový kód 3.18 obsahuje ukázku těchto způsobů generování objektů a jejich naplnění náhodnými daty.

```
PodamFactory factory = new PodamFactoryImpl();

// Vytvori objekt osoby konstruktorem s nejmensim poctem
// parametru a pote setterama naplni atributy
Person person = factory.manufacturePojo(Person.class);

// Vytvori objekt osoby konstruktorem s nejvetsim poctem
// parametru a pote setterama naplni atributy
person = factory.manufacturePojoWithFullData(Person.class);

// Naplni atributy jiz vytvoreneho objektu
person = new Person("Dummy Person")
factory.populatePojo(person);
```

Zdrojový kód 3.18: Ukázka generování a naplnění objektu osoby s využitím knihovny *PoDaM*

Dále podle [21] si uživatelé můžou přizpůsobit plnění generovaných objektů následujícími způsoby:

- Definováním globálních strategií.
- Definováním specifické strategie pouze pro atribut.
- Omezením číselných hodnot (určení minima, maxima, aj.) použitím anotací.

- Přizpůsobení formátu a délky řetězců použitím anotací.
- Vyloučením atributu z plnění použitím anotace nebo podle jeho jména.
- Definováním metod, které budou vykonány po dokončení plnění generovaného objektu.

PoDaM by měl zároveň mít plnou podporu anotací z knihovny *API for JavaBean validation*, které slouží především k určení formátu nebo okrajových podmínek anotovaných atributů a parametrů. Bohužel ale nenabízí žádné připravené generátory realistických náhodných dat jako jsou jména, adresy apod. Avšak díky možnosti definice strategie pro některý atribut je možné využít některé z výše popisovaných knihoven, které to umožňují.

```
class Person {
  @PodamStrategyValue(NameStrategy.class)
  private String name;

  private Gender gender;

  @PodamIntValue(minValue = 0, maxValue = 130)
  private int age;

  public String getName() { return name; }

  public void setName(String n) { name = n; }

  public Gender getGender() { return gender; }

  public void setGender(Gender g) { gender = g; }

  public int getAge() { return age; }

  public void setAge(int a) { age = a; }
}
...
PodamFactory factory = new PodamFactoryImpl();
Person person = factory.manufacturePojo(Person.class);
```

Zdrojový kód 3.19: Ukázka generování objektu osoby s využitím anotací knihovny *PoDaM*

Knihovna *PoDaM* se stala velikou inspirací během návrhu a vývoje této práce, protože nabízí velké množství způsobů a možností generování náhodných dat. Zároveň má i nejlépe zpracovanou dokumentaci s detailními ukázkami funkcionality.

Bohužel její hlavní nevýhodou je její závislost na mnoha knihovnách, mimo jiné i na značné části *Spring frameworku*. To není příliš vhodné pokud chceme využívat připravené anotace pro generování, které jsou pak součástí datových tříd aplikace, protože se celá aplikace stává závislá na všech závislostech *PoDaMu*. Vhodnější by bylo, kdyby knihovna byla rozdělena na `API` (obsahující rozhraní, anotace apod.) a implementaci, protože potom by aplikace mohla být závislá pouze na `API` a testy na implementaci. Avšak sám autor uvádí v dokumentaci, že je možné si projekt stáhnout a případně knihovnu upravit.

4 Návrh knihovny

Hlavním cílem je vytvoření nástroje, který umožní snadné generování testovacích dat na základě definicí objektů (tříd) obohacených o konkrétní anotace. Tímto způsobem je očekávaná struktura dat snadno viditelná přímo ze zdrojových kódů aplikace a není skryta v testech. Navíc použití stejných definic ve všech testech pomáhá testerům, protože se zabraňuje chybám, které často vznikají opakováním stejného kódu.

Nové anotace budou sloužit jako upřesňující popis datového typu každého generovaného atributu. Například místo práce s informací, že atribut je decimální číslo, je možné anotací specifikovat, že hodnota má normální rozdělení se zadanou střední hodnotou a směrodatnou odchylkou. Případně je možné určit minimální a maximální hodnotu (např. výška, váha nebo věk člověka). Dále je možné učit, že hodnota atributu je závislá na ostatních atributech (např. výška člověka je závislá na jeho věku). Stejného přístupu je možné využít nejenom pro čísla, ale také například pro řetězce, u kterých je možné anotacemi určit jejich požadovanou délku, strukturu apod. Příkladem mohou být řetězce pro IP adresy, čísla SPZ, aj. V případě, že atribut obsahuje odkaz na jiný objekt, je možné určit třídu, která bude pro atribut vygenerována.

Je třeba poznamenat, že i přes skutečnost, že tato myšlenka spojení datových objektů s popisem vlastností a struktury dat může být použita v libovolném programovacím jazyce, implementace bude odlišná pro každý jazyk. Obzvlášť proto, že cílem je vytvoření co nejobecnějšího generování dat, a tak je vhodné využití mechanismů, které umožňují analyzovat strukturu generovaných tříd za běhu, namísto analýzy jejich zdrojových kódů. Programovací jazyk *Java* toto snadno umožňuje využitím reflexe [8], ale v jiných jazycích by bylo nutné najít ekvivalentní mechanismus k implementaci tohoto přístupu.

Dále je nutné poznamenat, že tato kapitola se zabývá pouze teoretickým návrhem celé knihovny a neřeší všechny technické aspekty samotné implementace. Z toho důvodu je možné, že se bude implementace lišit od tohoto návrhu.

4.1 Struktura knihovny

Knihovnu můžeme rozdělit do dvou hlavních částí: analyzátor tříd a populátor objektů, který využívá náhodné generátory. Ty by mohly být považovány za třetí část knihovny.

Analyzátor tříd je spíše logickou částí, která je rozdělena do menších částí napříč celou knihovnou. Jeho hlavním úkolem je načtení a zpracování tříd pro generování a vytvoření jejich obecného popisu, který je tvořen anotacemi třídy a všemi jejími vlastnostmi (*properties*). Vlastností rozumíme atribut s metodami pro jeho přístup (viz kapitola 4.2) a všemi jeho anotacemi. Analyzátor dále pracuje se závislostmi mezi jednotlivými třídami, aby bylo možné generovat reference mezi nimi. Vzhledem k tomu, že všechny informace o atributech a jejich anotacích jsou součástí *bytecode* tříd, je možné jejich zpracování prostřednictvím reflexe [8], a proto není nutná přímá analýza zdrojových kódů.

Populátor objektů slouží jako `API`, které umožňuje jednoduché vytváření kolekcí náhodných objektů v závislosti na použitých anotacích, které určují způsob generování jednotlivých atributů. Poskytuje metody `populate` a `populateSingle`, které umožňují vygenerování kolekcí objektů s náhodnými daty. Tyto metody mají parametr pro určení třídy (první parametr), pro kterou bude vygenerován zadaný počet instancí (druhý parametr). V `API` dále existují pro obě metody jejich přetížené verze, které umožňují vygenerování pouze jediné instance zadané třídy. Metoda `populateSingle` poskytne pouze instance dané třídy, které budou mít nastaveny reference na ostatní objekty na `null`. Oproti tomu metoda `populate` umožňuje vygenerování instancí zadané třídy včetně všech referencí, které vytvoří libovolnou grafovou strukturu objektů.

Generátory náhodných dat jsou zodpovědné za generování hodnot v závislosti na pravidlech uvedených v anotacích. Generátory mohou být použity společně s populátory hodnot, které umožňují jejich formátování, transformaci apod.

4.2 Použití knihovny

Jak již bylo uvedeno, knihovna by měla sloužit jako nástroj k zjednodušení testování aplikace. Původně byla navržena pro práci s *JavaBeans* (popř. `POJO` objekty), tj. třídami, které mají veřejné *gettery* a *settery* pro každý atribut, prázdný konstruktor a jsou určeny pro nesení dat v aplikaci. Při generování instancí takových tříd je nutné nastavit všechny atributy s využitím *setterů*, kde každý *setter* by měl nastavovat pouze hodnotu jednoho atributu.

Vzhledem k tomu, že mnoho běžně používaných tříd v aplikacích nemá přesnou formu *JavaBeans*, bylo rozhodnuto, že přítomnost *getterů* a *setterů* pro každý atribut nebude striktně vyžadována. Zároveň nebude vyžadováno, aby tyto *gettery* a *settery* měly veřejný přístup (nemusí být `public`).

Pro změnu přístupu k atributům může být využita anotace `@Access`, kterou může být anotována celá třída nebo jednotlivé atributy. Tato anotace podporuje dva druhy přístupu:

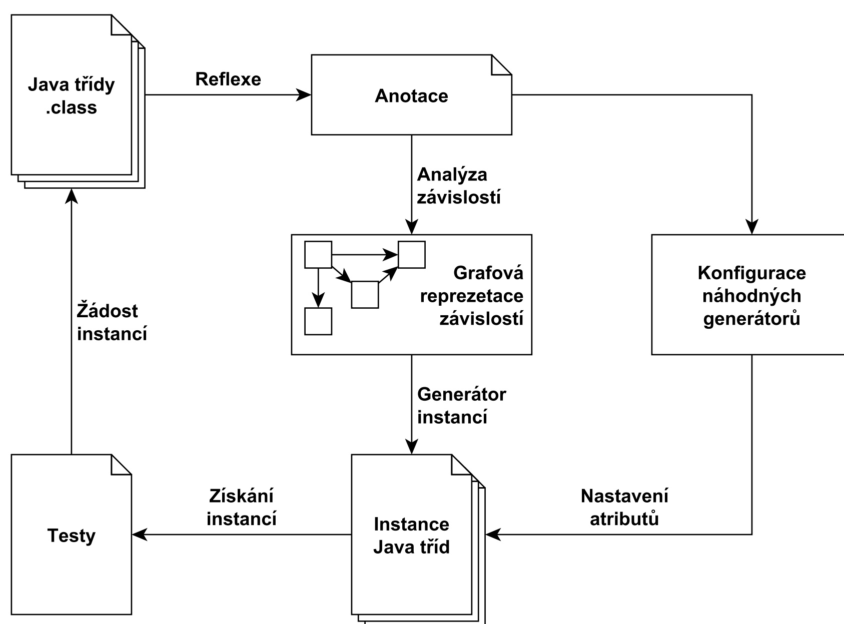
- `AccessType.FIELD` pro přímý přístup k atributu využitím reflexe. Atribut nemusí mít veřejný přístup a v případě potřeby zápisu nesmí být deklarován jako `final`.
- `AccessType.PROPERTY` pro přístup k atributu s využitím *getteru* nebo *setteru*, které nemusí mít veřejný přístup. Toto je výchozí druh přístupu k atributům.

Zároveň budou podporovány třídy s konstruktory, které nemají veřejný přístup nebo které mají více parametrů. Pokud má třída více konstruktorů, je nutné zabránit jejich nejednoznačnosti využitím anotace `@Constructor`. Ta slouží k označení konstruktoru, který bude použit pro vytvoření nové instance třídy. Zároveň může být použita k označení statické (tovární) metody, která bude použita k získání instance dané třídy (např. metoda `getInstance`).

Vzhledem k tomu, že konstruktor nebo tovární metoda může mít více parametrů a zároveň není možné snadno najít vztahy mezi parametry a atributy třídy, je možné anotovat tyto parametry anotacemi pro generování. Pokud některý z parametrů nebude anotován žádnou anotací, potom bude za tento parametr dosazena výchozí hodnota jeho deklarovaného typu.

Proces využití knihovny je znázorněn na obrázku 4.1. Knihovna může pracovat s libovolnými *Java* objekty, ale pokud jejich atributy nemají žádné

anotace, není pro ně schopná vygenerovat náhodná data. Jedinou možností by bylo vygenerovat náhodnou hodnotu z celého definičního prostoru každého atributu, ale tento přístup prozatím nebudeme zvažovat. Takže první věc, kterou testeři nebo programátoři musí udělat, je přidání vhodné anotace každému atributu, který chtějí náhodně generovat. Struktura a příklady konkrétních anotací budou popsány v dalších kapitolách návrhu knihovny. Když jsou jednotlivé objekty a jejich atributy anotovány, je možné začít využívat metody pro vytváření instancí objektů, které mohou být libovolně použity v rámci testů aplikace.



Obrázek 4.1: Základní využití knihovny

4.3 Závislost tříd a jejich atributů

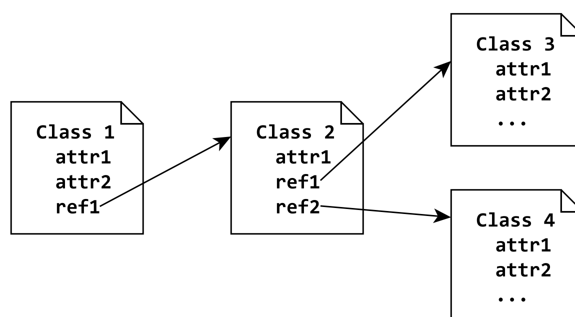
Každý *Java* objekt může obsahovat reference na jiné objekty (závislosti), které je nutné nastavit v případě rekurzivního generování. Zároveň některé atributy mohou být závislé na ostatních atributech, a nebo na atributech závislostí. Generování hodnot pro nezávislé atributy je vcelku jednoduchý úkol, ale práce se závislostmi přináší několik problémů. Je nutné analyzovat strukturu generovaných tříd, aby bylo možné určit pořadí generování nových instancí, nastavení referencí a generování závislých atributů.

4.3.1 Druhy závislostí

Celkem existují dva druhy závislostí, které mohou ovlivnit generování dat. Nejjednodušší z nich je závislost atributu na jednom nebo více dalších atributech ze stejné třídy. Tato situace může nastat i v případě, kdy generování není rekurzivní a všechny reference na objekty jsou nastaveny na `null`. Složitější situace nastává pro závislosti atributů na jiných objektech nebo na jejich atributech. Tato situace může nastat pouze v případě, kdy je generování rekurzivní a jsou generovány atributy obsahující reference na jiné objekty. Potom je během generování nutné zajistit, aby všechny reference na ostatní objekty byly nastaveny tak, aby mohly být všechny závislé atributy vyřešeny.

V současném návrhu může být atribut závislý pouze na atributech ze stejné třídy nebo atributech objektů, které jsou přímo referované z třídy. Atributy nemohou být závislé na volání libovolných metod ze stejné či referované třídy, protože by metody mohly využívat některé atributy, jejichž hodnota ještě nebyla nastavena. O těchto omezení bylo rozhodnuto především proto, aby bylo možné snadno určit pořadí generování atributů pouze na základě jejich deklarací a anotací. V opačném případě by byla nutná analýza zdrojových kódů metod nebo grafu závislostí mezi objekty.

Na obrázku 4.2 je uveden příklad závislostí mezi objekty. Atributy třídy `Class1` mohou být závislé na atributech `attr1` a `attr2` z třídy `Class1` a na atributu `attr1` z třídy `Class2`. Atributy z tříd `Class3` a `Class4` nejsou dostupné.



