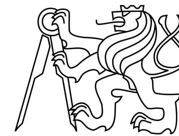
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název: Implementace DI kontejneru
Student: Samuel Butta
Vedoucí: Ing. Ji í Dan ek
Studijní program: Informatika
Studijní obor: Softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2017/18

Pokyny pro vypracování

Prostudujte techniku vkládání závislostí (Dependency injection).
Porovnejte jednotlivé kontejnery, které implementují DI pro jazyk Java, a navrhn te funkcionalitu vlastního ukázkového DI kontejneru.
Vlastní kontejner implementujte v jazyku Java a výsledek ov te na zkušební aplikaci.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

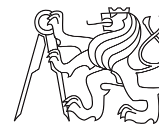
prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 20. listopadu 2016

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra Softwarového inženýrství



Bakalářská práce

Implementace DI kontejneru

Samuel Butta

Vedoucí práce: Ing. Jiří Daněček

14. května 2017

Poděkování

Rád bych poděkoval svému vedoucímu bakalářské práce Ing. Jiřímu Danečkovi, u kterého jsem absolvoval předměty AJP a EJA, za konzultace.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Samuel Butta. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Butta, Samuel. *Implementace DI kontejneru.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Při tvorbě netriviálních programů téměř v libovolném jazyce programátor dříve či později narazí na problém s předáváním závislostí mezi třídami. Řešením je použití různých návrhových vzorů, které mohou pomoci vyvarovat se problémů. Jedním z nejmodernějších přístupů k této problematice je použití návrhového vzoru *dependency injection*.

Ve své práci jsem se zaměřil na popis a porovnání nejpoužívanějších DI frameworků v jazyce Java a to konkrétně Google Guice, Spring IoC a Weld. Popsal jsem a diskutoval možnosti použití těchto projektů. Dále jsem navrhl a implementoval vlastní DI kontejner v jazyce Java, s použitím pokročilých konstruktů, jako je například reflexe, jehož praktické použití jsem ukázal na reálném příkladu.

Klíčová slova předávání závislostí, dependency injection, Java, návrhové vzory, reflexe, Google Guice, Spring, Weld, Gradle

Abstract

When creating non-trivial programs in almost any language, sooner or later the computer programmer encounters the problem of transferring the dependencies between different classes. A possible solution is to use different design

patterns that can help in avoiding these problems. One of the most state of the art approaches to solving this issue is using the design pattern *dependency injection*.

In my thesis I have focused on the description and comparison of the most frequently used DI frameworks in the Java language, specifically Google Guice, Spring IoC and Weld. I have described and discussed the possible application of these projects. Furthermore, I have designed and implemented my own DI container, written in Java, with the use of advanced constructs, such as reflection and I have demonstrated its practical use on a real-life example.

Keywords transferring dependencies, dependency injection, Java, design patterns, reflection, Google Guice, Spring, Weld, Gradle

Obsah

Úvod **1**

1 Cíl práce **3**

2 Jak lze chápat software **5**
2.1 Svět jako černá skříňka 5
2.2 Služby a komponenty 5
2.3 Klient a služba 6
2.4 Modulární aplikace 6
2.5 Aspektově orientované programování 6
2.6 Java Bean 7

3 Dependency injection **9**
3.1 Situace před DI 9
3.2 Co je tedy DI 12
3.3 DI kontejner (framework) 13
3.4 Výhody DI 13
3.5 IoC vs DI 13
3.6 Java Specification Request 13
3.7 Existující DI kontejnery na platformě Java 14
3.8 Metadata a konfigurace kontejneru 15
3.9 Způsoby propojení 16
3.10 Propojení konstruktorem (Contructor injection) 17
3.11 Propojení setterem (Setter injection) 17
3.12 Propojení atributem (Field injection) 17
3.13 Problém cyklických závislostí 17
3.14 Zavedení (bootstraping) DI 19
3.15 Scope 21

4 Existující DI frameworky **23**

4.1 Google Guice . . . 23
4.2 Spring IoC . . . 26
4.3 Weld (Java EE) . . . 30
4.4 Porovnání DI frameworků . . . 33

5 Vlastní DI kontejner **39**
5.1 Popis rozhraní . . . 39
5.2 Použité technologie . . . 39
5.3 Návrh . . . 40
5.4 Použití . . . 42
5.5 Konfigurace kontejneru . . . 42
5.6 Typy injektáže . . . 43
5.7 Setter injection . . . 44
5.8 Field injection . . . 44
5.9 Constructor injection . . . 44
5.10 Scope . . . 48
5.11 API kontejneru . . . 49
5.12 Testování . . . 49
5.13 Možné rozšíření . . . 50

6 Ukázková aplikace **51**
6.1 Sestavení a spuštění . . . 51
6.2 Použití . . . 52
6.3 Návrh aplikace . . . 54
6.4 Hibernate . . . 55
6.5 Vlastní DI framework . . . 57

Závěr **59**

Literatura **61**

A Seznam použitých zkratek **63**

B Obsah přiloženého CD **65**

Seznam obrázků

3.1 Způsoby konfigurace DI kontejneru [1] 16
3.2 Zavedení injektoru ve webové aplikaci [1] 20
3.3 Zavedení injektoru při spuštění aplikace [1] 20
3.4 Zavedení injektoru na vyžádání [1] 20
3.5 Líné zavedení injektoru [1] 21

4.1 Graf lineárně závislých služeb 34
4.2 Inicializace Weld kontejneru závislá na čase 35
4.3 Inicializace Spring kontejneru závislá na čase 36
4.4 Inicializace Spring kontejneru závislá na počtu služeb 37
4.5 Inicializace Weld kontejneru závislá na počtu služeb 38

5.1 Sekvenční diagram použití proxy třídy [2] 46
5.2 Grafický výstup testů DI kontejneru 50

6.1 Administrátorské menu ukázkové aplikace 52
6.2 Případy užití ukázkové aplikace 53
6.3 Diagram tříd servisní vrstvy ukázkové aplikace 54
6.4 Diagram tříd entitní vrstvy ukázkové aplikace 55
6.5 Diagram DAO tříd ukázkové aplikace 57

Seznam tabulek

4.1 Přehled scope v Google Guice frameworku 25
4.2 Přehled scope ve Spring IoC frameworku 29
4.3 Přehled scope v CDI Weld 32
4.4 Tabulka porovnání vlastností DI kontejnerů 33

5.1 Přehled rozšíření abstraktní třídy Injector v jádru frameworku . . 41
5.2 Přehled scope ve vlastním DI frameworku 48
5.3 Přehled metod API nad objektovým grafem 49

Úvod

Za dobu vývoje softwaru postupně vznikala různá programovací paradigmata. V jednoduchosti lze takové paradigma chápat jako styl struktury kódu, určuje tedy například základní stavební prvky programu. V dnešní době je ku příkladu velký zájem o takzvané funkcionální programování, přesto však je stále jedno z nejpoužívanějších paradigmat takzvané objektově orientované programování.

Objektově orientované programování je snaha přejít na takovou rovinu abstrakce, která je blízká člověku. Člověk nepřemýšlí v symbolických instrukcích jako procesor, ale svět vnímá jako množinu objektů, které jsou navzájem propojené nejrůznějšími vztahy a komunikují spolu. Kód je strukturován do tříd, což jsou jakési vzory pro konkrétní instance neboli objekty.

Při tvorbě ne zcela triviálních programů téměř v libovolném objektově orientovaném jazyce, vývojář dříve či později narazí na problém s předáváním závislostí mezi třídami. Problém nastává ve chvíli, když řeší, kdy a kde se budou vytvářet objekty, které spolu navzájem komunikují. Řešením je použití různých návrhových vzorů, které mohou pomoci vyvarovat se problémů. Jedním z nejmodernějších přístupů k této problematice je použití návrhového vzoru *dependency injection*. *Dependency injection* není svázán s žádným konkrétním jazykem, je však doménou objektově orientovaných jazyků.

Kapitola 1

Cíl práce

Cílem mé práce je představit problematiku spojenou s vytvářením a vkládáním závislostí při vývoji softwaru a popsat moderní návrhový vzor *dependency injection*.

Také chci zjistit, jaké jsou již existující DI frameworky, popsat je a případně porovnat. Nejdůležitější částí je implementace vlastního *dependency injection* kontejneru (frameworku).

Mým cílem je implementovat lehký DI kontejner pro menší desktopové, případně serverové aplikace, protože velká používaná řešení jsou mnohdy zbytečně robustní a nemusí se hodit pro menší aplikace. Vzhledem k tomu, že se jedná o mezivrstvu, která má smysl pouze jako součást jiné aplikace, dalším cílem je ukázat a otestovat jeho použití na reálné aplikaci.

Kapitola 2

Jak lze chápat software

2.1 Svět jako černá skříňka

Dnešní svět se stal díky exponenciálnímu technologickému růstu velmi složitý. Lidé používají různé předměty a technologie, jako je například televize, ale téměř nikdo nerozumí tomu, jak vlastně fungují. Uživatel ani nemusí, protože mu televize poskytuje rozhraní, které mu plně stačí. Například televize je tedy pro uživatele jakousi černou skříňkou (*black box*). Výrobce televize ovšem také nemusí téměř nic vědět o komponentech, ze kterých je složena. Nakoupí ku příkladu procesor, který také poskytuje určité rozhraní, a použije ho.

Při vývoji softwaru je situace zcela obdobná. Programátor používá ve své aplikaci knihovny, které jsou zpravidla výrazně kvalitnější, než pokud by je implementoval sám. To by dokonce v tvrdém konkurenčním prostředí nebylo vzhledem k náročnosti téměř možné a určitě ne ekonomické.

2.2 Služby a komponenty

Software v objektově orientovaném programování lze chápat jako množinu určitých komponent či služeb, které spolu navzájem komunikují a předávají si různé informace v podobě jiných objektů.

V tak dynamickém a mladém oboru jako je softwarové inženýrství nejsou mnohé často používané pojmy nijak striktně definovány. Stejná situace je i s termíny služba (*service*) a komponenta (*component*), které jsou důležité základní termíny pro *dependency injection*.

Jako službu lze chápat libovolný objekt, který poskytuje specializovanou funkcionalitu. [1] V kontextu DI lze prakticky službu a komponentu zaměnit.

Framework Spring, který DI kontejner obsahuje, používá mimo jiné jako anotace tříd právě slova *Component* a *Service*. Zde je služba chápána jako specializace komponenty. V oficiální dokumentaci frameworku Spring je spíše upřednostňováno používat označení *service*. [3]

Komponenta je dle definice Martina Folwera část softwaru, která má být použita "beze změny", tedy že užitím komponenty není měněn její zdrojový kód. Může být však měněno její chování různou konfigurací. [4]

Službu chápe obdobně jako komponentu, ovšem s tím rozdílem, že její použití je skrze vzdálené rozhraní (*remote interface*). Jako příklad uvádí webové služby nebo *messaging system*. [4]

Služby nemusí být a zpravidla nejsou triviální, jsou naopak závislé na mnohých dalších službách. Software lze dekomponovat na služby a komponenty, díky čemuž není problém vyměnit některou službu za jinou. Pokud jsou služby dostatečně abstraktní, mohou být znovu použitelné na mnohé jiné domény. [1]

Soubor propojených, tedy navzájem závislých komponent tvoří strukturu zvanou objektový graf (*object graph*). [1]

2.3 Klient a služba

Častým vzorem v softwaru, při komunikaci jednotlivých částí, případně různých systémů, je vzor klient služba. Jednotlivé části softwaru lze také rozdělit celkem logicky na službu a jejího klienta. Službou tedy lze chápat libovolný objekt s jasně definovanou funkčností. A jako klienta lze označit objekt, který tuto službu volá.

V kontextu DI lze tyto pojmy specifikovat na takzvanou závislost (*dependency*), tedy objekt, který je vyžadován jiným objektem, a závislého (*dependent*), tedy něco jako klienta, který potřebuje závislosti, aby mohl správně fungovat. [1]

2.4 Modulární aplikace

V OOP je celý software složen ze tříd, jakožto základních stavebních jednotek. Objekty lze sice chápat více způsoby, vždy se však jedná o konkrétní instanci třídy. Jde o jakousi jednotku programu, která poskytuje určitou funkcionalitu.

V OOP jazycích jsou třídy zpravidla shlukovány do větších celků, v Javě se takovýmto celkům říká balíčků. Balíčky mohou být součástí většího celku modulu. Modul by měla být nezávislá velká část aplikace, která zastřešuje jednu velkou funkcionalitu. Příkladem takového modulu může být persistentní modul, který má na starost komunikaci s databází a práci nad persistetními entitami. [1]

2.5 Aspektově orientované programování

V souvislosti s modulární aplikací stojí za zmínku koncept takzvaného aspektově orientovaného programování. AOP je programovací paradigma, které má za cíl zvýšit modularitu programu. Pokouší se rozdělit program na jakési části,

které se mezi sebou co nejméně překrývají svou funkcionalitou, jedná se o takzvané průřezové problémy (*crosscutting concerns*). [5]

Moderní velké DI frameworky s AOP počítají a umožňují tvorbu vlastních aspektů, viz Google Guice. [6]

2.6 Java Bean

Jazyk Java je jedním z nejpoužívanějších programovacích jazyků na světě. Díky své multiplatformnosti má ohromné uplatnění. Vzhledem k tomu, že je tento jazyk konzervativní, má dobrou zpětnou kopatibilitu, je léty prověřený a má ohromnou světovou komunitu vývojářů, je hojně používaný při tvorbě velkých podnikových (*enterprise*) aplikací.