Obrázek 4.2: Ukázka závislostí mezi objekty a viditelnost atributů

4.3.2 Závislosti v rámci jedné třídy

V případě závislostí v rámci jedné třídy je určení pořadí generovaných atributů velice jednoduché a bude prováděno následujícím způsobem:

1. Vytvoří se množina všech atributů A a prázdná množina atributů N.
2. Naleznou se všechny atributy, které nemají žádné závislosti a vygeneruje se pro ně hodnota. Po vygenerování hodnoty jsou atributy přesunuty z množiny A do množiny N.
3. V množině A se naleznou všechny atributy, které jsou závislé pouze na atributech v množině N. Ty jsou přesunuty z množiny A do množiny N a zároveň je pro ně vygenerována jejich hodnota, protože jsou závislé pouze na atributech z množiny N, tzn. že všechny vyžadované hodnoty již existují.
4. Je opakován krok 3 do té doby, dokud byl některý atribut přesunut z množiny A. Pokud nedojde k přesunu žádného atributu a zároveň množina A není prázdná, potom zbývající atributy obsahují cyklické závislosti, které nemohou být vyřešeny. V takovém případě dojde k vyhození výjimky se zprávou, které atributy a anotace problém způsobily.

4.3.3 Anotace pro závislosti tříd

Pro možnost kontroly generování grafu závislých objektů byly navrženy čtyři základní strategie, kde každá strategie má vlastní anotaci. Každý atribut může být anotován maximálně jednou anotací pro strategii generování. Zároveň některé anotace pro strategie mohou být použity i pro primitivní atributy. V takovém případě mají tyto anotace přednost před ostatními anotacemi pro generování hodnot.

Anotace `@Ignore`

Použitím anotace `@Ignore` je možné zakázat generování anotovaného atributu. V takovém případě bude anotovaný atribut přeskočen a nebude pro něj generována žádná hodnota. Zároveň nedojde ke změně jeho aktuální hodnoty. Tato strategie je výchozí pro všechny atributy, které nemají žádnou anotaci.

```
@Ignore
public Animal animal; // hodnota bude null
@Ignore
public Animal cat = new Cat(); // hodnota nebude změněna
public Animal dog = new Dog(); // hodnota také nebude změněna
```

Zdrojový kód 4.1: Ukázka použití anotace `@Ignore`

Anotace `@NullValue`

Dalším způsobem, jak zakázat generování anotovaných atributů, je použitím anotace `@NullValue`. V takovém případě je hodnota reference nastavena na `null`. Pokud atribut již obsahovat nějakou hodnotu, je tato hodnota nahrazena. Anotace může být použita také pro primitivní typy, pro které budou použity jejich výchozí hodnoty (např. 0 pro číselné hodnoty).

```
@NullValue
public Animal animal; // hodnota bude null
@NullValue
public Animal cat = new Cat(); // hodnota bude null
@NullValue
public int animalsCount = 10; // hodnota bude 0 (výchozí)
```

Zdrojový kód 4.2: Ukázka použití anotace `@NullValue`

Anotace `@NewInstance`

Pro vytvoření nové instance objektu pro anotovaný atribut (referenci na jiný objekt) je možné využít anotaci `@NewInstance`. V takovém případě je vytvořena nová instance třídy, pro kterou byl deklarovaný anotovaný atribut. Možnost vytváření nových instancí je však omezena, protože není možné vytvářet kruhové závislosti mezi objekty.

V případě, že atribut je deklarován rozhraním, abstraktní třídou nebo nadtřídou, musí být poskytnuta třída, pro kterou bude vytvořena nová instance. K tomuto účelu slouží anotace pro poskytovatele tříd (*ClassProvider*). Anotace `@NewInstance` může být použita společně s některou z následujících anotací pro poskytovatele tříd:

- `@TargetClass` – určuje třídu, která bude využita pro vytvoření nové instance.
- `@TargetClassForName` – určuje třídu jejím plně kvalifikovaným jménem, která bude využita pro vytvoření nové instance. Výhodou této anotace je to, že třída `Cat` z balíčku `domain` nemusí být v době kompilace přítomná na `classpath`.
- `@RandomClass` – určuje seznam tříd, které budou podle zadaných pravděpodobností (nepovinný parametr) použity pro vytvoření nové instance. Pro výběr třídy je použit kategorický generátor.
- `@RandomClassForName` – určuje seznam tříd jejich plně kvalifikovaným jménem, které budou podle zadaných pravděpodobností (nepovinný parametr) použity pro vytvoření nové instance. Pro výběr třídy je stejně jako u anotace `@RandomClass` použit kategorický generátor. Zároveň má stejné výhody jako anotace `@TargetClassForName`.
- `@CustomClassProvider` – speciální typ uživatelské anotace, která slouží k určení třídy implementující rozhraní `ClassProvider`, která bude použita pro získání třídy jejíž instance bude vytvořena. Pro více informací o uživatelských anotací viz kapitola 5.2.2.
- Libovolná uživatelská anotace, která musí být anotována pomocnou anotací `@ClassProviderAnnotation`, a která musí sloužit k určení třídy pro vytvoření nové instance. Pro více informací o uživatelských anotací viz kapitola 5.2.2.

Pro vytvoření nové instance bude použit standardní postup, při kterém dojde k nalezení konstruktoru nebo tovární metody anotované anotací `@Constructor` (viz kapitola 4.2). V případě, že nebude nalezen žádný takový konstruktor nebo tovární metoda, třída musí mít jediný konstruktor, protože jinak bude vyhozena výjimka kvůli nejednoznačnosti konstruktorů. Poté bude vytvořena nová instance.

Pokud pro vytvoření nové instance nelze využít výše popsaný standardní postup, je možné určit vlastní způsob konstrukce nové instance využitím anotace `@CustomConstructionStrategy`. Tato anotace může anotovat atribut, pro který bude vytvořena nová instance, nebo může anotovat třídu, pro kterou bude tento uživatelský způsob konstrukce použit. Příkladem využití může být konstrukce objektů s využitím továren, atp.

```
@NewInstance
public Cat cat; // instance třídy Cat

@NewInstance
@RandomClass(value = {Cat.class, Dog.class},
  probabilities = {0.75, 0.25})
public Animal animal; // instance třídy Cat nebo Dog

@NewInstance
@CustomClassProvider(RandomPetClassProvider.class)
@CustomConstructionStrategy(PetConstructionStrategy.class)
public Animal pet; // instance náhodného domázího mazlíčka
```

Zdrojový kód 4.3: Ukázka použití anotace `@NewInstance`

Anotace `@SearchInstance`

V případě, že chceme využít již existující instanci pro atribut obsahující referenci na jiný objekt, je možné využít anotaci `@SearchInstance`. Její využití může vést ke kruhovým závislostem mezi objekty.

Během generování objektů budou ukládány vytvořené instance (pro anotaci `@NewInstance`) do společné kolekce (*session*), takže bude možné některou z již vytvořených instancí nalézt a použít. Jelikož generování závislých objektů může vyžadovat více volání populátoru objektů (viz kapitola 4.1), budou uložené instance ve společné kolekci dostupné mezi jednotlivými voláními. Společnou kolekci instancí bude možné kdykoliv vyprázdnit.

S využitím této anotace budou prozkoumány všechny již vytvořené instance, zda některá z nich nemůže být dosazena do anotovaného atributu, a první vyhovující instance bude použita. Pokud tento přístup vyhledávání instancí není vhodný, je možné anotaci `@SearchInstance` zkombinovat s anotací pro vyhledávání instancí. Vzhledem k tomu, že nebylo možné navrhnout univerzální anotaci pro určení podmínek vyhledávání, jsou dostupné pouze uživatelské anotace:

- `@CustomInstanceMatcher` – speciální typ uživatelské anotace, která slouží k určení třídy implementující rozhraní `InstanceMatcher`, které slouží k určení instance třídy, která vyhovuje kritériím a může být použita pro anotovaný atribut. Pro více informací o uživatelských anotací viz kapitola 5.2.2.

- Libovolná uživatelská anotace jež musí být anotována pomocnou anotací `@InstanceMatcherAnnotation`, a která musí sloužit k určení použitelné instance pro anotovaný atribut. Pro více informací o uživatelských anotací viz kapitola 5.2.2.

```
@SearchInstance
public Animal animal; // náhodná instance zvířete

@SearchInstance
@RandomAnimalClass(cuteness = 0.75) // uživatelská anotace
// náhodná instance zvířete podle jeho roztomilosti
public Animal cuteAnimal;
```

Zdrojový kód 4.4: Ukázka použití anotace `@SearchInstance`

4.3.4 Generování závislých tříd

Je-li použito rekurzivní generování, musí být celý proces generování rozdělen do dvou částí. Nejprve jsou vygenerovány všechny instance tříd, které jsou spojeny referencemi. Poté jsou vygenerovány hodnoty atributů a jsou vyřešeny závislosti mezi nimi. Je možné, že datové objekty obsahují kruhovou závislost, ale na rozdíl od kruhové závislosti mezi atributy, lze tuto závislost snadno vyřešit využitím strategie pro hledání instancí. Algoritmus pro generování závislých tříd bude pracovat následujícím způsobem:

1. Vytvoří se prázdná množina A pro generované atributy, prázdná množina S pro atributy s anotací `@SearchInstance`, fronta vytvořených instancí Q, množina vytvořených instancí I a graf závislostí G.
2. Vytvoří se kopie společné kolekce instancí, kterou označíme jako množinu I_G.
3. Začne se s generovanou třídou, pro kterou je vytvořena nová instance, která je vložena do fronty Q a do množiny I.
4. Vybere se první instance z Q.
5. Vyhledají se všechny atributy pro generování hodnot a vloží se do množiny A.

6. Vyhledají se všechny atributy s anotací `@SearchInstance` a vloží se do množiny S.

7. Vyhledají se všechny atributy s anotací `@NewInstance`, pro které se zkontroluje, zda nevytvoří kruhovou závislost v grafu G. Pokud ano, je jejich hodnota nastavena na `null`. V opačném případě je pro ně vytvořena nová instance, která je vložena do fronty Q a do množiny I. Zároveň je třída přidána jako nový uzel do grafu G.

8. Pokud je fronta Q prázdná, je algoritmus ukončen. V opačném případě se pokračuje na krok 4.

Vzhledem k tomu, že hledání instancí může být prováděno na základě zadaných kritérií, je nutné vygenerovat hodnoty pro atributy, které nejsou závislé. Proto musí být vytvořena prázdná množina atributů N. Poté jsou postupně procházeny množiny A_i pro všechny vytvořené instance z I, a pro atributy, které nemají žádné závislosti, jsou vygenerovány hodnoty. Tyto atributy jsou poté přesunuty z množiny A_i do množiny N_i.

Nyní je možné provést hledání instancí v množinách I a I_G pro všechny atributy z množiny S. Pokud není nalezena vhodná instance pro některý z atributů, je jeho hodnota nastavena na `null`. V opačném případě je atributu nastavena nalezená instance.

Nyní je možné vygenerovat hodnoty pro zbývající atributy z množiny A. Proto je nutné projít množiny A_i pro všechny vytvořené instance z I a vygenerovat hodnoty atributů algoritmem z kapitoly 4.3.2.

4.3.5 Závislost atributů

Jak již bylo uvedeno, hodnoty některých atributů mohou být závislé na ostatních atributech, a nebo na atributech závislostí. V takovém případě může být využita anotace `@Expression`, která může být použita společně s ostatními anotacemi pro generování a osazování hodnot. Anotace obsahuje výraz, který bude vyhodnocen po dokončení generování a osazení všech atributů. Momentálně je zvažována podpora pouze matematických výrazů pro číselné atributy, avšak tato anotace by mohla být použita i pro atributy jiných primitivních typů. Příkladem by mohl být výraz pro spojení dvou řetězců do jednoho, dosazení logické hodnoty v závislosti na uvedených podmínkách apod. Formát

a možnosti výrazů jsou silně závislé na použité implementaci (knihovně) pro jejich zpracování a vyhodnocení.

```
  @Expression("rnd1 * atr1 + ref1.atr1")
  protected int rnd1;
```

Zdrojový kód 4.5: Ukázka použití anotace `@Expression`

Ve zdrojovém kódu 4.5 je uvedena ukázka matematického výrazu pro atribut `rnd1`, který může obsahovat proměnné pro všechny atributy v rámci stejné třídy (`rdn1`, `atr1`) nebo pro atributy závislostí (`ref1.atr1`, kde `ref1` je název atributu obsahujícího závislost a `atr1` je název atributu v třídě závislosti). V matematických výrazech je očekávaná podpora základních aritmetických operací a možnost volání funkcí z třídy `Math`.

Vzhledem k tomu, že je ve výrazech možné použít proměnné pro všechny atributy v rámci jedné třídy, je tedy možné použít je i pro atributy rodičovských tříd. To ovšem může přinést problém s nejednoznačností proměnných, protože je možné v třídách deklarovat atributy s názvy, které jsou již použity v rodičovských třídách. Proto byla navržena anotace `@ExpressionVariable`, která umožňuje určit přesný název proměnné pro daný atribut, a tím zabránit překrytí názvů atributů ve výrazech (viz zdrojový kód 4.6). Určení názvu proměnné pro atribut může být využito pro atributy, které mají příliš dlouhý či nevhodný název. Dalším případem může být využití může být situace, kdy dojde k přejmenování atributu, protože v takovém případě by musely být upraveny všechny výrazy obsahující proměnné pro změněný atribut.

```
class Foo {
  private int bar = 10;
}

class Bar extends Foo {
  @ExpressionVariable("bar2")
  private int bar;

  @Expression("bar + bar2 + foo")
  private int foo;
}
```

Zdrojový kód 4.6: Ukázka použití anotace `@ExpressionVariable`

Pokud by došlo k překrytí názvů atributů a nebyla by využita anotace `@ExpressionVariable`, je možné, že bude vyhozena výjimka při vyhodnocení

výrazu, který obsahuje proměnnou pro takový atribut. Další možností by bylo využití atributu, který je „nejblíže" v hierarchii tříd.

4.4 Generátory a jejich anotace

Generátory náhodných hodnot můžeme rozdělit do tří hlavních skupin:

1. Číselné generátory, které jsou odpovědné za generování celočíselných hodnot nebo desetinných hodnot (čísla s plovoucí čárkou).
2. Textové generátory, které se používají pro vytváření řetězců v souladu s pravidly určujícími délku, jazyk nebo strukturu (formát) řetězce.
3. Ostatní generátory, které jsou zodpovědné za generování různých *Java* objektů se specifickou strukturou jako jsou logické hodnoty, výčtové typy atd.

Každý atribut může být anotován maximálně jednou anotací pro generování hodnot. Pokud atribut není anotován žádnou anotací, bude ignorován stejně jako při použití anotace `@Ignore` (viz kapitola 4.3.3). Anotace generátorů mohou být libovolně kombinovány s anotacemi populátorů atributů, které budou popsány v kapitole 4.5.

4.4.1 Číselné generátory

Java umožňuje generování náhodných čísel, ale obsahuje pouze omezený počet generátorů s různými rozděleními pravděpodobnosti. Aktuálně podporuje pouze diskrétní rovnoměrné rozdělení pro kladné celočíselné hodnoty od 0 do jejich maximální hodnoty (např. `Integer.MAX_VALUE`). Pro desetinné hodnoty podporuje generování pouze na intervalu $\langle 0, 1)$ se spojitým rovnoměrným či normálním (Gaussovým) rozdělením.

Kvůli omezeným možnostem *Javy* bude nutná implementace vlastních generátorů pro další druhy pravděpodobnostních rozdělení, nebo bude nutné využít knihovny, která tyto generátory má již připravené. Jednou z nich je na-

příklad knihovna *Uncommons Maths*[1], která je dostupná pod licencí *Apache License 2.0* a poskytuje mimo jiné sadu různých generátorů náhodných čísel.

Knihovna *Uncommons Maths* obsahuje následující generátory náhodných číselných hodnot:

- Generátor desetinných hodnot normálním (Gaussovo) rozdělením;
- Generátor desetinných hodnot se spojitým rovnoměrným rozdělením;
- Generátor celočíselných hodnot s diskrétním rovnoměrným rozdělením;
- Generátor celočíselných hodnot s binomickým rozdělením;
- Generátor celočíselných hodnot s Poissonovo rozdělením;
- Generátor desetinných hodnot s exponenciálním rozdělením.

V prvotním návrhu knihovny byla zvažována pouze jediná anotace pro všechny číselné generátory, která by měla povinný parametr pro určení pravděpodobnostního rozdělení a povinný parametr pro pole parametrů pro toto rozdělení (viz zdrojový kód 4.7). Dalšími nepovinnými parametry bylo možné omezit generované hodnoty na zadaný interval. Zároveň bylo uvažováno o automatickém ořezávání vygenerovaných hodnot na definiční obor datového typu atributu (např. na hodnotu `byte`). Tento návrh přinášel řadu nevýhod:

1. Pro nově vytvořený číselný generátor by musela být vytvořena nová konstanta ve výčtovým typu pro pravděpodobnostní rozdělení (druhy generátorů).
2. Pro parametry generátorů může být použito pouze jednorozměrné pole, což ztěžovalo například definici parametrů pro kategorický generátor, který vyžaduje dvourozměrné pole parametrů. Každý generátor má rozdílný počet parametrů s různými významy a použití pole parametrů ruší jejich jednoznačnost.
3. Pro podporu ořezávání generovaných hodnot by generátory musely mít znalost o deklaraci atributů a musely by řešit, jak vygenerovanou hodnotu oříznout na cílový datový typ.

[1] `http://maths.uncommons.org`

```
@NumberValue(
  generator = NumberGeneratorType.CATEGORICAL,
  params = {1, 0.2, 2, 0.5, 3, 0.3},
  min = 5,
  max = 15
)
```

Zdrojový kód 4.7: Ukázka původního návrhu anotace pro kategorický generátor

Jako řešení prvního problému byly navrženy samostatné anotace pro každý druh číselného generátoru. Díky tomu byl odstraněn i druhý problém, protože každá anotace obsahuje jednoznačné parametry pro daný druh generátoru. Aby generátory zůstaly co nejjednodušší a generovaly pouze náhodné číselné hodnoty podle zadaných parametrů, vznikly anotace pro populátory atributů 4.5, které mají za úkol vygenerované hodnoty transformovat do požadovaného formátu.

```
@NumberValue(min = 5, max = 15)
@CategoricalGenerator(
  values = {1, 2, 3},
  probabilities = {0.2, 0.5, 0.3}
)
```

Zdrojový kód 4.8: Ukázka finálního návrhu anotace pro kategorický generátor a anotace pro populátor číselných hodnot

4.4.2 Textové generátory

Možnosti generování textových dat (řetězců) byly popsány v kapitole 2.2. V této kapitole budou popsány podporované generátory a jejich anotace.

Generátor využívající Markovovy řetězce

Princip tohoto generátoru byl popsán v kapitole 2.2.3. Anotace pro generátor bude mít parametr `corpus` pro určení cesty k souboru obsahující korpus generátoru (soubor obsahující pravděpodobnostní tabulky). Cesta bude moci být absolutní či relativní. Pokud cesta bude relativní vůči `classpath`, bude možné použít prefix `"classpath:"`. Dalším parametrem bude `depth`, který

bude nepovinný a bude sloužit k určení hloubky generování. Kvůli rostoucí složitosti generování bude podporována maximální hloubka 4. Posledním parametrem bude nepovinný `maxLen`, který bude určovat maximální délku vygenerovaného řetězce. V pravděpodobnostní tabulce je za konec slova obvykle považován speciální znak, ale protože pravděpodobnost ukončení slova je vždy dána předchozími znaky a ne délkou generovaného slova, mohou být generována nepřirozeně dlouhá slova. Proto je lepší omezit maximální délku generovaného řetězce.

```
@MarkovChain(
  corpus = "path/to/corpus",
  depth = 2,
  maxLen = 42
)
```

Zdrojový kód 4.9: Ukázka anotace pro generátor řetězců využívající Markovovy řetězce

Generátor využívající regulární výraz

Princip tohoto generátoru byl popsán v kapitole 2.2.1. Anotace pro generátor bude mít parametr `value` pro určení regulárního výrazu a nepovinný parametr `maxLen` pro určen maximální délky vygenerovaného řetězce. Určení maximální délky může být důležité v případě, kdy regulární výraz je schopen generovat pouze delší řetězce než je možné do atributu uložit. Generování je tak vždy ukončeno při dosažení maximální délky řetězce nebo náhodně při dosažení některého z koncových stavů automatu pro regulární výraz.

```
@RegularExpression(
  value = "[^0-9]*[12]?[0-9]{1,2}[^0-9]*",
  maxLen = 6
)
```

Zdrojový kód 4.10: Ukázka anotace pro generátor řetězců využívající regulární výraz

Generátor využívající slovník

Java umožňuje v anotacích deklarovat parametry jako pole řetězců či číselných hodnot s dynamickou velikostí. Díky tomu je možné pro generování ná-

hodných řetězců připravit generátor založený na slovníku (viz kapitola 2.2.4). Anotace bude mít povinný parametr pro pole řetězců (slovník) a nepovinný parametr pro pole pravděpodobností výskytu hodnot. Aby bylo zjištěno, že součet pravděpodobností bude vždy roven 1, budou hodnoty pravděpodobností normalizovány generátorem.

```
@RandomString(
  value = {"string1", "string2", "string3"},
  probabilities = {0.3, 0.5, 0.2}
)
```

Zdrojový kód 4.11: Ukázka anotace pro generátor řetězců využívající slovník

4.4.3 Ostatní generátory

Kvůli omezení atributů *Java* anotací [12] můžeme mezi ostatní generátory zařadit pouze generátor náhodných logických hodnot, který podle zadané pravděpodobnosti vrátí `true` nebo `false`. Dalším generátorem může být generátor náhodných znaků v určeném rozmezí.

Anotace v *Javě* dále podporují atributy pro `Class`, proto mohou být anotace pro poskytování tříd, které byly popsány v kapitole 4.3.3, použity také jako generátory `Class` hodnot.

Do této skupiny generátorů by mohl být zařazen generátor výčtových typů, který by využíval kategorické rozdělení (viz zdrojový kód 4.12), ale bohužel *Java* neumožňuje deklaraci obecného výčtového typu v parametrech anotací. Kvůli tomu není jednoduché připravit obecnou anotaci pro všechny výčtové typy. Jedinou možností by byla anotace, která by obsahovala parametr pro jména konstant, třídu výčtového typu a pravděpodobnosti jednotlivých konstant. Bohužel tento způsob není moc vhodný, protože v případě změny názvu některé konstanty se tato změna nepropíše do řetězcových názvů v anotaci. Z tohoto důvodu nebyla připravena žádná anotace pro výčtové typy, ale je zde možnost, aby si jí uživatel připravil sám pro daný druh výčtového typu.

```
@Enumeration(
  value = {Gender.MALE, Gender.FEMALE},
  probabilities = {0.45, 0.55}
)
```

Zdrojový kód 4.12: Ukázka navrhované anotace pro výčtové typy

Stejný problém nastává pro anotace pro konstanty objektů, které by pro generování využívaly kategorické rozdělení. Příkladem by mohla být anotace pro generování barev třídy `java.awt.Color` podle konstant, které tato třída obsahuje (viz zdrojový kód 4.13). Bohužel i zde *Java* neumožňuje v parametrech anotací používat objekty, a proto by tyto konstanty musely být referovány pouze podle jejich jména. To přináší stejné problémy jako u návrhu anotace pro výčtové typy.

```
@Object(
  value = {Color.RED, Color.BLACK},
  probabilities =  {0.3, 0.7}
)
```

Zdrojový kód 4.13: Ukázka navrhované anotace pro konstanty objektů

4.5 Populátory a jejich anotace

Populátory hodnot atributů vznikly během návrhu anotací pro číselné generátory (viz kapitola 4.4.1), protože bylo zvažováno automatické ořezávání vygenerovaných hodnot na definiční obor datového typu atributu (např. na hodnotu `byte`). To by ovšem znamenalo, že by generátory musely mít znalost o deklaraci atributů a musely by řešit jak vygenerovanou hodnotu oříznout na cílový datový typ. Kvůli zachování jednoduchosti generátorů vznikly populátory, které jsou primárně určeny k formátování a osazování dat do atributů.

Populátory mají k dispozici všechny informace o anotovaných atributech, proto mohou být využity ke generování komplexnějších hodnot, formátování vygenerovaných hodnot, k převodu hodnot do jiných datových typů apod. Zároveň je jejich hlavním úkolem uložit hodnotu do atributu.

Každý atribut může být anotován libovolným počtem anotací pro populátory. Pokud je anotován větším počtem anotací pro populátory, dojde k jejich zřetězení (viz kapitola 4.5.4).

4.5.1 Číselné hodnoty

Během návrhu číselných generátorů (viz kapitola 4.4.1) byl navržen populátor číselných hodnot, který umožňuje omezení číselných hodnot na zadaný interval. Zároveň umožňuje automatickém ořezávání číselných hodnot na definiční obor datového typu atributu popř. datového typu zadaného jako parametr.

```
@NumberValue(target = int.class)
@UniformGenerator(min = 50, max = 200)
private double height;
```

Zdrojový kód 4.14: Ukázka použití anotace `@NumberValue` pro osazení výšky osoby

4.5.2 Textové hodnoty

Tento populátor slouží k převodu hodnot na řetězce. Libovolnou vygenerovanou hodnotu či hodnotu ze zřetězeného populátoru převede na řetězec, který je oříznut nebo případně rozšířen na zadanou délku. Dále umožňuje převod upraveného řetězce do zadané cílové implementace rozhraní `CharSequence`.

```
@StringValue(length = 13, fill = '0',
    target = String.class)
@DiscreteUniformGenerator(min = 0, max = 999999999)
private CharSequence personalID;
```

Zdrojový kód 4.15: Ukázka použití anotace `@StringValue` pro generování osobních čísel s délkou 13 znaků

4.5.3 Pole a kolekce

Kvůli hojnému využití polí a kolekcí byly navrženy anotace pro jejich generování, které musí být použity v kombinaci s anotacemi pro generování hodnot atributů. Zároveň mohou být kombinovány s ostatními anotacemi pro populátory hodnot.

Pro generování polí může být využita anotace `@ArrayValue`, která obsahuje parametry pro určení minimální, maximální nebo přesné velikosti vy-

generovaného pole. Dále umožňuje nastavení cílového datového typu vytvořeného pole. To je velice užitečné v případě, kdy je pole deklarováno jako pole rozhraní nebo abstraktních tříd (např. `Number`). Použití anotace pro generování polí je znázorněno ve zdrojovém kódu 4.16.

```
@ArrayValue(min = 10, max = 100)
@GaussianGenerator(mean = 10, variance = 0.5)
protected double[] array;
```

Zdrojový kód 4.16: Ukázka použití anotace `@ArrayValue` pro generování pole čísel s velikostí v rozmezí mezi 10 a 100

Anotace pro generování kolekcí je velice podobná anotaci pro generování polí. Pro generování kolekcí může být využita anotace `@CollectionValue`, která má stejné parametry jako anotace pro generování polí. Avšak generování kolekcí přináší problém s generováním sad, u kterých minimální či přesná velikost nemusí být shodná s konečnou velikostí vygenerované sady hodnot, protože sada neumožňuje uchování více shodných hodnot. Je to prevence nekonečné smyčky při generování hodnot, kdy by použitý generátor či osazovač poskytuje pouze omezené množství hodnot. Použití anotace pro generování kolekcí je znázorněno ve zdrojovém kódu 4.17.

```
@CollectionValue(size = 255, target = ArrayList.class)
@GaussianGenerator(mean = 10, variance = 0.5)
protected Collection<Double> collection;
```

Zdrojový kód 4.17: Ukázka použití anotace `@CollectionValue` pro generování kolekce čísel s přesnou velikostí 255

Ve zdrojovém kódu 4.17 si můžeme všimnou parametru `target`, který slouží k ručení cílového typu (implementace) kolekce v případě, že atribut je deklarován rozhraním nebo abstraktní třídou. Pokud tento atribut nebude nastaven, bude v takovém případě použita výchozí hodnota (např. `ArrayList` pro rozhraní `List` nebo `Collection`). Seznam výchozích implementací bude upřesněn během implementace.

4.5.4 Zřetězení populátorů

Pokud je atribut anotován více anotacemi pro populátory hodnot, může dojít k jejich zřetězení, které musí být podporováno implementace daného popu-

látoru. Pořadí jejich vyvolávání by mělo být určeno jejich pořadím ve zdrojovém kódu třídy. Toto ovšem není možné zajistit na 100 %, protože existuje několik různých implementací `JVM`, které toto pořadí můžou ovlivnit. Z toho důvodu byla zavedena pomocná anotace `@PropertyPopulatorsOrder`, která umožňuje určit seznam populátorů, které budou zřetězeny v zadaném pořadí.

```
@GaussianGenerator(mean = 100, variance = 10)
@NumberValue(min = 100)
@ArrayValue(min = 10, max = 100)
@PropertyPopulatorsOrder({
  ArrayValue.class, // osazuje pole
  NumberValue.class // omezuje vygenerované hodnoty
})
protected double[] array;
```

Zdrojový kód 4.18: Ukázka zřetězení anotací populátorů a anotace pro určení jejich pořadí

5 Implementace knihovny

Vývoj knihovny byl rozdělen do dvou fází. Během první fáze byla navržena její struktura, jež byla rozdělena do několika částí (vrstev), pro které byla připravena jednotlivá rozhraní a jejich výjimky. Zároveň došlo k návrhu a přípravě všech anotací. V druhé fázi byly tyto části implementovány, tzn. že byly vytvořeny implementace připravených rozhraní, které zajistily propojení či využití jednotlivých částí (vrstev) knihovny.

Knihovna byla pojmenována *Java Objects Populator* (`JOP`) a byla realizována jako *Maven multi-module project*, který je tvořen čtyřmi moduly, dvěma hlavními a dvěma pomocnými. Prvním hlavním modulem je `jop-api`, který byl připraven v první fázi vývoje. Součástí tohoto modulu jsou všechny anotace, rozhraní, výjimky, abstraktní implementace a nástrojové třídy. Druhý hlavní modul je `jop-impl` a obsahuje samotnou implementaci knihovny. První pomocný modul `jop-tests` obsahuje jednotkové testy pro třídy z hlavních modulů. Druhý pomocný modul `jop-examples` obsahuje moduly s ukázkami využití knihovny.

Knihovna je volně dostupná na stránkách *GitHubu*[1] pod licencí *Apache License 2.0* a je závislá pouze na malém množství knihoven (viz kapitola 5.3), jejichž licence jsou kompatibilní se zvolenou licencí. Zároveň je dostupná pro *Javu* verze 5 a vyšší.

5.1 Struktura knihovny

Jak již bylo uvedeno, knihovna je rozdělena do několika částí (vrstev), a tato kapitola věnována jejich obecnému popisu, který bude obsahovat účel, principy a popis rozhraní dané části (vrstvy). Pro popis struktury knihovny byl také vytvořen `UML` diagram tříd, který je dostupný v příloze B.

[1] `https://github.com/mrfranta/jop`

5.1.1 Obecný popis tříd a jejich atributů

Při návrhu knihovny bylo rozhodnuto, že bude možné určit způsob přístupu k jednotlivým atributům, tzn. že bude možné určit, zda pro jejich čtení a zápis budou využity *gettery* a *settery* nebo přímý přístup k atributu (viz kapitola 4.2).