Speciálně pro tyto účely vznikla platforma Java EE (dříva J2EE). Snaha o vývoj modulárních podnikových aplikací vyústila v definici takzvaných *JavaBeans*. *JavaBean* je znovu použitelná komponenta, respektive jedná se pouze o specifikaci, jak má tato třída vypadat. [7]

Termín *Bean* je hojně používaný a to i mimo Java EE například ve framworku Spring. Tento termín je spojen i s DI. CDI v Java EE však původní koncept Java *Beans* redefinovalo. [8]

Kapitola 3

Dependency injection

Jak již bylo řečeno, OOP software je množina tříd. Čím větší je program, tím více má tříd a tím více vztahů mezi nimi obsahuje. Nastává tedy problém, kdy a kde mají jednotlivé objekty vznikat. Špatně zvolené postupy mohou vyústit v mnoho chyb.

Řešením je použití různých návrhových vzorů. Jedním z návrhových vzorů je i *dependency injection*. DI se snaží odebrat třídám zodpovědnost za vytváření objektů. Jinými slovy, pokud třída vyžaduje nějaký externí objekt (službu), nevytváří si ho sama, ale je jí určitým způsobem dodán z venku.

3.1 Situace před DI

V následující ukázce se vyskytuje servisní třída, která reprezentuje nákupní košík. Mimo jiné obsahuje metodu pro potvrzení objednávky. Součástí je i odeslání potvrzovacího emailu. Výhodné je použít externí knihovnu, která zajišťuje odesílání emailů. Rozhraním takové knihovny je instance třídy *Mailer*, která má metodu pro odesílání emailů.

3.1.1 Přímé volání konstruktoru

Nejpřímější a nejjednodušší postup je vytvoření objektu pomocí explicitního volání konstruktoru přímo v kódu na místě, kde je potřeba, přes klíčové slovo *new*. Takovéto objekty jsou pak takzvaně těsně spojeny (*tightly coupled*). Největšími nevýhodami takového propojení jsou špatné testování, špatná znovupoužitelnost jednotlivých částí a hůře pochopitelný kód. [9]

```
1  public class BasketService {
2      ...
3      public void confirmOrder() {
4          ...
5          // odeslani potvrzovaciho emailu
6          Mailer mailer = new Mailer();
```

```
7           mailer.send(message);
8
9           ...
10      }
11  }
```

Obtížným testováním je myšleno to, že prakticky není možné testovat třídu *BasketService* samostatně pomocí jednotkových testů (*unit test*). Znamená to, že pokud chce vývojář otestovat tuto třídu, zároveň s ní již testuje třídu *Mailer*. Není například možné třídě předat testovací objekt takzvaný *mock*.

3.1.2 Factory pattern

Tento návrhový vzor usnadní programátorovi vyměnit implementaci třídy. Jedná se v podstatě o přidání nové vrstvy. Tovární třída (*factory class*) ve své nejjednodušší formě zpravidla udržuje právě jednu instanci, kterou lze získat zavoláním statické metody.

```
1   public class MailerFactory {
2
3       private static Mailer instance;
4
5       public static void setInstance(Mailer mailer) {
6           instance = mailer;
7       }
8
9       public static Mailer getInstace() {
10          if(instance == null) {
11              return new Mailer();
12          }
13          return instance;
14      }
15  }
```

Případně může poskytovat rozhraní například *setter* pro nastavení její hodnoty. Tato vlastnost je výhodná při testování, kdy může být skutečná implementace nahrazena *mock* objektem. Získání objektu pomocí *factory pattern* vypadá následovně.

```
1   Mailer mailer = MailerFactory.getInstace();
```

Díky setteru je nyní možné nastavit testovací instanci *mailer*. Takovéto řešení má však také své výrazné nedostatky. Nejen že přidává zbytečný kód navíc, může způsobit problémy také při testování, kdy globální proměnná, která udržuje testovací implementaci, musí být jak řádně nastavena, tak od-

nastavena. Pokud se například nepodaří tovární třídu řádně ukončit, může to způsobit problémy v běhu dalších testů. [6]

3.1.3 Service locator

Service locator je návrhový vzor, který je dost podobný vzoru továrny. Prakticky se jedná o třídu, která je zodpovědná za tvorbu více služeb. Služba je zpravidla získána podle jména. V následující ukázce je použito rozhraní *Mailer* a jeho dvě implementace *GoogleMailer* a *AmazonMailer*. Třída *ServiceLocatorMailer*, která je zodpovědná za tvorbu objektů, by vypadala následovně.

```
1  public class ServiceLocatorMailer {
2
3      public Mailer lookUp(String name) {
4
5          if(name.equals("googleService")) {
6              return new GoogleMailer();
7          } else if(name.equals("amazonService")) {
8              return new AmazonMailer();
9          }
10         return null;
11     }
12 }
```

Použití by bylo následující.

```
1  ServiceLocatorMailer loc = new ServiceLocatorMailer();
2
3  // ziskani instanci podle jmena
4  Mailer googleService = loc.lookUp("googleService");
5  Mailer amazonService = loc.lookUp("amazonService");
```

Programovým třídám může být tato instance předávána. Nevýhodou takového postupu je, že má třída přístup k téměř všem instancím a není jasné jaké závislosti vlastně třída nutně potřebuje.

Service locator samozřejmě nemusí vytvářet třídy pouze jednoho rozhraní. Stejně jako u vzoru továrny přetrvávají stejné problémy při testovatelnosti. Identifikátor pomocí textových řetězců také není příliš vhodný kvůli snadné náchylnosti na špatně detekovatelnou chybu.

Tento návrhový vzor je používán jako mezipaměť (*cache*) při použití náročného vyhledávání služeb a objektů pomocí jmen (*JNDI lookup*). V současné době se však jedná spíše o zastaralou technologii. [1]

3.2 Co je tedy DI

Podobně jako *factory pattern* je i *dependency injection* návrhový vzor. [6] Základní princip DI spočívá v tom, že objekt není zodpovědný za vytváření servisních tříd. Závislost je mu dodána z venku. Martin Fowler definoval tři konvenční způsoby, jak je možné do třídy objekt předat.

1. **Vkládání rozhraním** - Externí modul, který je do objektu přidán, implementuje rozhraní, jež objekt očekává v době sestavení programu. [4]

2. **Vkládání setter metodou** - Objekt má setter metodu, pomocí níž lze závislost injektovat. [4]

3. **Vložení konstruktorem** - Závislost je do objektu injektována v parametru konstruktoru již při jeho zrodu. [4]

V následující ukázce je použita stejná služba jako v předchozí ukázce. Kód, který dodržuje DI princip, může vypadat následovně.

```
1  public class BasketService {
2
3      private Mailer mailer;
4
5      private BasketService() {
6          // prazdny privatni konstruktor
7      }
8
9      public BasketService(Mailer mailer) {
10         this.mailer = mailer;
11     }
12
13     ...
14     public void confirmOrder() {
15         ...
16         // odeslani potvrzovaciho emailu
17         mailer.send(message);
18
19         ...
20     }
21 }
```

V uvedené ukázce je použit případ předání závislosti v konstruktoru. Díky tomu není možné vytvořit instanci bez předání služby *Mailer*.

3.3 DI kontejner (framework)

DI kontejner je externí knihovna, která závislosti spravuje. Je zodpovědný za vytváření instancí, jejich propojení napříč celou aplikací a řídí životní cyklus objektů. [6]

Termín DI kontejner a DI framework je zaměnitelný. Osobně chápu DI kontejner jako součást DI frameworku.

3.4 Výhody DI

Podstatou je to, že kód psaný programátorem je zbaven nutnosti vytváření závislostí. Platí takzvaný holywoodský princip: "Nevolej nás, my zavoláme tobě." [1]

Díky tomu se může vývojář skutečně zaměřit pouze na specifika svého softwaru (*behaviorally focused*) a není obtěžován zbytečnou režií.

Software se stává modulární, skládá se z jednotlivých částí (komponent, služeb), které jsou jednodušší na údržbu a jsou znovupoužitelné. Největší výhodu osobně spatřuji v testování, především v jednotkovém testování. Jednotkový test by měl testovat právě jednu třídu. Ovšem třídy bývají závislé na mnoha dalších službách. Při použítí DI není problém místo skutečné implementace předat pouze *mock*.

3.5 IoC vs DI

Inversion of control je obecný princip návrhu softwaru, v kterém klientský kód získává řídící programové struktury (*control flow*) z generického frameworku. Zatímco v tradičním procedurálním programování je to uživatelský kód, který volá různé externí knihovny, v IoC principu je to framework, který volá klientův kód. Jedná se o předání zodpovědnosti jiné struktuře. DI je lze chápat jako konkrétní užití IoC principu. [10]

Tyto termíny však nejsou nijak striktně definovány a jsou velice často zaměňovány. Zajímavé například je, že pro framework Spring je termín IoC a *dependency injection* stejný. [3]

3.6 Java Specification Request

Jazyk Java byl vyvinut společností Sun Microsystems. V roce 2010 však byla tato společnost akvizována společností Oracle Corporation, která tím získala mimo jiné práva na jazyk Java. [11]

Oracle tvoří a má pod správou komunitu pro vývoj specifikací pro technologii Java (*Java Community Process*). Na webových stránkách komunity se může kdokoliv zaregistrovat a poskytovat tak zpětnou vazbu. [12]

Tato komunita určuje směr a kurz Javy a vytváří takzvané specifikace JSR (*Java Specification Requests*). Jedná se v podstatě o rozhraní, která popisují, jak se má daná technologie používat. Tuto specifikaci pak může implementovat kdokoliv. Většinou existuje i referenční implementace, kterou vytvoří velká společnost, jako je například JBoss nebo Red Hat.

Specifikace je vždy označena nějakým číslem. Specifikace pro *dependency injection* v Javě je JSR-330. Vedoucí specifikace je Rod Johnson a Bob Lee. [13] Mimo to existuje ještě specifikace přímo pro Javu EE JSR-346, která v sobě mimo JSR-330 obsahuje také specifikace JSR-342 pro takzvané *Managed beans*. [8]

3.7 Existující DI kontejnery na platformě Java

Java platforma je považována za rodiště *dependecy injection*, která má mnoho velmi vyspělých DI knihoven v porovnání s ostatními platformami. [1]

3.7.1 Apache Avalon

Apache Avalon je jedním z nejranějších DI knihoven ve světě Javy. Jeho vývoj započal již v roce 1999 [14], dávno před tím, než začala převládat Java EE a aplikační servery. Vývoj na tomto projektu je již ukončený.

3.7.2 Google Guice

Google Guice je DI framework s otevřeným zdrojových kódem vyvinutý především Bobem Lee ve společnosti Google. Prezentuje se jako lehký (*lightweight*) framework pro verzi Java 6 a vyšší. Vznikl v roce 2008 a byl první DI framework, který používal Java anotace. Vyhrál cenu *Jolt Award* za nejlepší knihovnu, framework nebo komponentu. [15]

3.7.3 Weld (Java EE)

Weld je referenční implementace *CDI: Contexts and Dependency Injection for the Java EE Platform*, což je JCP standard. Jedná se o jednu z nejdůležitějších součástí Java EE.

Weld je integrován do mnoha Java EE aplikačních serverů jako je WildFly, JBoss Enterprise Application Platform, GlassFish, Oracle WebLogic Server, WebSphere Application Server a jiné. Může být také integrován do webových aplikací běžících na servletovém kontejneru jako je Tomcat nebo Jetty. Dále může být také použit do Java SE aplikací. [16]

3.7.4 Dagger

Dagger je DI pro Android a Javu. Použití je velmi podobné Google Guice, ovšem by měl být rychlejší, protože používá generování kodu na místo reflexe.

Snaží se také používat standardní JSR anotace všude, kde je to možné. Má jednoduché API, oproti Google Guice však omezené. [17]

3.7.5 Spring Framework

Spring Framework je považován za průkopníka a základní kámen *dependency injection* knihoven ve světě Javy. [1] Spring framework byl vytvořen Rodem Johnsonem, publikovaný jako open source byl v roce 2003. Od té doby se velice rozrostl a stále rozrůstá, přispívá do něho velké množství autorů. [18] Dependenci injection je pouze malá, ale velmi důležitá část Spring frameworku.