Proto bylo připraveno generické rozhraní `Property`, které slouží jako obecný popis atributů (vlastností) objektu. Generický parametr slouží k určení deklarovaného typu atributu. Rozhraní pro vlastnosti deklaruje metody pro získání anotací, jména a deklarovaného typu atributu. Dále deklaruje velice důležité metody pro získání *getteru* a *setteru*, které mohou být použity pro čtení a zápis hodnot. Pro atributy s přímým přístupem je určena implementace `DirectAccessProperty` a pro atributy s klasickým přístupem je určena implementace `BasicProperty`.

Pro *gettery* je připraveno generické rozhraní `Getter`, jehož generický parametr slouží k určení deklarovaného typu vlastnosti. Analogicky je pro *settery* připraveno generické rozhraní `Setter`. Tato rozhraní obsahují užitečné pomocné metody, ale nejdůležitějšími jsou `get` pro získání a `set` pro nastavení hodnoty atributu, které přijímají jako parametr instanci vlastníka atributu (pro statické atributy může být `null`). Pro oba druhy vlastností je v těchto metodách použita reflexe [8] pro čtení i zápis hodnot.

Během návrhu výše popsaných vlastností bylo zároveň navrženo rozhraní `Bean`, které slouží jako obecný popis tříd. Obecný popis by měl být vytvářen pouze pro jedinou třídu, tzn. že pokud má třída nějakého rodiče, je nutné pro něj vytvořit samostatný popis. Pro získání popisu rodiče je pak možné využít metodu `getParent`, která může vracet `null` v případě, že třída nemá žádného rodiče, nebo pokud je jejím rodičem knihovní `Object`. Rozhraní `Bean` deklaruje metody pro získání vlastností (`Property`) třídy, kterou popisuje, nebo vlastností z celé hierarchie tříd, tzn. i z rodičovských tříd. Implementací rozhraní `Bean` je třída `JopBean`, která může být vytvořena pouze pro popis již existujících objektů, a která při vytváření popisů jejich vlastností respektuje použití anotace `@Access` (viz kapitola 4.2).

Výše popsaná rozhraní pro obecný popis tříd a jejich atributů (vlastností) jsou využívány ve všech částech knihovny, které pracují s třídami či jejich atributy. Jelikož připravená rozhraní a jejich implementace poskytují všechny důležité informace o třídách a jejich vlastnostech, nemělo by být potřeba v rámci ostatních vrstev využívat reflexi.

5.1.2 Továrny

Jelikož jsou jednotlivé části knihovny definovány rozhraními a zároveň je využíváno vkládání závislostí (viz kapitola 5.3.1), byly navrženy a implementovány továrny, které umožňují snadné vytvoření nové instance vyžadované třídy a zároveň zajišťuje vložení všech závislostí, které třída má.

V knihovně jsou připraveny dva typy továren. Prvním typem je obecná továrna, která pouze umožňuje vytváření instancí. Druhý typ továrny je rozšířením obecné továrny o možnosti svázání anotací s třídami implementací (např. anotaci generátoru hodnot s třídou implementující tento generátor), a proto je pak možné získat instanci svázané třídy pro zadanou anotaci.

Pro popis struktury továren byl vytvořen `UML` diagram tříd, který je dostupný v příloze C.

Obecné továrny

Pro obecné továrny je připraveno rozhraní `Factory`, které má generický parametr pro určení typu tříd, jejíž instance bude továrna vytvářet. Za tento parametr by mělo být zvoleno rozhraní (např. pro generátory hodnot). Rozhraní továrny deklaruje jedinou metodu `createInstance`, která vytvoří instanci zadané třídy, která musí dědit či implementovat třídu zvolenou jako parametr továrny. Pokud během vytváření nové instance nastane chyba, měla by být vyhozena výjimka `FactoryException`.

Továrna by zároveň měla zajistit, že pokud je třída anotována anotací `@Singleton` označující objekt jako jedináčka, potom bude pro každé volání metody `createInstance` vrácena stejná instance této třídy. Proto byla implementována abstraktní továrna `AbstractFactory`, která pro vytváření instancí využívá `Injector` (viz kapitola 5.3.1), který by měl zajistit vytvoření nové instance a vložení všech závislostí. Zároveň by měl zajistit, že v případě jedináčků bude vrácena vždy stejná instance.

Továrny založené na vazbách

Pro tento druh továren je připraveno rozhraní `BindingFactory`, které rozšiřuje generické rozhraní `Factory` o možnost svázání anotací s třídami. Roz-

hraní má generický parametr pro určení typu tříd, jejíž instance bude továrna vytvářet.

Pro vytváření vazeb (rozhraní `Binding`) mezi anotacemi a třídami bylo připraveno generické rozhraní `Binder`, které má generický parametr pro určení typu tříd, které bude možné s anotacemi svázat. Při použití vazače v továrně je jeho generický parametr určen parametrem továrny. Pro vytvoření vazby je možné využít metodu `bind`, která vrací rozhraní generické `BindingBuilder`, které je založeno na vzoru *Fluent interface*, tzn. že lze volání jeho metod zřetězit. Anotace je možné svázat s třídou, instancí či konstruktorem. Zároveň je možné programově či s využitím anotace `@Singleton` označit svázanou třídu za jedináčka. Anotaci není možné svázat s více třídami a v případě pokusu o její vytvoření je vyhozena výjimka `BindingException`. Avšak vazač umožňuje nahrazení existující vazby využitím metody `rebind`. Pro získání existujících vazeb je možné použít metodu `getBinding`, která vrací vazbu pro zadanou anotaci, nebo `getBindings` vracející všechny vytvořené vazby.

Vytvoření vazeb v továrně je možné v připravené metodě `configure`, která přijímá vazač jako parametr. Pro vytvoření a získání instancí svázaných tříd pro dané anotace je připravena metoda `createInstance`. Pokud při konfiguraci továrny nebo vytvoření či získání instance svázané třídy vznikne chyba, je možné vyhodit výjimku `FactoryException`.

Vzhledem k tomu, že rozhraní `BindingFactory` je jednoduché a příliš nevypovídá o způsobu přípravy vazeb, byla připravena jeho abstraktní implementace `AbstractBindingFactory`, která rozšiřuje `AbstractFactory` o možnosti konfigurovat vazby a vytvářet či získávat instance svázaných tříd. Abstraktní továrna má jedinou abstraktní metodu `configure`, v rámci které musí být vytvořeny všechny vazby (viz zdrojový kód 5.1). Poté je možné továrnu bez jakýchkoliv úprav používat. Avšak je velice doporučeno překrytí metody `checkAnnotation`, která umožňuje kontrolu správnosti anotací, pro které jsou vytvářeny vazby.

```
protected final void configure() throws BindingException {
  bind(Ignore.class).to(IgnoreStrategy.class);
  bind(NewInstance.class).to(NewInstanceStrategy.class);
  bind(NullValue.class).to(NullValueStrategy.class);
  ...
}
```

Zdrojový kód 5.1: Ukázka vytvoření vazeb v metodě `configure`

5.1.3 Populátor objektů

Populátor objektů je hlavní `API` celé knihovny jehož metody byly popsány v kapitole 4.1. Pro získání instance (implementace) populátoru je nutné využít statickou třídu `ObjectPopulatorProvider`, která má jedinou metodu `getObjectPopulator`. Důvodem použití této třídy je nutnost zavedení vkládání závislostí (viz kapitola 5.3.1), při kterém dojde k přípravě všech částí knihovny a jejich propojení.

```
ObjectPopulator populator =
    ObjectPopulatorProvider.getObjectPopulator();
List<Person> persons = populator.populate(Person.class, 100);
```

Zdrojový kód 5.2: Ukázka získání populátoru objektů a vygenerování 100 instancí objektu člověka (`Person`)

Princip populátoru je založený na algoritmu popsaném v kapitole 4.3.4, ve kterém je využíván graf závislostí, fronta objektů čekajících na vygenerování hodnot apod. Množina těchto prvků byla nazvána kontextem populátoru (rozhraní `PopulatingContext`). Pro každé volání metod `populate` nebo `populateSingle` by měl být vytvářen zcela nový kontext, který je nezávislý na ostatních kontextech, tzn. že populátor může být používán pro paralelní generování objektů.

Rozhraní kontextu obsahuje metody, které umožňují práci s jeho prvky, ale neposkytuje metody pro přímé získání těchto prvků, aby nebylo možné narušit jejich integritu. Pro prvky kontextu byla připravena rozhraní:

- `PopulatingQueue` – fronta obsahující objekty čekající na vygenerování jejich nezávislých atributů.
- `DependencyGraph` – graf závislostí, který slouží především pro prevenci vzniku kruhových závislostí.
- `PopulatingContextSession` – *session* pro ukládání nebo získávání vytvořených instancí. Měly by v ní být uloženy všechny vytvořené instance v rámci kontextu. Zároveň by měla obsahovat instance z kopie globální *session*, která je vytvářena společně s kontextem.

Kontext by měl metodou `canPopulate` zjišťovat, za je možné pro zadanou třídu vytvořit novou instanci (zda nevytvoří kruhovou závislost). Přetíženou

metodou `populate` by měl umožňovat vložení objektu do fronty. Metodou `getPopulatedInstances` by mělo být možné získat seznam všech vytvořených instancí s tím, že je možné parametrem určit, zda budou zahrnuty i instance z kopie globální *session*.

Kontext populátoru by měl také podporovat vytvoření konstrukčního kontextu (metoda `createConstructionContext`), který je využíván během konstrukce nové instance. Tento kontext je velice důležitý, protože v konstruktoru každého objektu může docházet ke generování jeho atributů či dalších závislostí objektu. K tomu je nutné, aby metoda `getCurrentBean` vracela obecný popis vytvářeného objektu a metoda `getCurrentObject` vracela `null`. Tím se liší od klasického kontextu, protože ten by měl v těchto metodách vracet objekt v čele fronty. Další odlišností by měla být implementace metody `addLazyPopulatingStrategyInvocation`, která by měla umožňovat vložení *lazy* (líných) strategií pro osazení atributů (viz kapitola 5.1.4), tzn. strategií uplatňovaných až po dokončení generování stromu objektů. Konstrukční kontext musí tyto líné strategie vykonávat hned.

Aktuální verze knihovny obsahuje celkem 6 implementací kontextů, které můžeme rozdělit do dvojic tvořených klasickým a konstrukčním kontextem. První dvojicí jsou kontexty používané pro volání metody `populate`. Druhá dvojice je pak používána pro vykonávání *lazy* strategií pro osazení atributů. Třetí dvojice je používána pro volání metody `populateSingle`.

Jelikož populátor objektů musí manipulovat s kontextem způsoby, které nemohou být umožněny ostatním vrstvám, bylo nutné připravit pomocné rozhraní `PopulatingContextHandler`, které tyto operace umožňuje. Jeho implementace by měla být zároveň zodpovědná za přípravu vhodného kontextu podle zvoleného způsobu generování závislostí objektů.

5.1.4 Strategie pro osazení atributů

Strategie pro osazování atributů vznikly pro generování závislostí (viz kapitola 4.3.4), avšak byly rozšířeny o výchozí strategii, která je použita pro atributy anotované anotacemi pro populátory a generátory hodnot.

Pro tyto strategie bylo připraveno rozhraní `PopulatingStrategy`, které má tři metody. První metodou je `isLazyStrategy`, která rozhoduje o tom, zda bude strategie uplatněna až po vygenerování celého stromu objektů. Tato metoda má význam především pro strategii hledání instancí. Další metodou

je `supports`, která rozhoduje o tom, zda může být strategie uplatněna pro deklarovaný typ atributu. Pro uplatnění strategie pro zadaný atribut slouží metoda `applyStrategy`, které musí být předán mimo atribut také kontext populátoru (viz kapitola 5.1.3).

K vytváření instancí strategií je možné využít připravenou továrnu (rozhraní `PopulatingStrategyFactory`), jejíž implementace umožňuje vytváření instancí zadaných tříd implementujících rozhraní `PopulatingStrategy`. Zároveň umožňuje vytvoření instance pro implementaci strategie, která je svázána se zadanou anotací (viz kapitola 5.1.2).

Pro výběr a uplatnění správné strategie pro každý atribut byla připravena „mezivrstva“ (rozhraní `PopulatingStrategyInvoker`), jejíž implementace má tento úkol na starosti. Deklaruje metodu `applyStrategy`, která slouží k výběru a uplatnění správné strategie na základě anotací daného atributu. V případě, že dojde během výběru či uplatnění strategie k nějaké chybě, je vyhozena výjimka `PopulatingStrategyException`. Rozhraní také deklaruje metodu `isAnnotationPresent` vracející informaci, zda je zadaný atribut anotován anotací pro strategii osazení atributů.

Strategie vytváření instancí

Strategie vytváření instancí je nejsložitější strategie. Je uplatňována pro všechny atributy, jež jsou anotované anotací `@NewInstance`, a které nejsou deklarovány jako anotace, pole, výčtový nebo primitivní typ. Při jejím uplatnění je nejprve zjištěno, zda je atribut anotován anotací pro poskytovatele tříd (viz kapitola 5.1.7). Pokud ano, je využita „mezivrstva“ pro výběr a použití vhodného poskytovatele (rozhraní `ClassProviderInvoker`) k získání třídy, jejíž instance bude vytvořena. Pokud ne, je použit deklarovaný typ atributu. Poté je zjištěno, zda vytvořením instance nevznikne kruhová závislost mezi třídami. Pokud ano, je hodnota atributu nastavena na `null`. Pokud ne, je získána anotace `@CustomConstructionStrategy`, podle které se rozhodne jaká strategie (zda výchozí či uživatelská) bude použita pro vytvoření nové instance, které zajišťuje rozhraní `ConstructionStrategyInvoker`. Nově vytvořená instance je vložena do fronty objektů, které čekají na naplnění, a je nastavena atributu.

Pro popis struktury strategie vytváření instancí byl vytvořen `UML` diagram tříd, který je dostupný v příloze D.

Strategie hledání instancí

Strategie hledání instancí je uplatňována pro všechny atributy, jež jsou anotovány anotací `@SearchInstance`, a které nejsou deklarovány jako anotace, pole, výčtový nebo primitivní typ. Strategie hledání instancí je *lazy* (líná), tzn. že je uplatněna až po dokončení generování stromu objektů, a při jejím uplatnění je nejprve zjištěno, zda je atribut anotován anotací pro vyhledávání instancí (viz kapitola 5.1.8). Pokud ano, je využita „mezivrstva“ pro vyhledání vhodné instance (rozhraní `InstanceMatcherInvoker`), která bude nastavena atributu. Pokud ne, je v seznamu již vytvořených instancí nalezena první instance, která může být nastavena. Pokud ani jedna z variant hledání nenajde žádnou vhodnou instanci, je hodnota atributu nastavena na `null`.

Pro popis struktury strategie hledání instancí byl vytvořen `UML` diagram tříd, který je dostupný v příloze E.

Strategie ignorování

Strategie ignorování je uplatňována na všechny atributy s anotací `@Ignore` a nevykonává žádnou činnost, protože hodnota ignorovaného atributu nesmí být změněna. Důvodem její existence je sjednocený přístup k uplatňování jednotlivých strategií, který by musel obsahovat podmínku právě pro tuto strategii.

Strategie vynulování

Strategie vynulování je uplatňována na všechny atributy, které jsou anotovány anotací `@NullValue`, a jejím jediným úkolem je výběr a nastavení výchozí hodnoty danému atributu. Výchozí hodnotou pro objekty je to `null`, pro číselné typy je to 0 a pro logický typ je to `false`.

Výchozí strategie

Výchozí strategie je uplatňována pro všechny atributy, které nemají žádnou anotaci pro generování závislostí. Tato strategie využívá „mezivrstvu“ (rozhraní `PropertyPopulatorInvoker`) pro výběr a vyvolání populátorů hodnot (viz kapitola 5.1.5) pro daný atribut. Pokud atribut není anotován anotací

pro populátor hodnot, je využit výchozí populátor. Ten v případě, že je atribut anotován anotací pro generátor hodnot (viz kapitola 5.1.6), využije *mezivrstvu* pro výběr a vyvolání generátorů a vygenerovanou hodnotu nastaví atributu. V opačném případě zůstane hodnota atributu nezměněna, což je shodný přístup jako při použití strategie ignorování. Důvodem její existence je sjednocený přístup k uplatňování jednotlivých strategií, který by musel obsahovat vyvolávání populátorů hodnot.

Pro popis struktury výchozí strategie byl vytvořen `UML` diagram tříd, který je dostupný v příloze F.

5.1.5 Populátory hodnot

Hlavním úkolem populátorů je nastavování hodnot atributům. Populátory je možné zřetězit a mohou být využity pro generování komplexnějších hodnot, formátování vygenerovaných hodnot, k převodu hodnot do jiných datových typů apod.

Pro populátory hodnot je připraveno rozhraní `PropertyPopulator`, které má generický parametr pro určení typu parametrů (anotace), na základě kterých budou data upravena a nastavena atributu. První deklarovanou metodou rozhraní je `supports`, která rozhoduje o tom, zda může být populátor použit pro deklarovaný typ atributu. Druhou deklarovanou metodou je `populate`, která slouží k naplnění daného atributu podle daných parametrů. Mimo parametry a atribut přijímá vlastníka atributu, který může být využit pro získávání či nastavování hodnot atributu. Metoda `populate` má přístup k obecnému popisu atributu (viz kapitola 5.1.1), ze kterého je možné získat dodatečné informace pro úpravu či nastavení jeho hodnoty.

K vytváření instancí strategií je možné využít připravenou továrnu (rozhraní `PropertyPopulatorFactory`), jejíž implementace umožňuje vytváření instancí zadaných tříd implementujících rozhraní `PropertyPopulator`. Zároveň umožňuje vytvoření instance populátoru hodnot, který je svázán se zadanou anotací (viz kapitola 5.1.2).

Jelikož výběr a vyvolání správného populátoru hodnot se správnými parametry je náročný proces, především kvůli možnosti využití uživatelských anotací (viz kapitola 5.2.2) a zřetězení populátorů, bylo nutné připravit „mezivrstvu“ (rozhraní `PropertyPopulatorInvoker`), která tento proces bude vykonávat. Rozhraní deklaruje metodu `isAnnotationPresent` vracející in-

formaci, zda je zadaný atribut anotován anotací pro strategii osazení atributů.

Další deklarovanou metodou „mezivrstvy“ je `populate`, která by měla vybrat podle anotací daného atributu první populátor pro vyvolání. Pokud atribut není anotován anotací pro populátor hodnot, je využit výchozí populátor. Ten v případě, že je atribut anotován anotací pro generátor hodnot (viz kapitola 5.1.6) využije *mezivrstvu* pro výběr a vyvolání generátorů a vygenerovanou hodnotu nastaví atributu. V opačném případě zůstane hodnota atributu nezměněna, což je shodný přístup jako při použití strategie ignorování (viz kapitola 5.1.4).

Poslední deklarovanou metodou „mezivrstvy“ je `invokeNextPopulator`, která by měla být využívána pro vyvolání zřetězených populátorů. Metodě musí být zadán atribut a cílový datový typ, který může být zřetězeným populátorem nastaven. Implementace této metody vytvoří virtuální atribut, což je speciální implementace rozhraní `Property` (viz kapitola 5.1.1), pro zadaný cílový typ. Virtuálnímu atributu jsou předány všechny anotace zadaného atributu: seřazené anotace pro populátory kromě anotace pro první populátor a všechny ostatní anotace mimo anotace pro pořadí populátorů. Poté je vyvolána metoda `populate` pro vytvořený virtuální atribut a osazená hodnota je vrácena. Metoda `invokeNextPopulator` musí být volána jednotlivými populátory v jejich implementaci metody `populate`.

5.1.6 Generátory hodnot

Generátory obecně slouží ke generování (náhodných) hodnot v závislosti na parametrech, které jsou jim předávány v podobě anotací. Jsou nezávislé na deklaracích atributů, a proto mohou být použity i pro jiné účely, než pro generování hodnot pro atributy.

Pro generátory hodnot je připraveno rozhraní `ValueGenerator`, které má dva generické parametry. První parametr slouží pro určení typu generovaných dat a druhý parametr slouží k určení typu parametrů (anotace), na základě kterých budou data generována. Rozhraní deklaruje pouze dvě metody. Metoda `getValueType` vrací třídu pro typ generovaných dat. Druhá metoda, `getValue`, přijímá parametry a slouží k samotnému generování hodnot. Pokud se během generování vyskytne chyba, je možné vyhodit výjimku `ValueGeneratorException`.

Pro vytváření instancí generátorů je možné využít připravenou továrnu (rozhraní `ValueGeneratorFactory`), jejíž implementace umožňuje vytváření instancí zadaných tříd implementujících rozhraní generátorů. Zároveň umožňuje vytvoření instance pro implementaci generátoru, která je svázána se zadanou anotací (viz kapitola 5.1.2).

Jelikož výběr a vyvolání správného generátoru hodnot se správnými parametry je náročný proces, především kvůli možnosti využití uživatelských anotací (viz kapitola 5.2.2), bylo nutné připravit „mezivrstvu“, která tento proces bude vykonávat. Proto bylo připraveno rozhraní `ValueGeneratorInvoker`, které deklaruje přetíženou metodu `getValue` pro jedno či více vyvolání generátoru hodnot v závislosti na použitých anotacích u atributu, pro který je tato metoda vyvolána. Metoda vrací vygenerovanou hodnotu (popř. pole hodnot) vybraným generátorem. Pokud dojde k chybě během výběru, přípravě či vyvolání generátoru, je možné vyhodit výjimku `ValueGeneratorException`. Rozhraní „mezivrstvy“ také deklaruje metodu `isAnnotationPresent` vracející informaci, zda je zadaný atribut anotován anotací pro generátor hodnot.

Pokud generátory hodnot využívají knihovní náhodný generátor, potom je vhodné, aby využívaly připravenou *session* pro náhodné generátory (viz kapitola 5.1.9).

5.1.7 Poskytovatelé tříd

Hlavním úkolem poskytovatelů tříd je výběr a případné nahrání tříd, které budou využity ve strategii vytváření instancí (viz kapitola 5.1.4). Jejich principy jsou velice podobné generátorům hodnot (viz kapitola 5.1.6).

Pro poskytovatele tříd je připraveno rozhraní `ClassProvider`, které má jediný generický parametr pro určení typu parametrů (anotace), na základě kterých budou poskytovány třídy. Definice rozhraní obsahuje jedinou metodu `get`, jež slouží k poskytnutí třídy na základě zadaných parametrů, a která v případě chyby může vyhodit výjimku `ClassProviderException`.

Pro vytváření instancí poskytovatelů tříd je možné využít připravenou továrnu (rozhraní `ClassProviderFactory`), jejíž implementace umožňuje vytváření instancí zadaných tříd implementujících rozhraní `ClassProvider`. Zároveň umožňuje vytvoření instance pro implementaci poskytovatele tříd, která je svázána se zadanou anotací (viz kapitola 5.1.2).

Stejně jako u generátorů hodnot byla připravena „mezivrstva“, která zajišťuje výběr a vyvolání správného poskytovatele tříd se správnými parametry. `ClassProviderInvoker`, které deklaruje přetíženou metodu `get` pro jedno či více vyvolání poskytovatele tříd v závislosti na použitých anotacích u atributu, pro který je tato metoda vyvolána. Metoda vrací třídu (popř. pole tříd) poskytnutých vybraným poskytovatelem. V případě, že dojde k chybě během výběru, přípravě či vyvolání poskytovatele, je možné vyhodit výjimku `ClassProviderException`. Rozhraní také deklaruje metodu `isAnnotationPresent` vracející informaci, zda je zadaný atribut anotován anotací pro poskytovatele tříd.

Během poskytování tříd mohou být dynamicky nahrávány třídy a k tomu musí být využit `ClassLoader` (nahrávač tříd). V běžných aplikacích není obvykle nutné řešit výběr vhodného nahrávače, ale některé komplexnější aplikace (obvykle *JavaEE*) mohou obsahovat speciální implementace nahrávačů, které musí být použity pro nahrání určitých tříd. Proto byla připravena *session* (rozhraní `ClassLoaderSesssion`), která umožňuje jejich ukládání a získávání podle unikátního jména.

Pokud poskytovatelé tříd využívají knihovní náhodný generátor, potom je vhodné, aby využívaly připravenou *session* pro náhodné generátory (viz kapitola 5.1.9).

5.1.8 Vyhledávání instancí

Jeho hlavním úkolem je výběr vhodné instance objektu, která bude využita ve strategii vyhledávání instancí (viz kapitola 5.1.4). Základem je rozhraní `InstanceMatcher`, které slouží k určení instance třídy, která vyhovuje kritériím a může být použita. Rozhraní má dvě metody. Metoda `supports` slouží k filtraci tříd, jejichž instance mohou být předány metodě `matches`. Ta by měla rozhodnout, zda předaná instance vyhovuje definovaným podmínkám. Pokud se vyskytne v metodě `matches` chyba, je možné vyhodit výjimku `InstanceMatcherException`.

Pro vytváření instancí je možné využít připravenou továrnu (rozhraní `InstanceMatcherFactory`), jejíž implementace umožňuje vytváření instancí zadaných tříd implementujících rozhraní `InstanceMatcher`. Zároveň umožňuje vytvoření instance pro implementaci, která je svázána se zadanou anotací (viz kapitola 5.1.2).

Jelikož vyhledání instancí s využitím správně vybrané implementace rozhraní `InstanceMatcher` a parametrů je náročný proces, především kvůli možnosti využití uživatelských anotací (viz kapitola 5.2.2), bylo nutné připravit „mezivrstvu“, která tento proces bude vykonávat. Proto bylo připraveno rozhraní `InstanceMatcherInvoker`, které deklaruje metodu `match`, která slouží k vyhledání vhodné instance pro zadaný atribut, pro který je tato metoda vyvolána. Vyhledávání probíhá v seznamu všech instancí, které jsou uloženy v předaném kontextu populátoru (viz kapitola 5.1.3). V případě, že dojde během hledání instance k nějaké chybě, je vyhozena výjimka `InstanceMatcherException`. Rozhraní „mezivrstvy“ také deklaruje metodu `isAnnotationPresent` vracející informaci, zda je zadaný atribut anotován anotací pro vyhledávání instancí.

5.1.9 *Session* pro náhodné generátory

Některé části knihovny (generátory dat, poskytovatelé tříd atd.) mohou využívat knihovní náhodný generátor `Random`. Tento generátor by neměl být vytvářen pro každé generování náhodných dat, protože jsou generována podle *seedu*, který je součástí generátoru. Pokud by byla pokaždé vytvářena nová instance tohoto generátoru, mohlo by docházet k narušení rovnoměrnosti generovaných dat. Zároveň by neměl být využívaný sdílený náhodný generátor mezi více třídami. Z toho důvodu by musela každá třída, která náhodný generátor využívá, obsahovat atribut nebo konstantu právě pro náhodný generátor.

Proto byla připravena `session`, jež uchovává náhodné generátory, a pro kterou bylo připraveno rozhraní `RandomGeneratorSession`. Toto rozhraní obsahuje metody umožňující uložení (`setRandomGenerator`) nebo získání (`getRandomGenerator`) náhodných generátorů pro zadané anotace. Pokud pro danou anotaci nebyl uložen žádný generátor, potom by metoda pro jeho získání měla zajistit jeho vytvoření, tzn. že není povoleno vrácení `null`. Rozhraní také obsahuje metodu pro vyčištění *session*.

Výhodou využití *session* je to, že je definována rozhraním, a proto mohou jeho implementace nabízet různé způsoby uchování a získávání náhodných generátorů. Výchozí implementace uchovává náhodné generátory podle typu anotací, ale součástí knihovny je implementace, která uchovává náhodné generátory podle typu anotace a zároveň podle jejích parametrů. Další výhodou

použití je možnost nastavení vlastních generátorů, popř. vlastních *seedů* pro jednotlivé anotace.

5.2 Anotace

Během přípravy `API` (modulu `jop-api`) byly navrženy a připraveny všechny anotace, které je možné používat pro generování hodnot, osazování atributů, vyhledávání instancí atd. Jelikož byly jednotlivé anotace již popsány v předchozích kapitolách, především v kapitole návrhu knihovny (viz kapitola 4), budou v rámci této kapitoly pouze popsány jejich základní principy a postup pro vytvoření či využití uživatelských anotací.

5.2.1 Označení a rozpoznávání typu anotací

Jelikož je nutné rozlišovat typy anotací (např. anotace pro generátory hodnot, populátory hodnot atd.) a *Java* nepodporuje dědičnost anotací [12], bylo nutné navrhnout mechanismus pro rozpoznávání jejich typů. Proto byla použita metoda označování anotací jinými anotacemi. V podstatě to znamená, že jednotlivé anotace jsou anotovány pomocnými anotacemi, které slouží k rozpoznání jejich typů. Během implementace byly připraveny v modulu `jop-api` tyto pomocné anotace:

- `@PopulatingStrategyAnnotation` – slouží k označení anotací pro strategie osazení atributů (viz kapitola 5.1.4).
- `@PropertyPopulatorAnnotation` – slouží k označení anotací pro populátory hodnot (viz kapitola 5.1.5).
- `@ValueGeneratorAnnotation` – slouží k označení anotací pro generátory hodnot (viz kapitola 5.1.6).
- `@ClassProviderAnnotation` – slouží k označení anotací pro poskytovatele tříd (viz kapitola 5.1.7).
- `@InstanceMatcherAnnotation` – slouží k označení anotací pro vyhledávání instancí (viz kapitola 5.1.8).

- `@CustomParameters` – slouží k označení uživatelských parametrů, které mohou být použity společně s uživatelskou anotací `@CustomXXX`, kde `XXX` je název rozhraní, pro který je anotace určena (viz kapitola 5.2.2). Tato anotace má nepovinný parametr pro určení uživatelské anotace, pro kterou jsou určeny. Využití tohoto parametru slouží jako ochrana před nejednoznačností, pro kterou anotaci jsou určeny.
- `@CustomAnnotation` – slouží k označení uživatelských anotací, které jsou často pojmenovány jako `@CustomXXX`, kde `XXX` je název rozhraní, pro který je anotace určena (viz kapitola 5.2.2).

Výše uvedené anotace mohou být použity pouze pro označení anotací, tzn. že je nelze použít pro třídy, atributy apod. Každá anotace může mít jednu a více anotací pro označení jejího typu. Příkladem jsou anotace pro poskytovatele tříd, které jsou označeny jako anotace pro poskytovatele tříd i jako anotace pro generátory hodnot.