3.7.6 Picocontainer

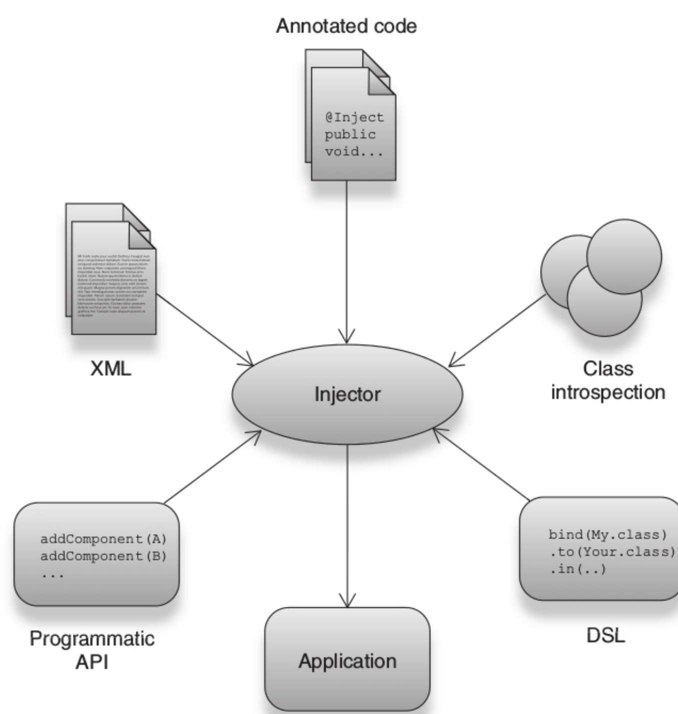
Picocontainer je velmi malá a odlehčená knihovna, jedná se pravděpodobně o první DI kontejner, který nabízel spojování přes konstruktor (*constructor wiring*). Byl vyvíjen spíše jako ukázka samotných teoretických principů spojených s DI. Za zmínku také stojí od něho odvozený projekt Nanocontainer. [1]

Toto je stručný přehled nejpoužívanějších knihoven současnosti, později detailněji popíši některé vybrané. Představím rozhraní, které programátorovi poskytují.

3.8 Metadata a konfigurace kontejneru

DI framework vždy obsahuje část kterou lze nazvat *injektor*. DI framework respektive jeho část injektor lze v jednoduchosti chápat jako službu, která předzpracuje aplikační kód na základě speciálních konfiguračních informací a pak se stará o předávání závislostí. Takovéto konfugrační informace se nazývají metadata. DI knihovny nabízí mnoho mechanizmů, jak se dá injektor nakonfigurovat. Větší řešení zpravidla umí všechny zde uvedenté typy konfigurací, případně také umožňují konfiguraci rozšířit o vlastní. [1]

1. XML soubor
2. Pomocí programového rozhraní
3. DSL soubor
4. Podle struktury tříd (reflexe, introspekce)
5. Pomocí anotací (reflexe, introspekce)



Obrázek 3.1: Způsoby konfigurace DI kontejneru [1]

V současné době (2017) se obecně minimálně ve světě Javy opouští od konfigurace pomocí mnoha XML souborů, protože fakticky nejsou součástí jazyka a tím, že se vyskytují mimo projekt, mohou být hůře pochopitelné. Je zde tedy jasná snaha o stoprocentní konfiguraci pomocí anotací a tříd. Tento trend je dobře patrný například v novějších verzích Spring frameworku. [3]

3.9 Způsoby propojení

Obecně v DI existují dva základní způsoby, jak objekty propojovat. První způsob je, že třída vyžaduje závislost v konstruktoru. Druhý způsob je nastavení závislosti setterem. DI frameworky zpravidla umožňují použít oba způsoby, některé přidávají i další varianty propojování. Použití určitého způsobu může mít velký dopad na architekturu a funkčnost celé aplikace. Může mít vliv jak na výkon, tak třeba na škálovatelnost. Tyto důsledky vyplývají z rozdílných jazykových vlastností metod a konstruktoru. Jako příklad lze uvézt, že konstruktor je volán pouze jednou, zatímco metoda může být volána kdykoliv. Konstruktor v Javě však umí například nastavit finální atributy. [1]

3.10 Propojení konstruktorem (Contructor injection)

Propojení konstruktorem je v současné době pravděpodobně nejvíce doporučený postup použití. [3] Jeho výhoda spočívá především v tom, že konstruktor je ve svém principu volán pouze jednou a umožňuje tak nastavit atributy označené modifikátorem *final*. Jinými slovy umožňuje vytvořit neměnné (*immutable*) objekty. Neměnné objekty jsou velice důležité pro vícevláknové aplikace, protože jsou bezpečné (*thread safe*).

3.11 Propojení setterem (Setter injection)

Atribut, který má být nastaven, má přístupný setter. Bohužel díky tomu již z principu nemůže být daný objekt neměnný.

3.12 Propojení atributem (Field injection)

Trochu kontroverzní a ne zcela doporučovaný způsob je propojení atributem takzvaný *field injection*. K injektáži není potřeba setter či konstruktor, ale pouze třídní atribut (*field*), který je anotován. Tento atribut není již nikde nastavován, o jeho inicializaci se stará DI framework. Tuto možnost poskytují všechny větší DI frameworky. [1]

```
1  // Guice, Weld
2  @Inject
3  private SomeService service;
4
5  // Spring
6  @Autowired
7  private SomeService service;
```

Výhodou je, že šetří počet řádků kódu, právě kvůli tomu ji velmi často můžeme vidět v různých návodech a ukázkách.

3.13 Problém cyklických závislostí

Při návrhu a vývoji softwaru občas nastane situace, kdy jsou dvě třídy, ať už přímo nebo tranzitivně, závislé obě navzájem na sobě. Obecně se dá prohlásit, že pokud se v programu vyskytne cyklická závislost, jedná se o špatný návrh. Na druhou stranu není teoreticky nemožné, aby takováto závislost vznikla, a může být i nutná.

Při použití injektáže pomocí setteru v podstatě nevzniká problém. Při inicializační fázi nejsou závislosti nastaveny, mají tedy hodnotu *null*. K jejich nastavení dojte až později. Jedním z problémů je, že takováto závislost nemůže

být konečná (*final*). Při použití contructor injection není možné propojení přímo pomocí konstuktorů.Vzniká variace odvěkého problému, zda bylo dříve vejce nebo slepice. [1]

Pro ilustraci uvedu následující příklad. V ukázce jsou uvedeny třídy A a B, které jsou na sobě přímo závislé a pro vytvoření instance potřebují v konstruktoru svoji závislost.

```
1  public class A {
2    private final B b;
3    public A(B b) {
4    this.b = b;
5     }
6  }
7
8  public class B {
9    private final A a;
10   public B(A a) {
11   this.a = a;
12    }
13 }
```

Při vytváření instance třídy A, je potřeba do parametru konstruktoru předat také instanci třídy B. Aby však bylo možné vytvořit instanci třídu B, je potřeba zase její instanci předat do parametru konstruktoru instanci třídy A.

```
1  A a  = new A(new B(....))
```

Je patrné, že to není možné. Řešením tohoto problému je použití jakési mezitřídy, takzvaná *proxy* třída. Prvním podstatným rozdílem je, že třídy nereferují přímo na sebe, ale na rozhraní, které implementují.

```
1  public interface A {}
2
3  public interface B {}
4
5
6  public class AImpl implements A {
7
8      private final B b;
9
10     public AImpl(B b) {
11         this.b = b;
12     }
13 }
14
15 public class BImpl implements B {
16
17     private final A a;
```

```
18
19     public BImpl(A a) {
20         this.a = a;
21     }
22 }
```

Pro jednu třídu je nutné vytvořit *proxy* třídu, která implementuje stejné rozhraní.

```
1 public class AProxy implements A {
2
3     private A a;
4
5     public void setA(A a) {
6         this.a = a;
7     }
8 }
```

Vytvoření objektů je pak možné. Nejprve je vytvořen objekt *proxy* třídy A a do instance třídy B je v konstruktoru předána reference. Instanci třídy A je možné již vytvořit i s parametrem třídy B. V posledním kroku je nutné nastavit požadovanou instanci třídy A.

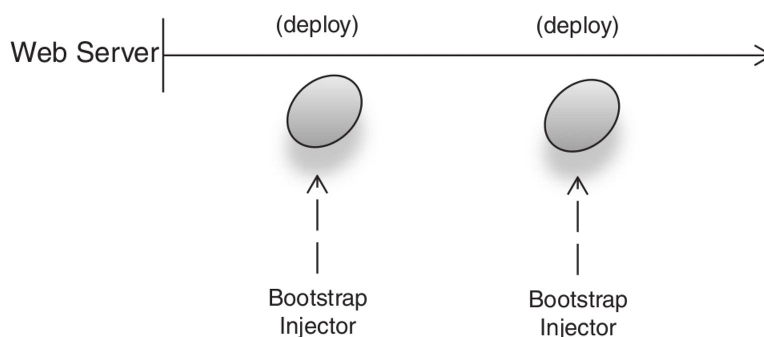
```
1 AProxy aProxy = new AProxy();
2
3 B b = new BImpl(aProxy);
4 A a = new AImpl(b);
5
6 aProxy.setA(a);
```

V tuto chvíli pokud B volá metody z A, nevolá přímo instanci třídy A, ale zavolá její proxy, která již volá metody samotného objektu A, který byl nastaven setterem. Současné velké DI frameworky dokáží vytvořit *proxy* třídu až za běhu aplikace, takže se o ni vývojář nemusí starat.

3.14 Zavedení (bootstraping) DI

Dependency injector je ve své podstatě také služba, která musí být nějak zavedena a inicializována. Fáze inicializace se liší v závislosti na typu aplikace.

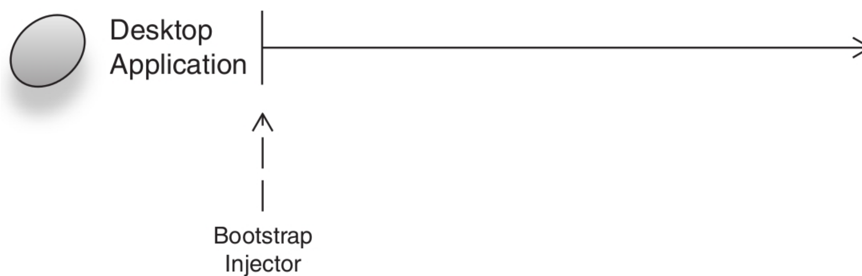
V případě webové aplikace, která běží na webovém serveru, dojde k zavedení při inicializační fáze životního cyklu aplikace při nasazení aplikace (*deployment*).



Obrázek 3.2: Zavedení injektoru ve webové aplikaci [1]

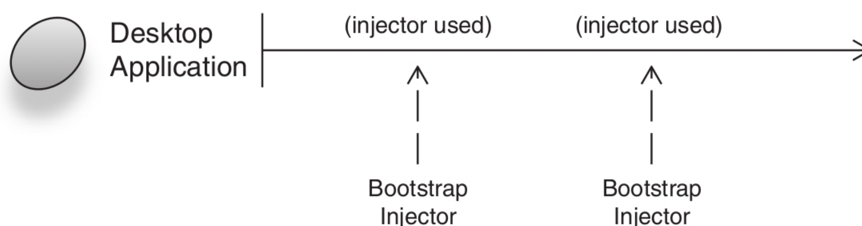
U klasických desktopových aplikací může dojít k inicializaci v následujících situacích.

1. Při spuštění (*on startup*) - okamžitě po spuštění aplikace



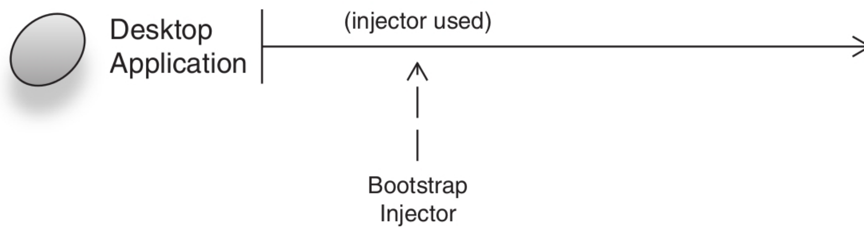
Obrázek 3.3: Zavedení injektoru při spuštění aplikace [1]

2. Na vyžádání (*on demand*) - pokaždé, když je závislost vyžadována, dojde k nové inicializaci injektoru



Obrázek 3.4: Zavedení injektoru na vyžádání [1]

3. Líně (*lazy*) - poprvé, kdy je závislost vyžadována



Obrázek 3.5: Líné zavedení injektoru [1]

3.15 Scope

Velice důležetý pojem spojený s DI kontejnery je takzvaný *scope*. *Scope* lze chápat jako dobu, po kterou daná instance existuje. Jinými slovy, *scope* je kontext, v kterém existuje pouze jedna instance tohoto daného typu. [1]

Nejjednodušší typy *scope* jsou takzvaný *singleton* a *no-scope*. Zasazeno do předchozí definice, *singleton scope* lze chápat jako kontext pro celou aplikaci. Tedy při požádání instance je vždy vrácena jedna a ta samá. Zatímco při použití *no-scope* je vždy vytvářena nová instance.