Pro usnadnění práce s označenými anotacemi byla připravena v rámci modulu `jop-api` nástrojová třída `AnnotationUtils`. Metody pro práci s označenými anotacemi vycházejí z knihovního rozhraní `AnnotatedElement`, které implementují všechny anotovatelné prvky. První deklarovanou metodou této nástrojové třídy je `isAnnotatedAnnotation`, která vrací hodnotu `true` nebo `false` v závislosti, zda zadaná anotace (první parametr) je anotována zadanou pomocnou anotací (druhý parametr). Další deklarovanou metodou je `isAnnotatedAnnotationPresent`, která vrací hodnotu `true` nebo `false` v závislosti, zda je zadaný anotovatelný prvek anotován alespoň jednou anotací, která je označena zadanou pomocnou anotací. Poslední deklarovanou metodou je přetížená metoda `getAnnotatedAnnotations`, která vrací seznam všech anotací zadaného anotovatelného prvku (první parametr), které jsou označeny zadanou pomocnou anotací (druhý parametr).

5.2.2 Uživatelské anotace

Během návrhu knihovny byla navržena pouze základní množina anotací, které mohou být užitečné v mnoha aplikacích. Bohužel tato množina je velice omezená, a proto se může stát, že programátor či tester bude vyžadovat specifickou anotaci, která není součástí knihovny. Proto byla přidána podpora uživatelských anotací, které mohou být použity pro generátory hodnot, populátory hodnot, poskytovatele tříd atd. Jedinou výjimkou jsou anotace pro

strategie osazení atributů (viz kapitola 5.1.4), které možnost uživatelských anotací nenabízí.

Momentálně existují dva způsoby přípravy uživatelských anotací. Následující návody budou pro jednoduchost věnovány přípravě uživatelských anotací pro generátory hodnot (viz kapitola 5.1.6), ale mohou být podle nich analogicky připraveny i ostatní podporované anotace.

Použití anotací `@CustomXXX`

První možností je využití anotace `@CustomXXX`, kde `XXX` je název rozhraní, pro který je anotace určena. Pro generátory náhodných hodnot byla připravena anotace `@CustomValueGenerator`, která má jediný parametr pro určení třídy implementující generické rozhraní `ValueGenerator`. Generátory obecně vyžadují pro generování hodnot parametry, které jsou předány generující metodě v podobě anotace, kterou je anotován atribut. Typ vyžadovaných parametrů je deklarován v generickém parametru rozhraní. Pokud generátor nevyžaduje žádné parametry, je možné v generickém parametru použít pomocnou anotaci `@EmptyParameters` (viz zdrojový kód 5.3), která označuje prázdné parametry.

```
@CustomValueGenerator(GenderValueGenerator.class)
protected Gender gender;

@Singleton
class GenderValueGenerator implements
          ValueGenerator<Gender, EmptyParameters> {

  public Class<Gender> getValueType() {
    return Gender.class;
  }

  public Gender getValue(EmptyParameters params) {
    if (Math.random() < 0.5) {
      return Gender.MALE;
    }

    return Gender.FEMALE;
  }
}
```

Zdrojový kód 5.3: Ukázka využití uživatelského generátoru hodnot výčtového typu pohlaví, který nevyžaduje parametry

Pokud jsou pro generování hodnot vyžadovány parametry, je možné vytvořit novou anotaci, kterou může být anotován atribut, pro který je generována hodnota (viz zdrojový kód 5.4). Tato nově vytvořená anotace musí být anotována pomocnou anotací `@CustomParameters` (viz kapitola 5.2.1). Implementace rozhraní `ValueGeneratorInvoker` by pak měla zajistit, že anotace bude nalezena a předána generátoru hodnot.

```
@CustomValueGenerator(GenderValueGenerator.class)
@GenderProbability(0.48)
protected Gender gender;

@Target({ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@CustomParameters
@interface GenderProbability {
  public double value() default 0.5;
}

@Singleton
class GenderValueGenerator implements
          ValueGenerator<Gender, GenderProbability> {

  public Class<Gender> getValueType() {
    return Gender.class;
  }

  public Gender getValue(GenderProbability params) {
    if (Math.random() < params.value()) {
      return Gender.MALE;
    }

    return Gender.FEMALE;
  }
}
```

Zdrojový kód 5.4: Ukázka využití uživatelského generátoru hodnot výčtového typu pohlaví, který využívá parametry

Vytvoření nové anotace

Druhou možností je vytvoření zcela nové anotace, která bude svázána s implementací generátoru. Tento proces je složitější a je vhodný pro větší množství uživatelských anotací. V následujících odstavcích bude popsán postup vytvoření uživatelské anotace pro generátor náhodného prvočísla.

Nejprve je nutné navrhnout a vytvořit anotaci pro generátor hodnot, kterou je nutné anotovat pomocnou anotací `@ValueGeneratorAnnotation` (viz kapitola 5.2.1). Pro generátor bude vhodné zvolit jako parametry minimální a maximální hodnotu prvočísla. Ve zdrojovém kódu 5.5 je uvedena výsledná podoba uživatelské anotace.

```
@ValueGeneratorAnnotation
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface PrimeNumber {
  int min() default 2; // first prime number
  int max() default Integer.MAX_VALUE; // last prime number
}
```

Zdrojový kód 5.5: Uživatelská anotace pro generátor náhodných prvočísel

Nyní je možné vytvořit samotný generátor náhodných prvočísel, který musí implementovat rozhraní `ValueGenerator`. Za generické parametry budou zvoleny `Integer` a `PrimeNumber`, protože budou generována celá čísla podle parametrů v navržené anotaci.

```
public class PrimeNumberGenerator implements
    ValueGenerator<Integer, PrimeNumber> {

  public Class<Integer> getValueType() {
    return Integer.class;
  }

  public Integer getValue(PrimeNumber params) {
    return generatePrime(params.min(), params.max());
  }

  public Integer generatePrime(int min, int max) {
    // algoritmus pro generování prvočísel
  }
}
```

Zdrojový kód 5.6: Implementace generátoru náhodných prvočísel

Mezi vytvořenou anotací a implementací generátoru náhodných prvočísel musí být vytvořena vazba. K tomu je nutné rozšířit výchozí implementaci rozhraní `ValueGeneratorFactory` o vytvoření této vazby. K tomu je možné využít připravenou metodu `customConfigure` (viz zdrojový kód 5.7).

```
@Singleton
public class CustomVGF extends ValueGeneratorFactoryImpl {
  protected void customConfigure() throws BindingException {
    bind(PrimeNumber.class).to(PrimeNumberGenerator.class);
  }
}
```

Zdrojový kód 5.7: Rozšíření továrny pro generátory hodnot, ve kterém je vytvořena nová vazba mezi uživatelskou anotací a generátorem

Posledním krokem je nastavení vytvořené třídy ve zdrojovém kódu 5.7 jako implementaci rozhraní `ValueGeneratorFactory`. Pro tento účel je nutné vytvořit konfigurační soubor `ioc.properties` ve zdrojové složce projektu, který musí obsahovat hodnoty uvedené ve zdrojovém kódu 5.8.

```
cz.zcu.kiv.jop.generator.ValueGeneratorFactory=CustomVGF
```

Zdrojový kód 5.8: Nastavení třídy `CustomVGF` jako implementace rozhraní `ValueGeneratorFactory`

5.3 Použité technologie

5.3.1 Vkládání závislostí

Jelikož knihovna byla rozdělena do několika částí, které jsou definovány rozhraními, bylo nutné nalézt způsob jejich propojení. Hlavním cílem byla možnost jednoduchého nahrazování implementací jednotlivých částí, tzn. aby knihovna byla volně rozšiřitelná. Zvolená metoda tedy musela zajistit, že jednotlivé části nebudou přímo závislé na implementacích těch ostatních.

Proto byla využita technika vkládání závislostí (anglicky *Dependeny Injection*), což je implementace techniky obrácené kontroly (anglicky *Inversion of Control*). Ta je založena na principu přenesení odpovědnosti za nalezení, konfiguraci a spojení komponent (částí, modulů, vrstev) aplikace na někoho třetího [14]. V případě použití techniky vkládání závislostí je tím „třetím“ poskytovatel závislostí.

[14, 15] obsahují popis a porovnání existujících řešení pro vkládání závislostí, kterými jsou například: *PicoContainer*[2], *Google Guice*[3], *Spring framework*[4]. Pro JOP byl zvolena knihovna *Google Guice* ve verzi 3.0, na které je závislý modul `jop-impl`, především kvůli její dobré znalosti. Dalším důvodem byla její podpora standardu JSR-330[5], v rámci kterého vznikla velice jednoduchá knihovna *javax.inject*. Ta obsahuje všechny anotace a rozhraní, která jsou využívána pro vkládání závislostí. Na této knihovně je závislý modul `jop-api`.

Nejdůležitější anotací z *javax.inject* je `@Inject`, kterou může být anotován konstruktor, metoda či atribut. Umístění této anotace pak určuje způsob vložení závislostí. Podle [14, 15] mohou být závislosti vloženy během konstrukce objektu nebo až po jeho konstrukci s využitím *setterů* či přímým vložením do atributů objektu.

Další důležitou anotací je `@Singleton`, která již byla použita v kapitole 5.2.2 v ukázkách implementací uživatelských generátorů hodnot. Slouží k označení třídy jako jedináčka, tzn. že by měla být v rámci vkládání závislostí použita pouze jediná instance anotované třídy.

Aby nebyla knihovna přímo závislá na zvolené implementaci vkládání závislostí, bylo připraveno rozhraní *Injector* (poskytovatel závislostí), které deklaruje přetíženou metodu `getInstance` pro získání instance nebo pojmenované instance zadané třídy (může být rozhraní). Další deklarovanou metodou je `injectMembers`, která by měla zajistit vložení závislostí do již vytvořené instance. V případě, že se v rámci těchto metod vyskytne nějaká chyba, může být vyhozena výjimka `InjectorException`.

Podle [15] je vkládání závislostí službou, která musí být před jejím použitím nejprve vytvořena a nastavena. Poté je možné její zavedení, při kterém dojde k přípravě všech komponent a k jejich propojení. Proces zavedení může být proveden v následujících situacích:

- během startu programu;
- na požádání (pokaždé, když je to potřebné);
- až v momentě první potřeby.

[2] http://picocontainer.com
[3] https://github.com/google/guice
[4] http://springframework.org
[5] https://www.jcp.org/en/jsr/detail?id=330

Pro přípravu a získání poskytovatele závislostí bylo připraveno rozhraní `InjectorProvider`, které pro jeho získání deklaruje metodu `get`. Jeho implementací je třída `InjectorManager`, která má za úkol vytvoření instance poskytovatele závislostí, v závislosti na dostupné implementaci, a tuto instanci poskytovat metodou `get`. Instanci třídy `InjectorManager` lze získat pouze zavoláním metody `getInstance`, při kterém tedy dojde k zavedení vkládání závislostí. To znamená, že v knihovně `JOP` je použita třetí varianta zavádění.

Jak již bylo uvedeno, během zavedení vkládání závislostí dojde k přípravě všech komponent a k jejich propojení, které by mělo vytvořit stromovou strukturu. Pokud dojde k vytvoření cyklických závislostí, nemusí být implementace vkládání závislostí schopná tuto strukturu vytvořit. Proto se využívá tzv. vkládání závislostí mimo kontext zavedení, tzn. že některé závislosti jsou vloženy manuálně s využitím poskytovatele závislostí. K tomu je v `JOP` možné opět využít třídu `InjectorManager`, která umožňuje využití poskytovatele závislostí mimo kontext zavedení.

Díky využití techniky vkládání závislostí a použití *Google Guice*, který je konfigurován programově, bylo možné přidat podporu uživatelského rozšiřování knihovny, které je založeno na nahrazování implementací jednotlivých rozhraní. Tento princip byl znázorněn v kapitole 5.2.2, kde byla využitím souboru `ioc.properties` nahrazena výchozí implementace rozhraní `ValueGeneratorFactory` za uživatelsky rozšířenou verzi.

5.3.2 Logování

Všechny činnosti knihovny jsou podrobně logovány, což může sloužit k odhalení případné chyby v implementaci či využití knihovny. K logování bylo využito *commons-logging*[6] (`JCL`) ve verzi 1.2, což je `API`, které podporuje značné množství populárních implementací logování. Jednou z nich je například velice známé `log4j`[7], které může být využito společně s `JOP` v cílové aplikaci. Avšak díky využití logovacího `API` je možné využít libovolnou podporovanou implementaci logování.

Podle [1] `JCL` poskytuje několik úrovní logování: `TRACE` (trasovací zpráva), `DEBUG` (ladící zpráva), `INFO` (informační zpráva), `WARN` (varování), `ERROR` (chy-

[6] `https://commons.apache.org/proper/commons-logging`

[7] `https://logging.apache.org`

bová zpráva) nebo `FATAL` (zpráva pro fatální chybu). Pokud není využita žádná implementace pro `JCL`, jsou vypisovány pouze varování, chyby a fatální chyby do chybové konzole aplikace. V rámci `JOP` jsou využívány pouze ladící a informační zprávy, varování a chybové zprávy.

5.3.3 Testy

Pro otestování správné funkčnosti implementovaných tříd bylo využito jednotkové testování s využitím knihovny *jUnit*[8] ve verzi 4.12. Ta umožňuje jednoduché testování tříd, které nemají žádné závislosti, ale bohužel většina implementovaných tříd je závislá na ostatních částech knihovny, které jsou do nich vkládány (viz kapitola 5.3.1). Jelikož tyto závislosti mají podobu rozhraní, bylo možné využít knihovnu, která umožní jejich *mockování.*

V [15] je uvedena ukázka *mockování* vkládaných závislostí s využitím knihovny *EasyMock*[9], avšak kvůli její lepší znalosti byla využita knihovna *jMock*[10], která také umožňuje snadné propojení s použitou knihovnou *Google Guice* pro vkládání závislostí. Při vytvoření *mocku* rozhraní či (abstraktní) třídy knihovnou *jMock* je vytvořena jejich instance, kterou je možné nastavit jako jejich „implementaci“ pro vložení.

Jelikož knihovna *jMock* vyžaduje *Javu* verze 6, bylo nutné všechny testy přesunout do samostatného pomocného modulu `jop-tests`, který jako jediný vyžaduje tuto verzi *Javy.* Modul pro testy obsahuje speciální zdrojovou složku `src/test-helpers/java`, ve které je připravena podpora *Mockování* závislostí a jejich vkládání do testovaných tříd. Nejdůležitější je abstraktní třída `AbstractContextTest`, která obsahuje integraci *mockování* závislostí a měla by být rozšířena všemi testy tříd se závislostmi. Další důležitou třídou je `MockModule`, v rámci které lze vytvořit *mockované* závislosti pro vložení.

[8] `http://junit.org/junit4`
[10] `http://www.jmock.org`

6 Možnosti rozšíření knihovny

Jelikož je návrh a implementace zcela nové knihovny, technologie či aplikace velice náročný proces, snadno se může stát, že je při jejím návrhu opomenuta nějaká funkcionalita. Chybějící funkcionalitu a ostatní nedostatky často odhalí až uživatelské testy výsledného řešení. Dále se často stává, že je do návrhu zahrnuto příliš mnoho funkcionality, která se poté nestihne implementovat. Také se během samotné implementace může stát, že některá zvolená řešení nejsou příliš vhodná či optimální.

Knihovna `JOP`, která byla navrhnuta a implementována v rámci této práce, není žádnou výjimkou. Proto tato kapitola slouží jako tzv. *TODO* list, tedy seznam funkcionality, která musí být dokončena, nebo která může být později změněna či implementována.

Dokončení podpory závislých atributů

Podpora závislých atributů byla zahrnuta v samotném návrhu knihovny (viz kapitola 4.3.5), avšak nebyla z časových a technologických důvodů implementována, a proto tato funkcionalita není uvedena v kapitole 5. Během návrhu `API` knihovny byly připraveny jednotlivé anotace, rozhraní a výjimky pro jejich podporu. Nyní je nutné pouze nalézt vhodnou knihovnu, která umožňuje řešení matematických či libovolných *Java* výrazů v podobě řetězců. Další možností je vlastní implementace této funkcionality. Poté bude možné implementovat připravené rozhraní a integrovat řešení závislých atributů do procesu generování objektů (viz kapitola 5.1.3) podle algoritmů uvedených v kapitolách 4.3.2 a 4.3.4.

Přidání závislých generátorů hodnot

Během uživatelského testování byl objeven nedostatek návrhu knihovny v podobě chybějící podpory závislých generátorů hodnot, tj. generátorů, které by generovaly hodnoty v závislosti na hodnotách objektu. Příkladem může být jméno a příjmení v závislosti na pohlaví člověka. Nyní je možné připravit pouze generátor, který vygeneruje náhodná jména, ale pak se může stát, že žena bude mít mužské jméno.

Pro generování hodnot závislých na stavu objektu je možné využít populátory hodnot (viz kapitola 5.1.5), ale u nich nebude zajištěno, že během jejich vyvolání bude mít objekt nastaven všechny „závislé“ atributy. Proto by bylo vhodnější návrh a implementace speciálního druhu generátorů, které by byly volány po vygenerování všech atributů objektu (nejlépe po dokončení generování celého grafu objektů). Pro přidání jejich podpory bude nutné navrhnout jejich rozhraní a výjimky, připravit pro ně továrnu a „mezivrstvu“ pro jejich volání. Poté bude nutné jejich volání integrovat do procesu generování objektů (viz kapitola 5.1.3).

Přidání výchozích generátorů

Během návrhu knihovny (viz kapitola 4.2) bylo rozhodnuto, že pro primitivní atributy, které nemají anotaci pro generátor hodnot, nebude generována žádná hodnota. Avšak konkurenční řešení *RandomBeans* (viz kapitola 3.6) a *PoDaM* (viz kapitola 3.8) tuto možnost nabízejí. Proto by bylo vhodné zvážit možnosti generování hodnot pro všechny atributy, které nejsou závislostmi, nejsou vyloučeny z osazení a nemají žádnou anotaci pro generátor hodnot. Tuto podporu by bylo nutné dodat do výchozího populátoru hodnot (viz kapitola 5.1.5). Zároveň by bylo vhodné umožnit svázání jednotlivých datových typů s jejich výchozími generátory. Výchozí generátory by mohly generovat hodnoty pro následující datové typy:

- **Logický typ** – náhodně zvolená hodnota `true` nebo `false`.
- **Číselné typy** – náhodná hodnota na celém jejich definičním oboru.
- **Výčtové typy** – náhodně zvolená konstanta.
- **Řetězce** – náhodně vygenerovaný řetězec z `ASCII` znaků o pevné či variabilní délce.
- **Typy pro datum a čas** – náhodný datum a/nebo čas mezi 1. 1. 1970 (počátek *Unixového času*) a současností.

Atributům, pro jejichž deklarovaný datový typ by nebyl dostupný výchozí generátor, by mohla být nastavena výchozí hodnota daného typu. Další možností by bylo jejich ignorování, tak jako je to prováděno nyní.

Přidání podpory závislostí v polích a kolekcích

Některé objekty mohou mít závislosti uložené v polích či kolekcích a aktuální implementace knihovny tento typ závislostí nepodporuje. Příkladem může být kolekce dětí v třídě rodiče. Proto by bylo vhodné rozšířit strategie osazování atributů (viz kapitola 5.1.4) o možnost vytváření a hledání instancí pro pole či kolekce závislostí. K tomu bude nejspíš nutné připravit nové anotace a implementace pro strategie. Další možností může být přidání speciálních anotací právě pro určení počtu instancí, typu kolekce apod. Přidání této možnosti by nemělo být příliš složité a mělo by být řešeno na úrovni strategií osazení atributů.

Přidání podpory anotací z *API for JavaBean validation*

V dnešní době existuje velké množství aplikací (především webových), které využívají anotace pro validaci hodnot atributů. Kvůli jejich hojnému využití došlo k jejich standardizování v podobě *API for JavaBean validation*[1]. Jelikož konkurenční řešení *RandomBeans* (viz kapitola 3.6) a *PoDaM* (viz kapitola 3.8) již mají podporu těchto anotací, bylo by vhodné ji implementovat také do `JOP`. Jejich podpora by nejspíš musela být integrována v podobě populátoru hodnot (viz kapitola 5.1.5).

Rozšíření možností vyhledávání instancí

Během uživatelského testování byl objeven nedostatek v návrhu hledání vytvořených instancí (viz kapitola 5.1.8), který neumožňuje hledání v závislosti na hodnotách atributů objektu, pro jehož atributy jsou vyhledávány instance. To může být problém například pro hledání objektů dětí pro objekt rodiče, kde by bylo vhodné použít vyhledání podle shody příjmení. Nyní je možné jen vyhledání první vyhovující instance či vyhledání instance, jejíž atributy splňují určené podmínky. Jelikož je vyhledávání instancí prováděno až po vygenerování celého stromu objektů, tzn. že všechny nezávislé atributy objektů mají vygenerované hodnoty, je tedy možné rozšířit metodu `matches` z rozhraní `InstanceMatcher` o parametr pro objekt, pro který je instance vyhledávána. Jedinou nevýhodou tohoto řešení je to, že implementace roz-

[1] `http://beanvalidation.org/1.1/spec`

hraní `InstanceMatcher` budou moct ovlivňovat hodnoty objektu, pro který mají pouze schvalovat instance.

Přidání generátorů realistických dat

Jelikož knihovna plně podporuje generování primitivních i komplexních hodnot, které mohou být nastaveny atributům objektů, je možné ji rozšířit o další připravené generátory. V úvahu přichází generátory realistických hodnot jako jsou jména, příjmení, adresy, e-maily apod. Generování realistických hodnot na základě vícejazyčných slovníků je plně implementováno v knihovně *Java Faker* (viz kapitola 3.2). Proto by bylo možné, stejně jako v konkurenčním řešení *RandomBeans* (viz kapitola 3.6), snadno jeho připravené generátory. To by ale vyžadovalo navýšení vyžadované verze *Javy*. Další možností by byla implementace vlastního řešení, které by vycházelo z implementace *Java Fakeru*.

Přidání podpory generiky

Během návrhu a implementace knihovny byly zcela opomenuty generické typy [13], které se v dnešních aplikacích hojně využívají. To ovšem neznamená, že není možné generování generických typů. Knihovna umožňuje jejich generování, ale pouze v jejich „syrové“ podobě (tzv. *raw type*), tzn. že za všechny generické parametry jsou dosazeny výchozí hodnoty (např. `Object`). Avšak konkurenční řešení *PoDaM* (viz kapitola 3.8) zcela podporuje generování generických typů s možností určení jejich generických parametrů, a proto by bylo vhodné implementovat jejich podporu i do `JOP`. Jako inspirace může sloužit zmiňované řešení z *PoDaMu*, ale odhadově bude muset nejspíš dojít k úpravě obecného popisu tříd a jejich atributů (viz kapitola 5.1.1), které bude nutné doplnit o podporu generických typů.

Přidání hledání v `classpath`

Během implementace byl navržen a implementován princip svazování anotací s třídami (např. anotací a implementací generátorů hodnot) (viz kapitola 5.1.2). Tyto vazby je však nutné vytvářet ručně a v případě, kdy chce uživatel přidat novou vazbu mezi anotací a třídou (např. pro uživatelskou anotaci), je nutné rozšířit či překrýt implementaci továrny, v rámci které

se tyto vazby vytvářejí (viz kapitola 5.2.2). Proto by bylo vhodnější využít techniky vyhledávání v `classpath`, v rámci které by byly nalezeny a spojeny všechny anotace se správnými třídami. Pro tento účel je možné využít knihovnu `reflections`[2], která nabízí velkou škálu metod pro vyhledávání tříd, implementací rozhraní, anotovaných tříd atd. Její hlavní nevýhodou je to, že vyžaduje `Javu` verze 7, což by zbytečně navýšilo požadavky knihovny `JOP`. Proto by bylo vhodnější nalézt hotové či implementovat vlastní řešení, které by bylo zaměřeno pouze na hledání tříd pro anotace.

Přidání programové konfigurace

Pro určení způsobu generování hodnot jednotlivých atributů jsou využity pouze anotace, avšak jejich použití nemusí být vždy žádané či možné. Dále může být potřeba změnit způsob generování některého atributu (např. mimo povolené meze) pouze pro jediný případ, což není možné, protože anotace nelze snadno měnit za běhu aplikace (testů). Další vyžadovanou funkcionalitou by mohla být změna generátorů pouze pro jedno volání populátoru objektů (viz kapitola 5.1.3), nastavení *seedů* generátorů atd. Proto by bylo vhodné implementovat podporu programové konfigurace, kterou nabízí konkurenční řešení *Random Beans* (viz kapitola 3.6).

[2] `https://github.com/ronmamo/reflections`

7 Závěr

V rámci této práce byla navržena a úspěšně implementována knihovna, která je schopna na základě anotací generovat náhodná data pro primitivní i komplexní atributy libovolného *Java* objektu a případně i vytvořit graf objektů, které jsou navzájem propojeny referencemi. Zároveň poskytuje připravené anotace, které lze využít jak k popisu generovaných hodnot, tak k určení způsobu jejich nastavení atributům. Dále je lze využít k určení struktury a obsahu výsledného grafu objektů.

Během návrhu knihovny byly navrženy různé metody generování, které nebyly zvažovány v původním zadání, ale z časových důvodů nebyly dokončeny. Proto v knihovně není dokončená implementace všech navrhovaných generátorů textových dat a podpora závislých atributů, avšak jejich implementace je povětšinou závislá pouze na výběru a integraci vhodné knihovny.

Další generátory a jiná funkcionalita mohou být dodány díky snadné rozšiřitelnosti knihovny a díky podpoře uživatelských anotací. Je nutné zdůraznit, že již nyní je knihovna schopna konkurovat existujícím řešením, která jsou vyvíjena delší dobu a na jejich vývoji se podílí více vývojářů.

Pro demonstraci možností, které knihovna nabízí, byl připraven jednoduchý příklad, jenž je tvořen doménovým modelem s několika třídami. Úkolem bylo vygenerování základní množiny objektů tak, aby mohly být použity pro testování tříd, které s modelem pracují. Řešení této úlohy tedy slouží jako uživatelský test a zároveň prokazuje funkčnost knihovny. Také je možné si na tomto příkladu vyzkoušet odolnost generování proti případným změnám struktury dat. Jelikož jsou zdrojové kódy (včetně *javadoc* dokumentace) součástí pomocného modulu knihovny, nebyl tento příklad popsán v rámci textu této práce.

Všechny body zadání byly splněny a výsledná knihovna může být použita pro generování náhodných dat za účelem testování softwaru nebo naplnění databází. Další možností jejího využití může být generování ukázkových dat pro vyvíjené aplikace. Zároveň otevírá prostor pro její případná rozšíření, která byla podrobně popsána v předchozí kapitole. Aktuální podobu knihovny tedy nepovažuji za výslednou a věřím, že dojde k jejímu dalšímu vývoji.

Seznam zkratek

API Application Programming Interface – programové rozhraní aplikace, které si programátor vytvoří sám, nebo použije již někým vytvořené.

ASCII American Standard Code for Information Interchange – znaková sada, která definuje znaky anglické abecedy a jiné znaky používané v informatice.

CSV Comma-separated values – jednoduchý souborový formát určený k ukládání tabulkových dat.

GUI Graphical user interface – umožňuje ovládání programu interaktivními grafickými prvky.

HTML HyperText Markup Language – značkovací jazyk používaný pro tvorbu webových stránek, který byl vyvinut a standardizován konsorciem W3C.

HTTP Hypertext Transfer Protocol – internetový protokol určený pro výměnu hypertextových dokumentů.

JAR Java Archive – kompresní souborový formát, používaný platformou *Java* k distribuci programů a knihoven.

JSON JavaScript Object Notation – datový formát nezávislý na počítačové platformě, který je určený pro přenos dat.

JVM Java Virtual Machine – virtuální stroj, který slouží ke spuštění počítačových programů vytvořených v jazyce Java.

LDIF LDAP Data Interchange Format – je standardizovaný formát pro textovou reprezentaci záznamů v adresáři (adresářového serveru).

POJO Plain old Java object – obyčejný *Java* objekt, který je tvořen pouze atributy a jejich *gettery* a *settery*.

REST Representational state transfer – způsob jednoduchého vytvoření, čtení, úpravy nebo smazání informací ze serveru s použitím HTTP volání.

SQL Structured Query Language – standardizovaný strukturovaný dotazovací jazyk, který je používán pro práci s daty v relačních databázích.

UML Unified Modeling Language – grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů.

XML Extensible Markup Language – obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C.

YAML YAML Ain't Markup Language – snadno čitelný formát pro serializaci strukturovaných dat.

Literatura

[1] APACHE. Apache Commons Logging – User Guide. [online], 2014 [cit. 5. května 2016]. Dostupné z: https://commons.apache.org/proper/commons-logging/guide.html.

[2] BARTO, A. jeneratedata – Quick Start Guide. [online], 2013 [cit. 26. dubna 2016]. Dostupné z: https://code.google.com/archive/p/jeneratedata/wikis/QuickStartGuide.wiki.

[3] BERGMANN, V. *Feed4JUnit*, 2014 [cit. 27. dubna 2016]. Dostupné z: http://databene.org/feed4junit.html.

[4] BREMAUD, P. *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues.* Springer, 2008. ISBN 978-0387985091.

[5] BUCKLAND, R. random-data-generator. [online], 2011 [cit. 25. dubna 2016]. Dostupné z: https://code.google.com/archive/p/random-data-generator.

[6] DIUS. Java-faker: Brings the popular ruby faker gem to Java. [online], 2016 [cit. 25. dubna 2016]. Dostupné z: http://dius.github.io/java-faker.

[7] DříMAL, J. – TRUNEC, D. – BRABLEC, A. *Úvod do metody Monte Carlo.* [online], Brno, 2006 [cit. 10. května 2016]. Dostupné z: http://www.physics.muni.cz/~trunec/mc.pdf.

[8] FORMAN, I. R. – FORMAN, N. *Java Reflection in Action.* Portland : Manning Publications Co., 2004. 300 s. ISBN 978-1932394184.

[9] HASSINE, M. B. *Random Beans Wiki*, 2016 [cit. 27. dubna 2016]. Dostupné z: https://github.com/benas/random-beans/wiki.

[10] KEEN, B. *Data Generator*, 2016 [cit. 26. dubna 2016]. Dostupné z: http://benkeen.github.io/generatedata.

[11] MAUTNER, P. Programovací techniky. [online], 2016 [cit. 10. května 2016]. Dostupné z: http://www.kiv.zcu.cz/~mautner/Pt/.

[12] ORACLE. *The Java*^{TM} *Tutorials – Lesson: Annotations*, 2015 [cit. 4. května 2016]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/annotations.

[13] ORACLE. *The Java*^{TM} *Tutorials – Lesson: Generics (Updated)*, 2015 [cit. 8. května 2016]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/generics.