Pro webové aplikace existují specifické typy *scope*. *Request scope* je spojen s příslušným HTTP *request* jednoho klienta. *Session scope* zase je spojen s příslušnou HTTP *session* daného klienta. [1]

Kapitola 4

Existující DI frameworky

V této kapitole představím tři nejpoužívanější větší DI frameworky, konkrétně Google Guice, Spring IoC a Weld. Vzhledem k tomu, že každý framework nabízí množství nejrůznějších možností pro specifické případy použití, omezím se pouze na ukázku základního použití při vývoji aplikací na platformě Java SE.

Následně představím tabulku srovnání určitých vlastností kontejnerů a porovnám inicializační dobu.

4.1 Google Guice

Google Guice je středně velký DI framework, vyvíjený spolčeností Google. V tuto chvíli (březen 2017) je poslední stabilní vydaná (*release*) verze 4.1.0 (leden 2016).

4.1.1 Konfigurace

Konfigurace frameworku probíhá pomocí třídy, která implementuje rozhraní *Modul*. Pro jednodušší zápis je výhodnější vytvořit potomka třídy *AbstractModule*, který toto rozhraní implementuje a již poskytuje některé předpřipravené metody včetně abstraktní metody *configure*, kterou je potřeba překrýt.

V této metodě se registrují všechny objekty, které má Guice spravovat. Definuje se zde takzvaný *binding*, který probíhá pomocí metody *bind*. Tímto je explicitně řečeno, jaká třída má být kontejnerem spravována. Je zde také možné napojit rozhraní na jeho implementaci. [6]

Potomek třídy *AbstractModul* se nazývá modul. Moduly jsou základní stavební prvky, podle kterých je vytvořen objektový graf.

Guice nabízí několik typů *binding*, například takzvaný *Linked Binding* napojuje typ třídy na jeho implementaci. V následující ukázce je ukázána jednoduchá konfigurační třída *AppModule*, která v metodě *configure* propojuje rozhraní služby *MessageService* s implementací *EmailService*.

```java
public class AppModule extends AbstractModule {

    protected void configure() {
        bind(MessageService.class).to(EmailService.class);
    }

}
```

4.1.2 Použití kontejneru

Při inicializaci kontejneru je vstupním bodem instance třídy *Injector*, jehož instanci lze získat pomocí statické metody *createInjector*. Tato metoda přijímá v parametru instance modulů. Při zavolání dojde k inicializaci a konfiguraci DI frameworku. Většina aplikací zavolá tuto metodu právě jednou v metodě *main*, která je vstupním bodem celé aplikace. [6]

```java
Injector inj = Guice.createInjector(new AppModule());
```

4.1.3 Získání instance

Vytvořená instance reprezentuje objekt injektoru. Požadovanou třídu lze získat metodou *getInstance*, která jako parametr přebírá typ třídy, jejíž instance je vyžadována. [6]

```java
MailClient client = inj.getInstance(MailClient.class);
```

4.1.4 Scope

Scope definuje dobu životnosti služby. Ve výchozím stavu Guice vrací vždy nově vytvořenou instanci. Toto chování je možné nakonfigurovat změnou *scope*. Definice *scope* probíhá pomocí anotací nebo pomocí metody *bind*. [6]

```java
// definice scope pomoci anotace
@Singleton
public class ServiceImpl implements Service {
}

// definice scopu pomoci metody bind
bind(Service.class).to(ServiceImpl.class).in(Singleton.class);
```

Tabulka 4.1: Přehled scope v Google Guice frameworku

Scope	**Popis**
Implicitní	Pokud scope není uveden, vždy se vytváří nová instance.
Singleton	Existence jedné instance napříč celou aplikací.
RequestScope	Existence instance je spojená s existencí HTTP request.
Session	Existence instance je spojená s existencí HTTP session.
Custom	Je možné definovat vlastní scope, pro specifické použití. Tento postup není doporučován.

4.2 Spring IoC

Spring Framework je jedním z nejpoužívanějších nástrojů na tvorbu podnikových aplikací. Jeho původním autorem je Rod Johnson, první verze byla vydána v březnu roku 2004. Jedná se prakticky o konkurenci nebo také rozšíření v Java EE a EJB. Tento framework výrazně ovlivnil a ovlivňuje celý svět Javy včetně Java EE.

Spring také stejně jako Java EE používá termín *Bean*, chápe ho jako jakoukoliv komponentu spravovanou kontejnerem, může to být i obyčejný POJO objekt. [19]

DI kontejner, který se ve Springu nazývá IoC (*inversion of control*) kontejner, je součástí modulu *spring-contexts*. V tuto chvíli (březen 2017) je poslední stabilní verze *4.3.7.RELEASE*.

4.2.1 Konfigurace

Rozhraní *BeanFactory* poskytuje pokročilý konfigurační mechanismus a základní funkcionality kontejneru. Jeho podrozhraní *ApplicationContext* ho rozšiřuje o další funkčnosti.

Rozhraní *ApplicationContext* reprezentuje Spring IoC kontejner. Je plně zodpovědný za načtení definic *Bean* a jejich propojení (*wiring*). Součástí Springu jsou také implementace tohoto rozhraní. Pro běžnou samostatně běžící aplikaci je typické použít implementace *ClassPathXmlApplicationContext* případně *FileSystemXmlApplicationContext*.

Konfigurační metadata mohou být předána jako XML soubor, pomocí Java anotací, případně konfiguračních Java tříd. Rozdílnou konfiguraci poskytují různé implementace rozhraní *ApplicationContext*. [3]

4.2.2 Ukázka konfigurace

Konfigurace lze provézt pomocí konfiguračních tříd. Takováto konfigurační třída je označena anotací *Configuration* a obsahuje metody s anotací *Bean*, které spojují rozhraní s implementací a kde název metody je identifikátor *Bean*.

```
1  @Configuration
2  public class Config {
3    @Bean
4    public MyService myService() {
5        return new MyServiceImpl();
6    }
7  }
```

Ekvivalentní zápis v XML notaci by vypadal následovně.

```
<beans>
  <bean id="myService" class="com.services.MyServiceImpl"/>
</beans>
```

Propojování *Bean* v XML souboru je pomocí vnořených tagů. Argument konstruktoru třídy *A* odkazující na *Bean* s identifiátorem *B*.

```
<bean id="A" class="cvut.fit.spring.A">
<constructor-arg ref="B" />
</bean>

<bean id="B" class="cvut.fit.spring.B">
</bean>
```

4.2.3 Použití kontejneru

Vytvoření kontejneru je velice přímočaré.

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "daos.xml"});
```

ClassPathXmlApplicationContext implementace se použije ve chvíli, kdy jsou konfigurační soubory na *classpath*. Názvy souborů se pak předají jako pole textových řetězců.

V případě definice konfiguračních souborů v jazyce Groovy [20], který je v dnešní době velice populární, viz například konfigurace sestavovacího nástroje Gradle [21], je použití následující.

```
ApplicationContext context =
new GenericGroovyApplicationContext("services.groovy", "daos.groovy");
```

4.2.4 Získání instance

Instance se získá z aplikačního kontextu, pomocí metody *getBean*. Tato metoda je dostupná v několika definicích, lišících se v parametrech. Základní metodou je získání *Bean* podle jména, podle definovaného textového identifikátoru. [9]

Vytvoření kontejneru a získání instance podle definice výše uvedené třídy *Config* by vypadalo následovně.

```
ApplicationContext context =
new AnnotationConfigApplicationContext(Config.class);

```

```
4 MyService client = context.getBean("myService", MyService.class);
```

4.2.5 Typy injektáže

Spring framework podporuje všechny zmíněné type injektáže. Je možné kombinovat *setter injection* a *constructor injection*. Obecně ve Springu však platí konvence, že pro nutné závislosti se používá *constructor injection*, zatímco pro nepovinné se používá *setter injection*.

Obecná doporučení Spring komunity jsou:

1. Používat *constructor injection* všude, kde je to možné
2. Implementovat aplikační komponenty (třídy) jako neměnné
3. Žádné z požadovaných závislostí nesmí být *null*

Cyklická závislost je detekována za běhu programu, pokud se jedná o *constructor injection*, Spring IoC kontejner vyhodí výjimku *BeanCurrentlyInCreationException*. Obecně se dá říct, že se jedná o špatný návrh aplikace. Tímto je vývojář na takovouto chybu upozorněn. Pokud není možné z různých důvodů software dekomponovat tak, aby cyklickou závislost neobsahoval, je doporučováno použít *setter injection*. [3]

4.2.6 Scope

V následující tabulce jsou uvedeny možné typy *scope*. Pokud není *scope* uveden, je použita výchozí hodnota, kterou je *scope Singleton*. První čtyři uvedené typy jsou zcela běžné a v různých variacích je nalezneme ve všech větších DI frameworkách. Další jsou již specifické pro Spring framework. Spring IoC také umožňuje definovat vlastní (*custom*) *scope*. Kromě prvních dvou typů *scope* je kontext jejich použití validní pouze u webových aplikací. [3]

Tabulka 4.2: Přehled scope ve Spring IoC frameworku

Scope	**Popis**
Singleton	Jedná se o výchozí scope. Bean s tímto scope má v IoC kontejneru jedinou instanci.
Prototype	Při každém požádání dojde k vytvoření nové instance.
Request	Bean je spojená s existencí příslušného HTTP požadavku.
Session	Bean je spojená s existencí příslušné HTTP session.
GlobalSession	Bean je spojená s existencí globální HTTP session.
GlobalSession	Bean je spojená s existencí instance ServletContext. Jedná se o kontextový soubor u webových servletových aplikací.
Websocket	Bean je spojená s existencí instance Websocket.

4.3 Weld (Java EE)

Weld je takzvaná referenční implementace *Context and Dependency injection for the Java EE Platform*. Implementuje CDI specifikaci JSR-346, která definuje typově bezpečný mechanismus na platformě Java EE.

Bean, která je injektována, má dobře definovaný životní cyklus (*life cycle*) a je svázána s kontextem životního cyklu (*life-cycle contexts*). Injektována může být téměř kterákoliv POJO třída, včetně například EJB a JNDI zdrojů. [19]

Jak už název napovídá, poskytuje CDI dvě základní služby. *Contexts* umožňuje asociovat *Bean* (komponentu) s dobře definovaným (*well-defined*) životním cyklem. Jedná se o přiřazení *scope*.

Druhou službou je klasické *dependency injection*, které umožňuje injektáž komponent. [19]

4.3.1 Konfigurace

CDI potřebuje jako konfigurační soubor *bean.xml*, který musí být součástí JAR souboru aplikace. Musí být přítomen v adresáři *META-INF*, v případně webové aplikace v adresáři *WEB-INF* nebo na *classpath*.

Konfigurace v souboru *bean.xml* umožňuje nastavit způsob objevování Bean (*Bean mode*).

Soubor *bean.xml*, v kterém je explicitně uvedena verze CDI (1.1), vypadá následovně.

```
1 <beans
2 xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5 http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
6 bean-discovery-mode="all">
7 </beans>
```

V souboru však nemusí být uvedena verze CDI. Případně může být zcela prázdný. Pokud chybí zcela, je CDI vypnuto. Jeho absence bývá častou chybou.

```
1 <beans xmlns="http://xmlns.jcp.org/xml/ns/javaee">
2 </beans>
```

4.3.2 Použití kontejneru

Při použití sestavovacího nástroje stačí přidat závislost na knihovně *weld-se-core*.

V tuto chvíli (duben 2017) je poslední stabilní verze *2.4.3.Final*. V případě použití nástroje Gradle vypadá přidání závislosti následovně.

```
1 compile group: 'org.jboss.weld.se',
```

```
name: 'weld-se-core', version: '2.4.3.Final'
```

4.3.2.1 Inicializace

Nejprve je potřeba vytvořit instanci třídy *Weld*, z které se získá kontejner metodou *initialize*.

4.3.2.2 Získání instance

Nad kontejnerem lze zavolat metodu *select* s parametrem typu třídy. Tato metoda vrací instanci třídy *WeldInstance*, na které je možné zavolat metodu *get*, která vrátí již konkrétní instanci.

Pro správné ukončení a uzavření všech kontejnerů je nutné zavolat metodou *shutdown*.

Jednoduché použití vypadá následovně.