[14] PICHLíK, R. Dependency Injection Frameworky v Javě. [online], 2009 [cit. 9. května 2016]. Dostupné z: http://www.slideshare.net/pichlik/dependency-injection-frameworky.

[15] PRASANNA, D. R. *Dependency Injection*. Greenwich : Manning Publications Co., 2009. 330 s. ISBN 978-1933988559.

[16] PROWELL, S. J. TML: A description language for Markov chain usage models. *Information and Software Technology*. 2000, Volume 42, Issue 12, s. 835–844.

[17] RACEK, S. – ROUBíN, M. *Pravděpodobnostní modely počítačů*. Plzeň : ZČU, 1996. 154 s. ISBN 80-7082-300-3.

[18] REIF, J. – KOBEDA, Z. *Úvod do pravděpodobnosti a spolehlivosti*. Plzeň : Západočeská unirverzita, 2000. 111 s. ISBN 80-7082-702-5.

[19] ROSENFELD, R. Two decades of statistical language modeling: where do we go from here? *Proceedings of the IEEE*. 2002, Volume 88, Issue 8, s. 1270–1278.

[20] SALNIKOV-TARNOVSKI, N. Java version statistics: 2015 edition. [online], 2015 [cit. 9. května 2016]. Dostupné z: https://plumbr.eu/blog/java/java-version-statistics-2015-edition.

[21] TEDONE, M. *podam*. Jemos, 2015 [cit. 28. dubna 2016]. Dostupné z: http://mtedone.github.io/podam/.

[22] VAIS, V. Teoretická informatika. [online], 2016 [cit. 10. května 2016]. Dostupné z: http://home.zcu.cz/~vais.

[23] WALLACH, H. M. Topic modeling: beyond bag-of-words. *ICML '06 Proceedings of the 23rd international conference on Machine learning*. 2006, s. 977–984.

A Souhrn informací o nástrojích pro generování *Java* objektů

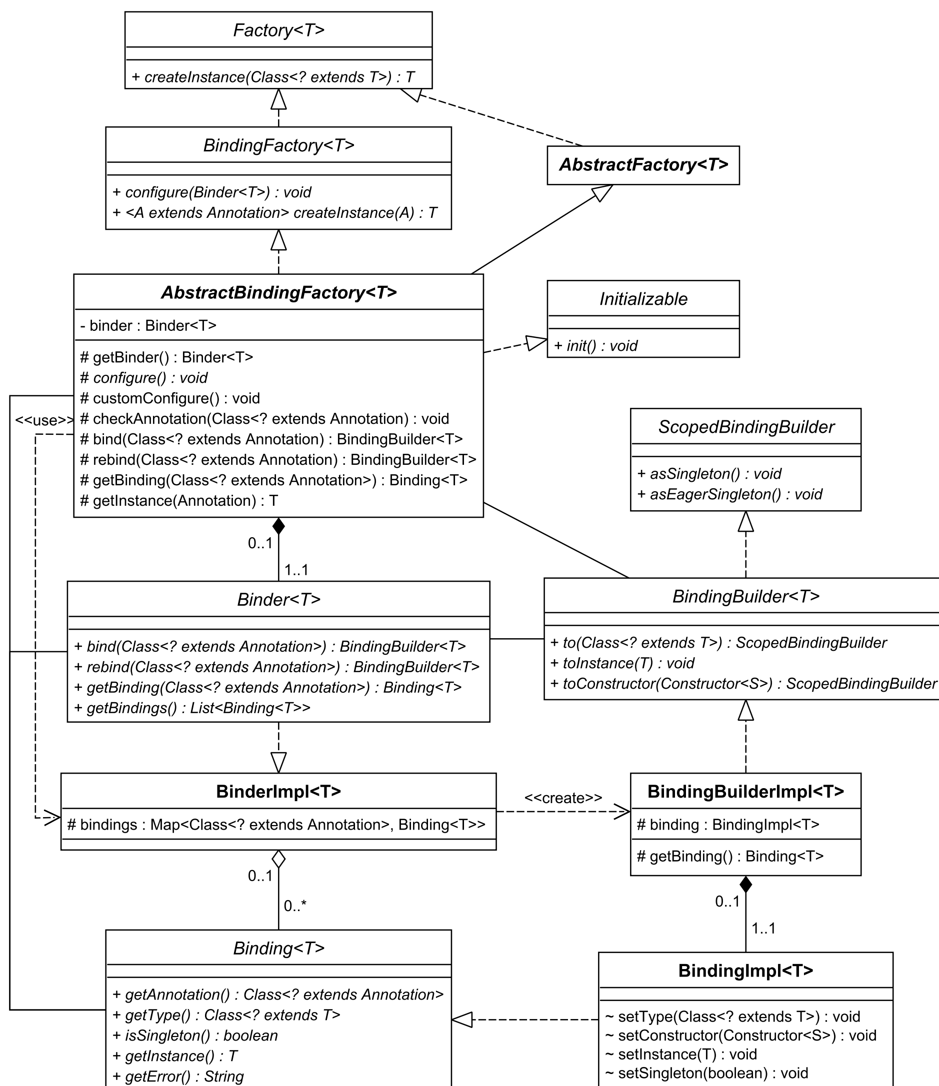
Název	**Verze**	**Datum vydání**	**Licence**	**Požadavky**
DataFactory	0.8	8. září 2015	*LGPL Version 3*	Java 5
Java Faker	0.8	13. dubna 2016	*Apache License 2.0*	Java 6
jeneratedata	0.2	25. července 2010	*Apache License 2.0*	Java 6
random-data-generator	0.01	16. září 2011	*Apache License 2.0*	Java 5
jPopulator	1.2.0	19. března 2015	*MIT*	Java 6
Random Beans	2.0.0	19. února 2016	*MIT*	Java 7
Random Beans	3.0.0	Ve vývoji	*MIT*	Java 8
Feed4JUnit	1.2.0	25. září 2014	*GNU GPL 3.0*	Java 6
PoDaM	6.0.2	27. prosince 2015	*MIT*	Java 6

Tabulka A.1: Souhrn informací o nástrojích pro generování *Java* objektů

B UML diagram tříd znázorňující strukturu knihovny

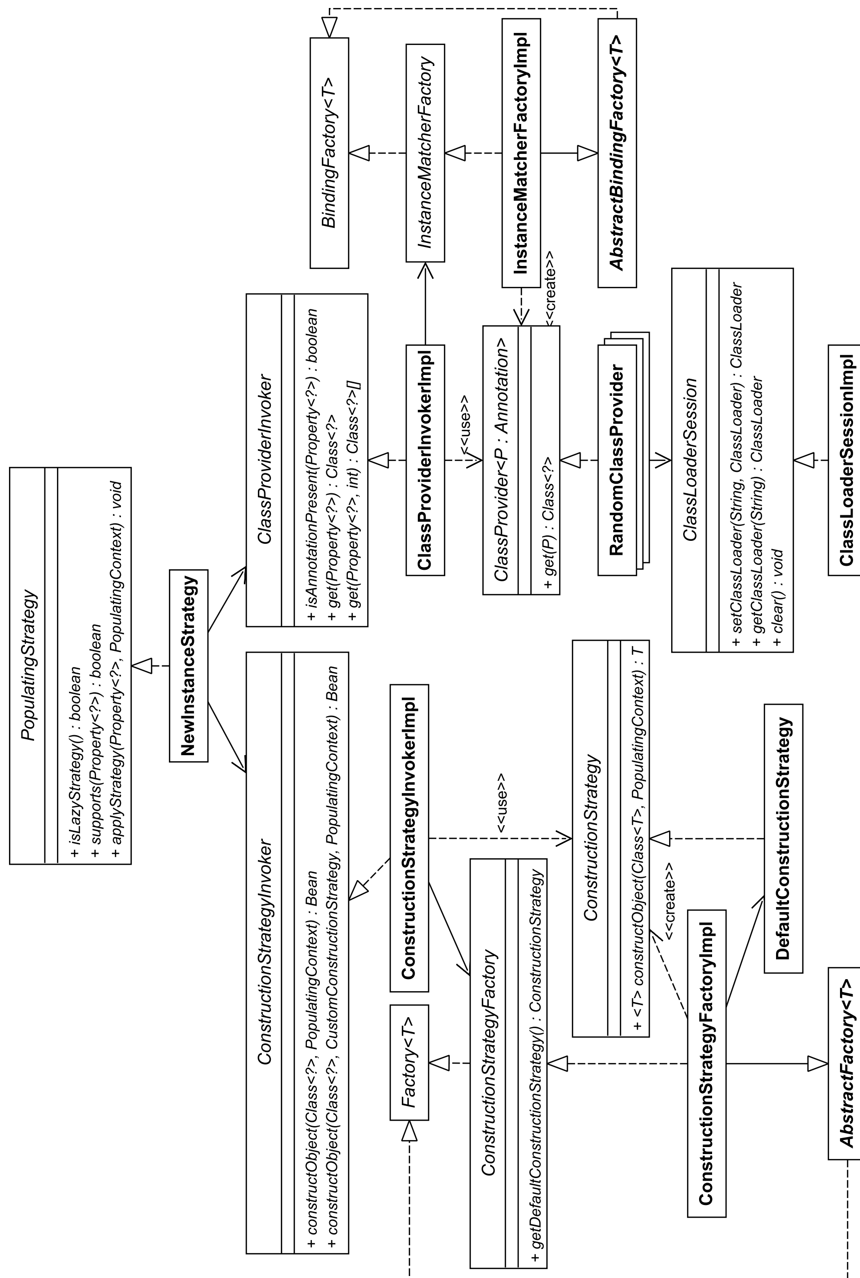
Obrázek B.1: UML diagram tříd znázorňující strukturu knihovny

C UML diagram tříd pro továrny



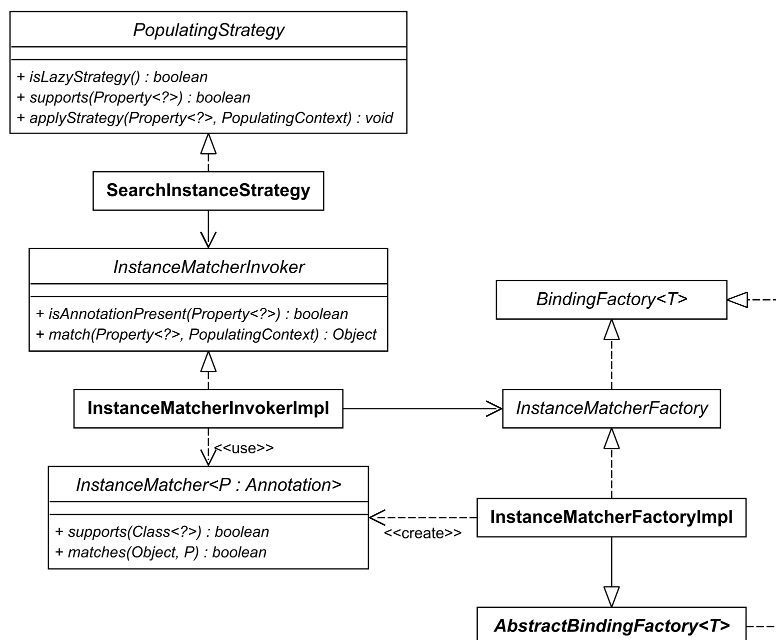
Obrázek C.1: UML diagram tříd pro továrny

D UML diagram tříd pro strategii vytváření instancí



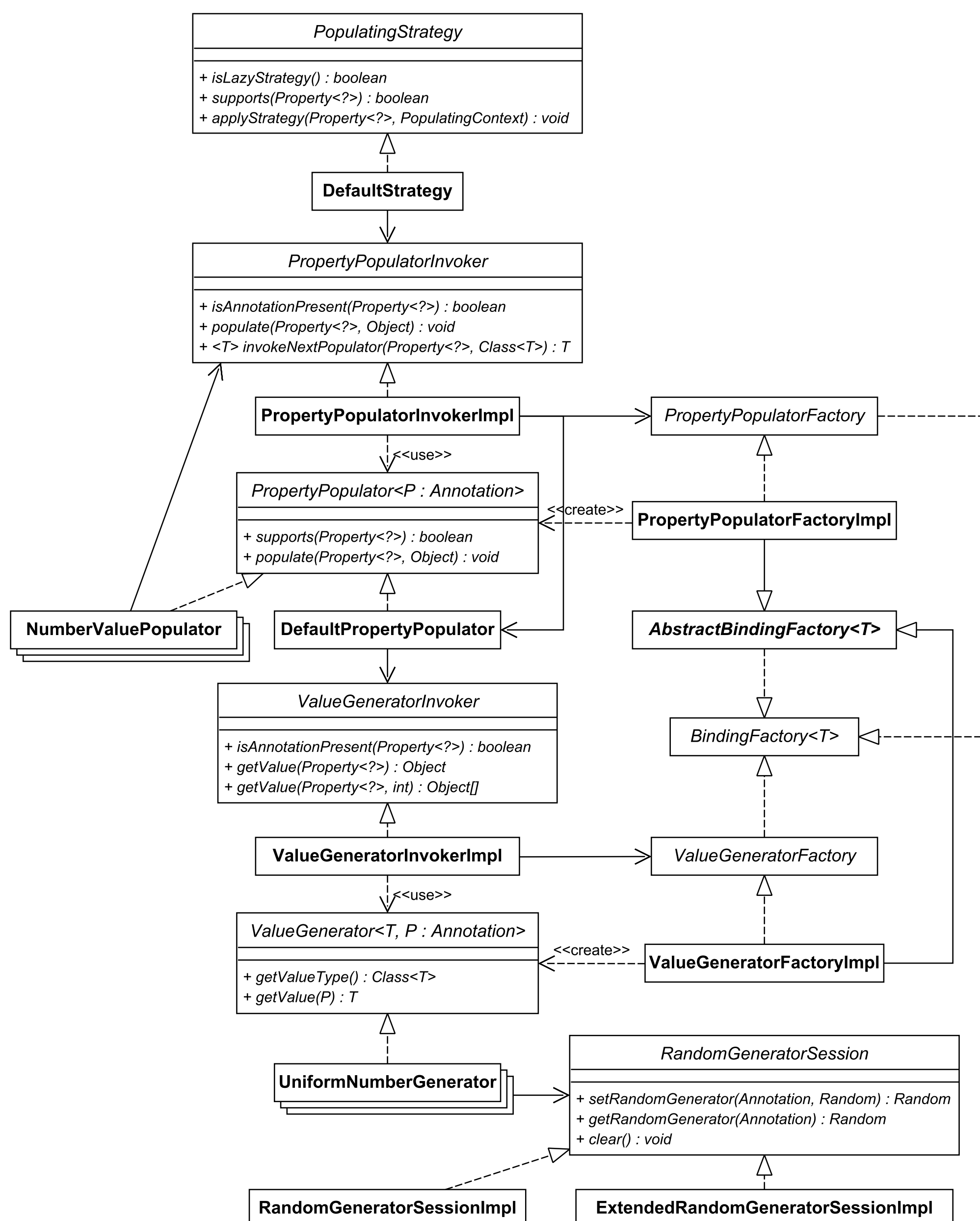
Obrázek D.1: UML diagram tříd pro strategii vytváření instancí

E UML diagram tříd pro strategii hledání instancí



Obrázek E.1: UML diagram tříd pro strategii hledání instancí

F UML diagram tříd pro výchozí strategii osazení atributů



Obrázek F.1: UML diagram tříd pro výchozí strategii osazení atributů