```
Weld weld = new Weld();
WeldContainer container = weld.initialize();

// ziskani instance
Service service = container.select(Service.class).get();

// uzavreni vsech kontejneru
weld.shutdown();
```

Weld implementuje všechny tři dříve zmíněné typy injektáže. Všechny tyto typy je možné kombinovat. Konstuktor nemusí být veřejný (*public*). Pokud je použita injektáž konstruktorem, umožňuje *Bean* být neměnná (*immutable*). Inicializační fáze má následující průběh.[19]

1. Zavolá se výchozí konstruktor, případně ten s anotací *Inject*
2. Inicializují se všechny atributy (*field*) s anotací *Inject*
3. Zavolají se všechny metody s anotací *Inject*
4. Zavolá se metoda s anotací *PostConstruct*

Za zmínku také stojí anotace *Qualifier*, která dovoluje specifikovat *Bean*, jež má být použita v případě, že je přítomno více implementací stejného rozhraní.

4.3.3 Scope a Context

Bean má stejně jako v přechozích případech definovaný *scope*, tedy dobu své existence. Je spojená s kontextem (*context*). Asociovaný kontext se stará o ži-

votní cyklus *Bean* v daném *scope*. O to se stará běhové prostředí s CDI frameworkem. *Bean* je vytvořena právě jednou v rámci daného *scope*. [19]

4.3.4 Scope přehled

Tabulka 4.3: Přehled scope v CDI Weld

Scope	**Popis**
RequestScoped	Bean je vytvářena při každém HTTP požadavku (request) a zničena při dokončení požadavku. Týká se webových aplikací.
SessionScoped	Bean je spojena s HTTP session. Bean je sdílena po dobu platnosti session v rámci všech HTTP požadavků, které jsou s danou session spojeny. Opět se týká webových aplikací.
ApplicationScoped	Bean je spojena s aplikací. Je vytvořena při startu aplikace a zničena při jejím ukončení. Napříč celou aplikací existuje pouze jedna instance.
ConversationScoped	Bean je spojena s takzvanou konverzací (conversation). Může být dvou typů: *transient* a *long-running*. Výchozí je transient, je vytvořena s JSF requestem. *Transient* konverzace může být konvertována do *long-running* pomocí statické metody *Conversation.begin* a ukončena pomocí *Conversation.end*. Díky tomu je možné držet stav například v různých kartách prohlížeče. Dalším vhodným použitím je držení stavu v různých oknech GUI aplikací.
Dependent	Jedná se o tazvaný pseudoscope. Jde o výchozí *scope*, tedy pokud nemá Bean scope uvedený, použije se právě tento. Vždy dochází k vytvoření nové instance. Ve Spring IoC je analogií pro scope Prototype.

Důležité je poznamenat, že takzvaná kontextová reference (*contextual reference*) na *Bean* není přímou referencí, pokud se nejedná o *scope Dependent*. Místo toho je vytvořen klientský *proxy* objekt. Instance *Dependent scope* je vždy vytvářena jako nová a je na ni odkazováno přímo. CDI umožňuje definovat vlastní *scope*, ale takové použití není doporučováno. [19]

4.4 Porovnání DI frameworků

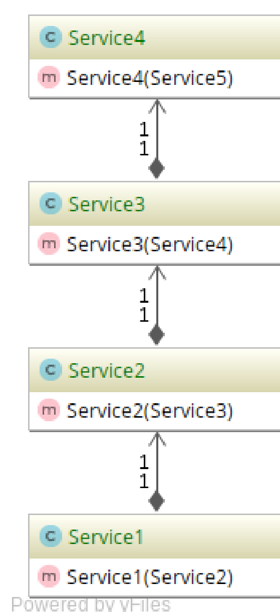
Obecně je velice těžké DI frameworky porovnávat. Vzhledem k tomu, jaké různé možnosti nabízejí, prakticky neexistuje jednotná metrika, jak je porovnat. Nejprve uvádím tabulku porovnání možností, které jsou z mého pohledu důležité pro programátora. V další části jsem se pokusil porovnat inicializační dobu kontejnerů.

Tabulka 4.4: Tabulka porovnání vlastností DI kontejnerů

	Google Guice	**Spring IoC**	**Weld SE**
identifikátor skupiny (*group*) v centrálním repozitáři [22]	*com.google.inject*	*org.springframework*	*org.jboss.weld.se*
identifikátor jména (*name*) v centrálním repozitáři [22]	*guice*	*spring-context*	*weld-se-core*
anotace pro vkládání	*Inject* z balíčku *javax.inject*	anotace *Autowired*, anotace *Inject* z balíčku *javax.inject*	anotace *Inject* z balíčku *javax.inject* (doporučeno), anotace *Inject* z balíčku *com.google.inject*
velikost knihovny	303 *KB*	1.1 *MB*	50 *KB*
umožňuje *constructor injection*	ANO	ANO	ANO
umožňuje *setter injection*	ANO	ANO	ANO
umožňuje *field injection*	ANO	ANO	ANO
konfigurační soubor	potomek třídy *AbstractModule*	třída s anotací *Configuration*	*bean.xml*
automatické vyhledávání tříd	NE	ANO	ANO
podpora *scope* pro webové aplikace (*Request*, *Session*)	ANO	ANO	ANO
možnost definovat vlastní *scope*	ANO	ANO	ANO
integrováno v aplikačních serverech	ANO	NE	NE
průměrná doba inicializace kontejneru s jednou službou	5 *ms*	30 *ms*	43 *ms*

Rozhodl jsem se ve své práci porovnat inicializační dobu frameworku Spring a Weld SE v závislosti na počtu tříd, které jsou na sobě závislé. Závislé třídy tvoří lineární grafovou strukturu.

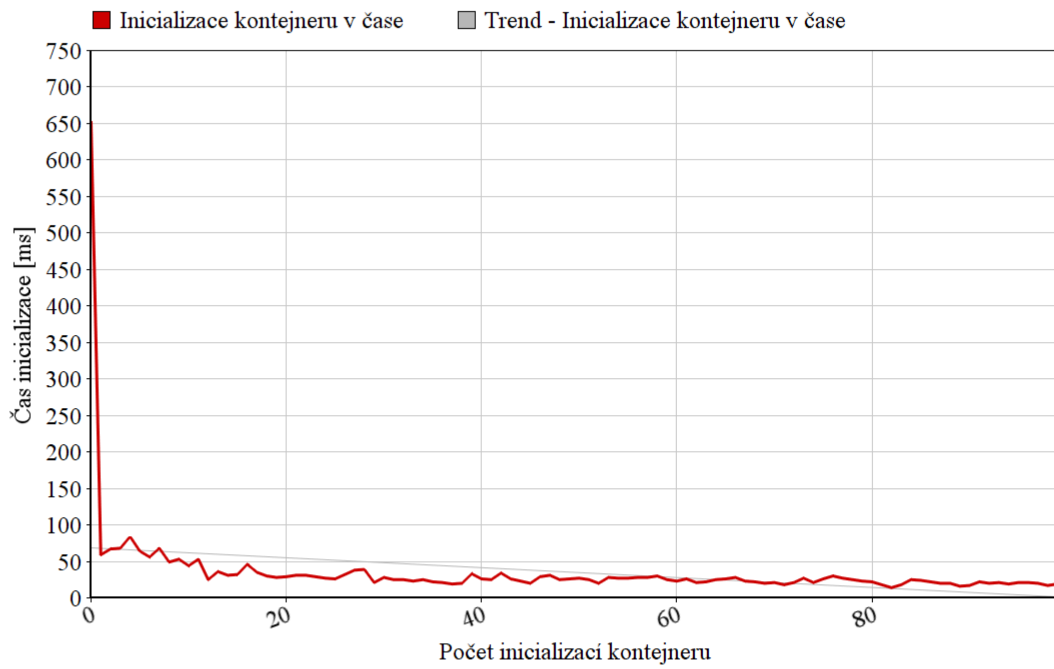
Vytvořil jsem skript, díky kterému je možné vygenerovat určitý počet navzájem lineárně závislých služeb.



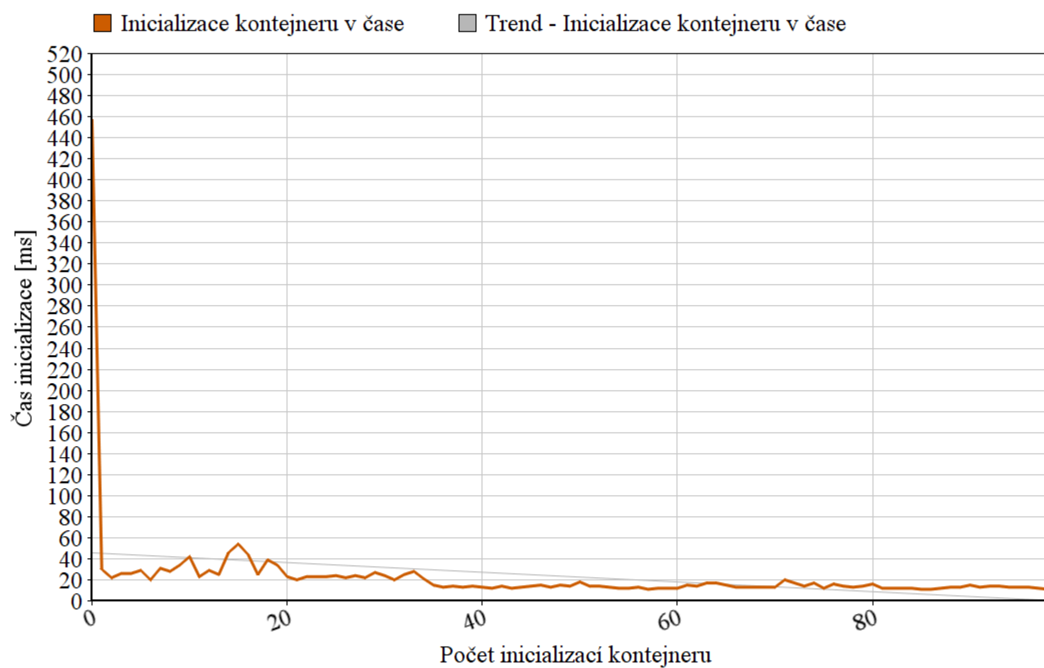
Obrázek 4.1: Graf lineárně závislých služeb

Nejprve jsem měřil čas inicializace jednotlivých kontejnerů. V následujících grafech je vidět inicializační doba pro 100 vytvořených instancí kontejneru. Kontejner je vytvořen, následně je z něho získána určitá služba a poté je ukončen. Měření jsem provedl pro Spring framework a Weld.

Z grafu lze vyčíst zajímavou první hodnotu, která je extrémní. Jedná se výkyv, který s největší pravděpodobností nesouvisí se samotnou inicializací kontejneru, ale s načítáním tříd do paměti. Je vidět, že poté se inicializační čas ustálí přibližně na stejné hodnotě.



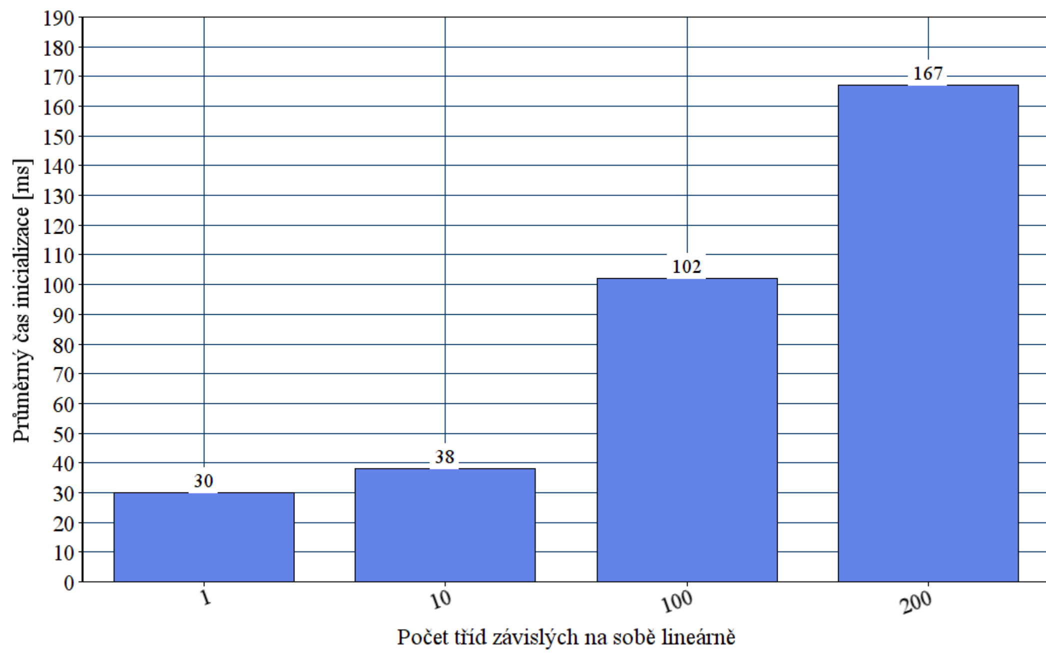
Obrázek 4.2: Inicializace Weld kontejneru závislá na čase



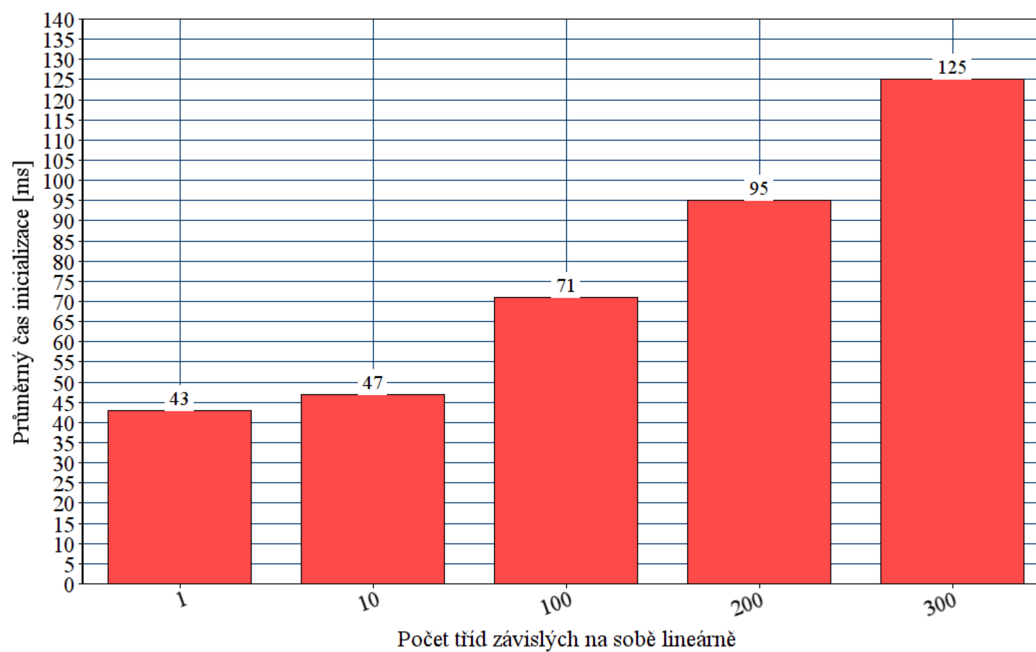
Obrázek 4.3: Inicializace Spring kontejneru závislá na čase

Výsledky inicializace obou frameworků s jednou servisní třídou byly přibližně stejné. Pokusil jsem se tedy porovnat inicializaci v závislosti na vyšším počtu tříd.

V následujících grafech je uvedena inicializační doba v závislosti na počtu lineárně závislých tříd. Je nutné uvést, že třídy byly v jednom balíčku. Přesto je však zajímavé, že při pokusu inicializovat 300 lineárně závislých servisních tříd skončil Spring framework s chybou přetečeného zásobníku *StackOverflowError*.



Obrázek 4.4: Inicializace Spring kontejneru závislá na počtu služeb



Obrázek 4.5: Inicializace Weld kontejneru závislá na počtu služeb

Z předchozích grafů vyplývá, že inicializační doba DI kontejneru v závislosti na počtu tříd má téměř lineární charakter u frameworku Weld. Je však nutné poznamenat, že tato metrika je pouze jedna z téměř neomezeného množství různých kombinací.

Kapitola 5

Vlastní DI kontejner

Vytvořil jsem vlastní lehký DI framework. Vnitřní implementace využívá některé funkce z verze Javy 1.8, pro běh je tedy vyžadována právě tato verze. Aplikace je distribuovaná jako Java knihovna ve formátu archivu JAR. Tato knihovna se dá přilinkovat k projektu. Nejjednodušší je použití moderního sestavovacího nástroje, jako je Gradle nebo Maven [23], který závislosti spravuje. Použití je ukázáno v ukázkové aplikaci. Knihovna se stará stejně jako ostatní DI frameworky o propojení tříd, které jsou na sobě závislé, a o inicializaci jejich instancí.

5.1 Popis rozhraní

DI kontejner udržuje množinu tříd, které spravuje. Aby věděl, která třída může být spravována a která ne, musí být třída anotována *scope* (*Singleton*, *Prototype*). Pokud je požadována třída, která takto anotována není, dojde k vyhození běhové (*runtime*) výjimky.

Rozhraní služby nemá anotaci *scope*, pokud ano, je ignorována; zatímco jeho implementace nebo služba bez rozhraní anotaci *scope* mít musí.

5.1.1 Terminologie

Z počátku návrhu a vývoje jsem chtěl zachovat termín z Java EE a Spring frameworku, tedy *Bean*. Vzhledem k tomu, že mě osobně přijde tento termín matoucí, zvolil jsem název služba (*service*).

5.2 Použité technologie

Jako sestavovací nástroj jsem použil Gradle. Použil jsem knihovnu *Reflections* pro práci s Java reflexí.

5.2.1 Sestavovací nástroj

V adresáři mého projektu je k nalezení tazvaný Gradle *wrapper*, jehož výhodou je, že pro použití nad projektem nemusí mít uživatel žádný Gradle na svém lokákním stroji nainstalovaný. Gradle je mocný nástroj pro automatizaci nad projektem. Definuje takzvané tasky. Task může být například zkompilování zdrojového kódu či puštění testů.

Základním konfiguračním souborem je soubor *build.gradle*, který se nachází v kořenovém adresáři celého projektu. Je definovaný v jazyce Groovy. V tomto souboru je také uveden identifikátor projektu a číslo verze. Závislosti na externích knihovnách jsou definovány v sekci *dependencies*.

DI framework je závislý na knihovně *JUnit*, která je využita pro testování. Dále na knihovně *Reflections*, usnadňující práci s reflexí, a na API *javax.inject*, z kterého jsou využity některé anotace.

```
1 dependencies {
2     testCompile group: 'junit', name: 'junit', version: '4.11'
3     compile group: 'org.reflections', name: 'reflections',
4                     version: '0.9.10'
5     compile group: 'javax.inject', name: 'javax.inject', version: '1'
6 }
```

5.2.2 Java Refexe

Reflexe je schopnost jazyka získat informace o vlastní struktuře za běhu programu. Jedná se o pokročilou techniku, která by se měla využívat velice střídmě. Některé jazyky ani reflexi nepodporují. Jsou však situace, kdy je její použití vhodné nebo dokonce nutné. Tvorba DI frameworku je jeden z typických příkladů.

5.3 Návrh

V následujících odstavcích popíši základní návrh implementace vlastního DI frameworku.

5.3.1 Objektový graf

Jak již bylo řečeno, systém služeb spravovaných kontejnerem tvoří objektový graf. V mém návrhu je struktura objektového grafu nezávislá entita.

Základní struktura objektového grafu v balíčku *cvut.fit.di.graph* je třída *ObjectGraph*. Graf je tvořen uzly reprezentované třídou *ServiceNode*.

5.3.2 Uložiště služeb

Komponenta DI frameworku, která reprezentuje uložiště služeb, se nachází v balíčku *cvut.fit.di.repository.store* ve třítě *ServiceStore*. Samotné služby jsou udržovány jako instance třídy *Service* z balíčku *cvut.fit.di.repository.entity*. Je však potřeba zmínit, že odstranění instancí z paměti již neřídí DI kontejner, ale standardní mechanismus správy paměti v Javě (*Garbage collection*).

5.3.3 Výjimky

Při chybném použití kontejneru dojde k vyhození specifické výjimky. Všechny výjimky jsou potomky takzvané běhové (*runtime*) výjimky. Tuto výjimku není potřeba ošetřovat blokem *try-catch*. Pokud k takové výjimce dojde, ukončí se celá aplikace. [24] V případě inicializace DI kontejneru je takovéto chování zcela žádoucí.

5.3.4 Injektor

Injektor je základní komponenta frameworku. Jedná se o třídu, jejíž rozšíření jsou zodpovědné za vytváření a injektování instancí. Vzhledem k tomu, že se jedná o abstraktní třídu, je nutné vytvořit jejího potomka, který překryje a implementuje metodu *getInstance*. Zpravidla se jedná o rekurzivní metodu. DI kontejner je díky tomu případně možné rozšířit o různé další implementace.

Framework ve svém jádru obsahuje následující rozšiřující třídy, liší se podle toho, jakým způsobem je prováděna injektáž.

Tabulka 5.1: Přehled metod API nad objektovým grafem

Rozšiřující třída	**Popis**
FieldInjector	Injektáž je prováděna do třídních atributů (fields).
SetterInjector	Injektáž je prováděna do setterů.
NotCycleConstructorInjector	Injektáž je prováděna do konstruktoru, následně je ověřeno, zda sestavený objektový graf obsahuje cyklus. Pokud obsahuje, je vyhozena běhová (runtime) výjimka typu *CircularDependencyFoundException*.
CycleConstructorInjector	Injektáž je prováděna do konstruktoru. Objektový graf může obsahovat cykly. Všechny služby však musí implementovat rozhraní.

5.4 Použití

Předpokládaný scénář použití knihovny je - stejně jako například ve frameworku Google Guice - takový, že ve vstupním bodu aplikace, zpravidla v metodě *main*, je vytvořen objekt kontejneru.

```
DIContainer container = new DIContainer();
```

Zavoláním konstruktoru proběhne základní inicializace. Samotné propojení závislostí a sestavení objektového grafu proběhne až ve chvíli, kdy uživatel požádá kontejner o získání závislosti.

Předpokládaným použitím je nějaká startovací třída, v níže uvedeném příkladu je nazvaná *StarterService*. Samotná uživatelská aplikace se spouští například metodou *start* právě na tomto objektu. Instanci z kontejneru lze získat zavoláním metody *getInstance* s parametrem typu třídy, případně rozhraní.

```
MyService srv = container.getInstance(MyService.class);
srv.start();
```

Konkrétní rozšíření třídy injektor je uvedeno jako parametr při vytváření DI kontejneru. Pokud není parametr uveden, je jako výchozí konfigurace použita třída *SetterInjector*.

Definice DI kontejneru podporující field injektáž by vypadala následovně.

```
DIContainer container =
    new DIContainer(new FieldInjector());
```

5.5 Konfigurace kontejneru

Závislosti jsou uspořádány do grafové struktury nazvané objektový graf. Objektový graf je v případě mé implementace prakticky nezávislá entita. Inicializace objektového grafu je možná dvěma způsoby.

1. Ruční konfigurace

2. Automatická konfigurace - při zavolání metody *getInstance* nad kontejnerem

Konfigurace je předávána jako parametr do instance injektoru. Pokud parametr není uveden, je použita automatická konfigurace jako výchozí.

V případě použití výchozího injektoru, tedy není explicitně volán jako parametr, je možné konfiguraci předat v konstruktoru kontejneru.

```
// volani s vychozim injektorem
DIContainer container = new DIContainer(ConfigType.MANUALLY);
```

5.5.1 Ruční konfigurace

Pokud je zvolena ruční konfigurace, je potřeba do kontejneru služby přidat, aby kontejner věděl, které objekty má spravovat. Přidání je možné přes API kontejneru. Je možné přidat jak samotnou službu bez implementace, tak službu s implementací.

Pro přidání je nutné zavolat přetíženou metodu *addService*, která má jako parametr typ třídy, případně jako první parametr rozhraní, které implementuje, a jako druhý parametr samotnou implementaci. Služba musí implementovat právě jedno rozhraní a to uvedené v parametru.

```
1 DIContainer container = new DIContainer(
2 new FieldInjector(ConfigType.MANUALLY));
3 
4 container.getAPI().addService(AInterface.class, AImpl.class);
5 container.getAPI().addService(AConst.class);
```

5.5.2 Automatická konfigurace

Automatická konfigurace znamená, že jsou služby, které má DI kontejner spravovat, hledány automaticky. Hledání probíhá pomocí Java reflexe, která je bohužel relativně časově náročná. Znamená to tedy, že inicializace kontejneru je pomalejší, než při ruční konfiguraci. Na druhou stranu není nutné explicitně vypisovat všechny služby.

5.6 Typy injektáže

Framework poskytuje všechny tři standardní, již dříve zmíněné typy injektáže. Omezení je však takové, že lze použít právě jeden typ injektáže. Typy se nedají se navzájem kombinovat. Tedy uživatel knihovny použije pouze například napříč celou aplikací *setter injection*.

Přestože u ostatních existujících frameworků je mísení různých typů injektáže možné, osobně se domnívám, že by napříč celou aplikací měl být styl jednotný, protože mísení může být pro vývojáře matoucí.

Typ injektování se určí při vytvoření samotného kontejneru, podle instance podtřídy *Injector* (viz sekce Injektor). Při zavolání prázdného konstruktoru je inicializována výchozí hodnota typu *setter injection*.

Použití je velice přímočaré a zcela obdobné ostatním DI frameworkům. Třídní proměnná, setter nebo konstruktor je označen anotací *Inject*, která je součástí *javax.inject* API. V případě konstruktoru může mít třída pouze jeden konstruktor označen anotací *Inject*, pokud je jich uvedeno více, dojde k vyhození výjimky *AmbiguousConstructorException*.

5.7 Setter injection

Proměnná, jež má být do třídy injektována, musí obsahovat veřejnou metodu k nastavení (*setter*), která je označena anotací *Inject*.

```
import javax.inject.Inject;

...

private Service srv;

@Inject
public void setService(Service srv) {
    this.srv = srv;
}
```

5.8 Field injection

Proměnná, která má být do třídy injektována, musí být označena anotací *Inject*. Modifikátor přístupu k proměnné může být libovolný, to znamená, že nemusí být uveden, je veřejný (*public*) nebo soukromý (*private*).

```
import javax.inject.Inject;

...

@Inject
private Service srv;
```

5.9 Constructor injection

Proměnná, která má být do třídy injektována pomocí konstruktoru, musí obsahovat právě jeden konstruktor s anotací *Inject*. Injektovány jsou všechny parametry v takto označeném konstruktoru.

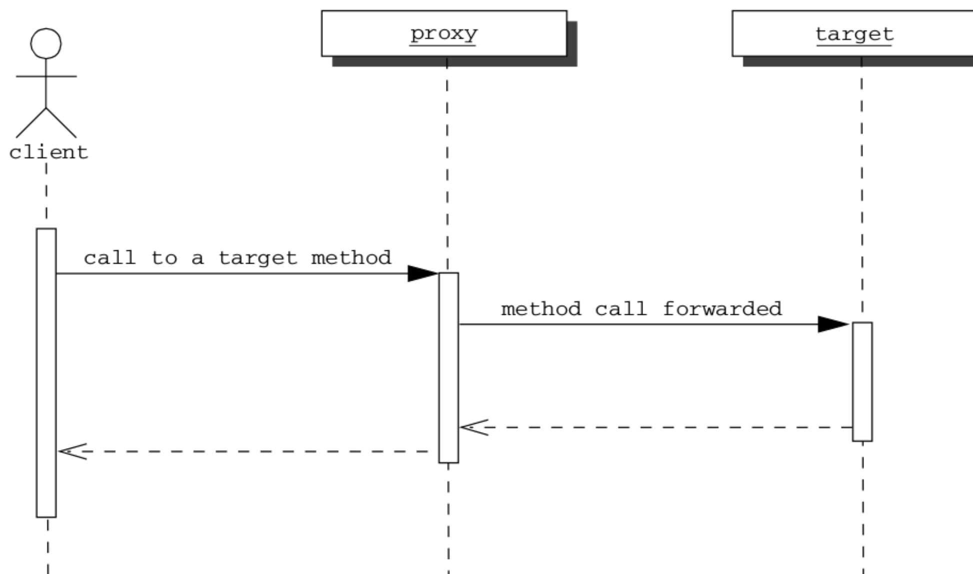
```
import javax.inject.Inject;

...

private Service srv;

@Inject
public A(Service srv) {
    this.srv = srv;
}
```

DI kontejner obsahuje, jak již bylo zmíněno výše, dvě implementace injektoru pro injektáž pomocí konstruktoru.

Jak bylo uvedeno v teoretickém úvodu, při existenci cyklu není možné vytvořit přímo instance pomocí konstruktoru, ale je nutné přidat mezikrok v podobě *proxy* třídy.

Jazyk Java umožňuje vytvoření dynamických *proxy* tříd za běhu aplikace (*runtime*). Použití *dynamické proxy* třídy v mém frameworku je jedním z typických ukázek použití. Dále se používají například při tvorbě testovacích takzvaných *mock* tříd. Další zajímavé použití je například ve frameworku Spring v modulu *Data*, ve kterém stačí vytvořit pouze rozhraní určitého typu třídy a implementace se vytváří dynamicky za běhu aplikace a je vytvářena z názvu metody. [25]

Následující sekvenční diagram ukazuje typické použití *proxy* třídy. Klient volá *proxy* třídu, která volá skutečný objekt. Díky tomu je možné před nebo po zavolání provést určitou proceduru a změnit tím chování.



Obrázek 5.1: Sekvenční diagram použití proxy třídy [2]

V mém DI frameworku používám *proxy* třídy při injektáži pomocí konstruktoru. Díky tomu je možné vytvořit cyklické závislosti. Nutnou podmínkou je, že všechny takové třídy spravované DI kontejnerem musí rozšiřovat právě jedno rozhraní.

Samotná logika vytváření *proxy* tříd je v DI kontejneru implementována v balíčku *cvut.fit.di.proxy*.

Nadefinoval jsem si pomocné generické rozhraní *InstanceSetter*. Toto rozhraní obsahuje právě jednu metodu, která je označena nově vytvořenou anotací *TargetInstanceSetter*.

Tato anotace říká, že se jedná o metodu, jejíž parametr je skutečný objekt.

```
public interface InstanceSetter<T> {

    @TargetInstanceSetter
    void setInstance(T instance);

}
```

Jakousi tovární třídou pro tvorbu *proxy* objektu je třída *ProxyUtil*. Tato třída obsahuje statickou generickou metodu *createProxy*, která jako parametr přijímá rozhraní a vrací proxy objekt daného rozhraní.

Proxy lze získat zavoláním statické metody *newProxyInstance* na třídě *Proxy*. Prvním parametrem je zavaděč tříd (*classloader*), druhým parametrem je pole rozhraní. Důležité je poznamenat, že výsledná *proxy* implementuje všechna zde uvedená rozhraní a díky tomu je možné *proxy* přetypovat a zavolat metodu jiného rozhraní. Tuto skutečnost využívám při nastavení skutečné instance na *proxy*. Třetím parametrem je třída implementující rozhraní *InvocationHandler*. V mém případě se jedná o třídu *ProxyInvocationHandler*.

```
1  public static <T> T createProxy(Class<T> iface) {
2      return (T) Proxy.newProxyInstance(
3              iface.getClassLoader(),
4              new Class<?>[] { iface, InstanceSetter.class }, // vraci
                   objekt co implementuje obe rozhrani
5              new ProxyInvocationHandler());
6  }
```

ProxyInvocationHandler implementuje metodu *invoke*. Při vyvolání libovolné metody na proxy objektu je vyvolána právě tato metoda. Má tři parametry. Prvním parametrem je objekt *proxy*, tedy ten objekt, na kterém byla metoda vyvolána.

Druhým parametrem je instance třídy *Method*, tedy reflexivní třídy poskytující informace o samotné vyvolané metodě. Díky tomu je možné rozhodnout, jaká konkrétní metoda byla volána. V mém případě díky anotaci se rozlišuje, zda byl vyvolán setter nastavující hodnotu skutečného objektu.

Posledním parametrem je pole parametrů vyvolané metody. V mém případě *handler* obsahuje proměnnou *target*, do které je v případě vyvolání speciálního setteru uložena hodnota skutečného objektu. [26]

```
1  public class ProxyInvocationHandler implements InvocationHandler {
2
3      private Object target;
4
5      @Override
6      public Object invoke(Object proxy, Method method, Object[] args)
           throws Throwable {
7
8          if(method.isAnnotationPresent(TargetInstanceSetter.class)) {
9              this.target = args[0];
10             return null;
11         } else {
12             return method.invoke(target, args);
13         }
14     }
15 }
```

5.10 Scope

Vzhledem k tomu, že můj framework je míněný pro použití na desktopové, případně jednoduché serverové aplikace, poskytuje pouze dva *scope*. Typy *scope* jsou pojmenovány stejně jako ve Spring frameworku - *Singleton* a *Prototype*.

Scope se určí příslušnou anotací nad službou. Definice *scope* také určuje, že se jedná o službu spravovanou DI kontejnerem. Anotace *Singleton* je přímo z balíčku *javax.inject* specifikace JSR-330. Anotace *Prototype* je nově vytvořená součástí DI kontejneru v balíčku *cvut.fit.di.anotation.scope*.

Tabulka 5.2: Přehled scope ve vlastním DI frameworku

Scope	**Popis**
Singleton	Napříč aplikací je vždy použita pouze jedna instance.
Prototype	Při požádání daného objektu se vždy vytváří nová instance.

5.11 API kontejneru

Samotný kontejner, tedy třída DI kontejneru, poskytuje pouze dvě veřejné metody. Jednu pro získání instance, *getInstance* a druhou metodu, která vrací instanci API objektoveho grafu, *getAPI*. Třída *ObjectGraphAPI* poskytuje následující metody.

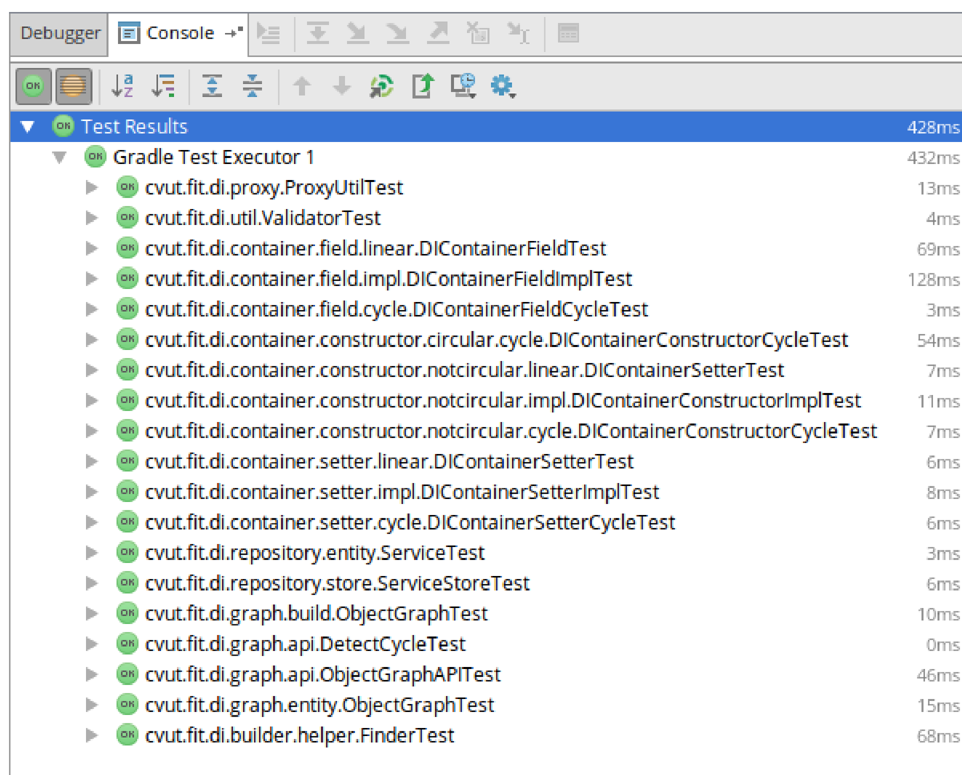
Tabulka 5.3: Přehled metod API nad objektových grafem

Deklarace metody	**Popis**
addService(Class clazz)	Přidá službu bez implementace rozhraní do objektového grafu.
addService(Class clazz)	Přidá službu, která implementuje rozhraní do objektového grafu.
detectConstructorCycle(Class clazz)	Detekuje, zda se v objektovém grafu vyskytuje cyklus v závislostech v konstruktoru.
servicesCount()	Vrátí počet služeb spravovaných DI kontejnerem.
interfacesCount()	Vrátí počet služeb, které jsou implementací rozhraní, spravovaných DI kontejnerem.
servicesWithoutInterfaceCount()	Vrátí počet služeb bez rozhrani spravovaných DI kontejnerem.
allServicesHasInterface()	Ověří, zda všechny služby implementují rozhraní.
prototypesCount()	Vrátí počet služeb se scope prototype spravovaných DI kontejnerem.
singletonsCount()	Vrátí počet služeb se scope singleton spravovaných DI kontejnerem.

5.12 Testování

Projekt je pokryt testy v adresáři *test*. Především se jedná o jednotkové testy, které ověřují správnou funkčnost na testovacích entitách. Testování probíhá pomocí testovacího frameworku JUnit. Spuštění všech testů je možné vyvoláním Gradle tasku *test*.

```
$ ./gradlew test
```



Obrázek 5.2: Grafický výstup testů DI kontejneru

5.13 Možné rozšíření

DI framework je možné dále rozšiřovat. Jak již bylo uvedeno dříve, použití mého DI kontejneru je v tuto chvíli myšleno pro desktopové, případně serverové aplikace. Jako logický krok se jeví rozšíření použití pro webové aplikace tím, že by se do frameworku přidala podpora *session* a *request scope*, případně přidat možnost definice vlastního *scope*. Takovýto zásah by si však pravděpodobně vyžadoval výraznou změnu jádra frameworku.

Další možností je například rozšířit API nad objektovým grafem o nové metody nebo přidat možnost označení služby identifikátorem a pomocí něho ji volat.

Případně definovat a implementovat vlastní injektor, který umožní například kombinaci různých typů injektáže, nebo vytvořit injektor pro cyklickou závislost, jehož třídy nemusí implementovat rozhraní.

Kapitola 6

Ukázková aplikace

Abych demonstroval použití svého DI kontejneru, vytvořil jsem ukázkovou aplikaci registru automobilů a řidičů. Ukázková aplikace je opět vytvořená jako Gradle projekt a je součástí přílohy.

Ukázková aplikace je ve třech různých verzích, tak abych ukázal použití pomocí všech typů injektáže.

6.1 Sestavení a spuštění

Aplikaci je možné zkompilovat pomocí Gradle tasku *build*. Tento task nejprve spustí testy, které jsou v projektu obsažené, a sestaví aplikaci do JAR archivu. Výsledný spustitelný JAR soubor je v adresáři *build/libs*.

Součástí projektu je Gradle *wrapper*. Z toho důvodu, jak již bylo řečeno, není nutné mít nainstalovaný lokální Gradle. Pro sestavení stačí v kořenovém adresáři projektu vyvolat příslušné tasky. Task *clean* nejprve smaže staré soubory z původního buildu.

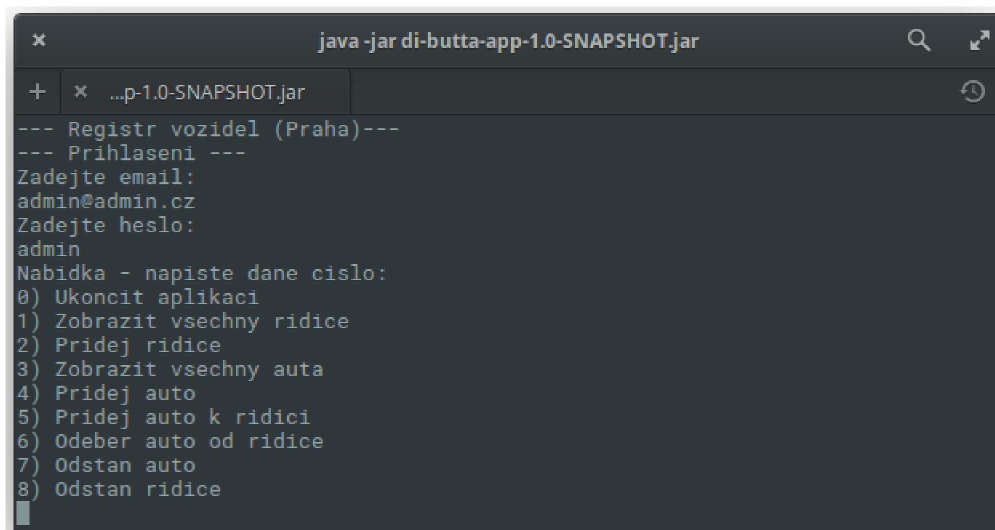
```
1  $ ./gradlew clean build
```

Vzhledem k tomu, že výsledný archiv v sobě obsahuje všechny závislosti, je možné ho jednoduše spustit, jako Java aplikace.

```
1  $ java -jar di-butta-app-1.0.jar
```

6.2 Použití

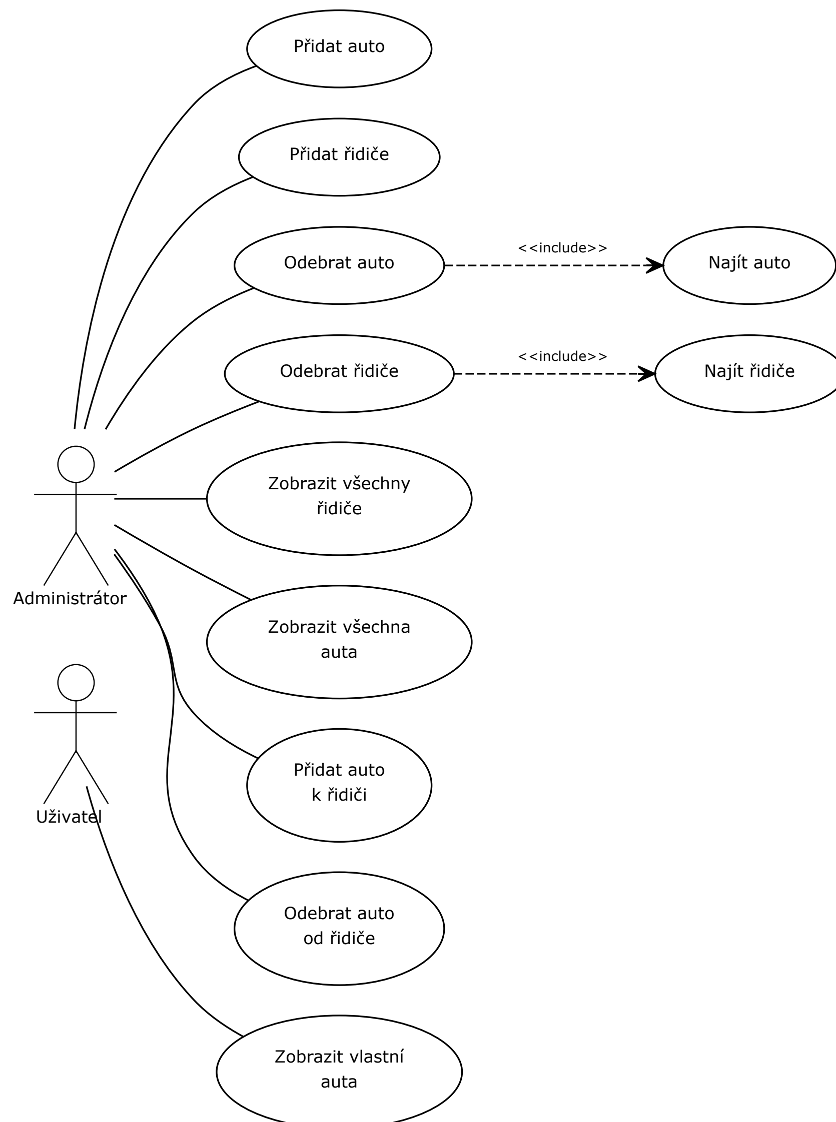
Jedná se o konzolovou aplikaci, jejíž ovládání je pomocí zadávání příkazů do terminálu. Po spuštění aplikace je uživatel vyzván, aby se přihlásil. Poté, co je uživatel autentizován a autorizován, je mu vypsáno menu s možnostmi akcí.



Obrázek 6.1: Administrátorské menu ukázkové aplikace

6.2.1 Diagram užití

Přístup do aplikace má pouze ověřený uživatel. Uživatel má svoji roli, která ho opravňuje vykonávat určité činnosti. V aplikaci existují dvě role - administrátor a uživatel. Administrátor spravuje databáze a má právo přidat či případně odebrat libovolné auto nebo řidiče z databáze. Uživatel má k dispozici pouze výpis automobilů, u kterých je evidován jako řidič.



Obrázek 6.2: Případy užití ukázkové aplikace

6.3 Návrh aplikace

Aplikace je tvořená tak, aby demonstrovala možnosti, které DI framework nabízí. Aplikace ukládá data do databáze. V balíčku *entity* jsou definovány entitní POJO třídy reprezentující auto, řidiče a uživatele.

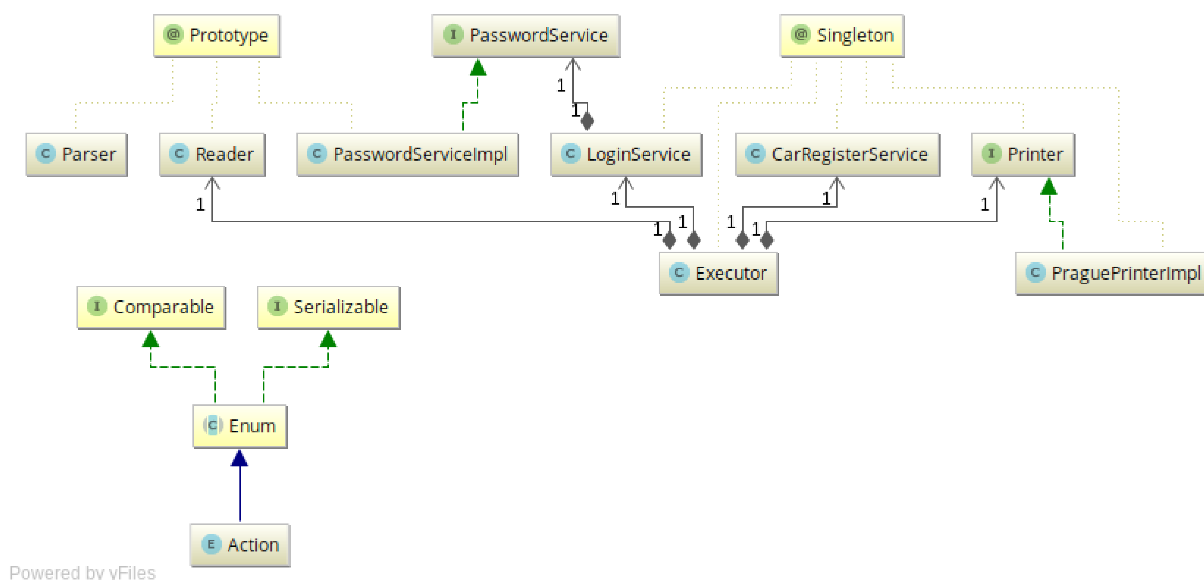
Takzvané *data access objects* (DAO) jsou definovány v balíčku *dao*. DAO třídy jsou využívány ve vyšších servisních vrstvách. Servisní vrstva je definována v balíčku *service*.

Každá servisní třída obsahuje anotaci příslušného *scope* tedy *Singleton* nebo *Prototype*. Takovéto třídy jsou spravovány DI kontejnerem.

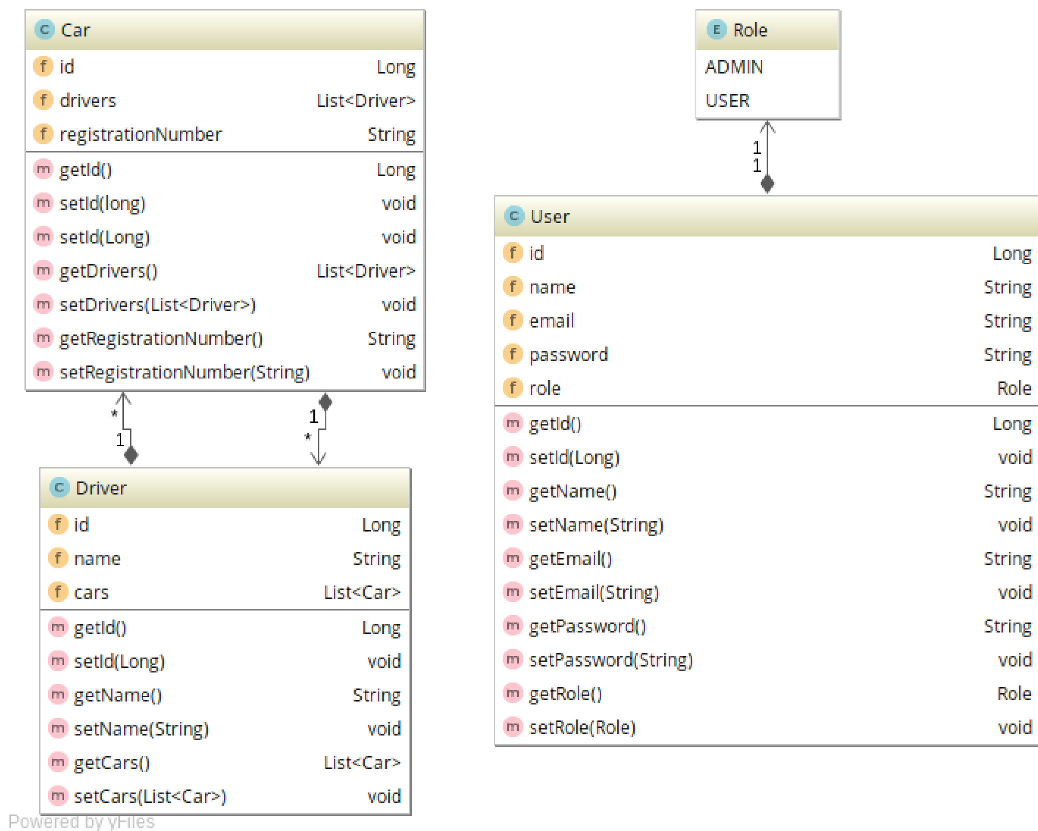
Aplikace obsahuje jak servisní třídy bez implementace, to je například třída *LoginService*, *Executor* nebo *Parser*, tak i třídy, které jsou implementací rozhraní. Příkladem takové třídy je třída *PraguePrinterImpl*, jejíž rozhraní je *Printer* a třída *PasswordServiceImpl*, která implementuje rozhraní *Password-Service*. Ve službách, které je využívají, jsou tyto závislosti definovány pouze rozhraním. Díky tomu je možné velmi jednoduše nahradit implementaci.

Typickým příkladem je třída *PraguPrinterImpl*, která již svým názvem napovídá, že se jedná o implementaci nějaké pražské pobočky. Pokud by aplikace měla být použita v jiném městě, stačí ji nahradit jinou třídou, například *BrnoPrinterImpl*, která implementuje stejné rozhraní.

6.3.1 Diagramy tříd



Obrázek 6.3: Diagram tříd servisní vrstvy ukázkové aplikace



Obrázek 6.4: Diagram tříd entitní vrstvy ukázkové aplikace

6.4 Hibernate

Hibernate je ORM framework, jedná se o implementaci JPA. Umožňuje mapování POJO entitních tříd na databázové tabulky a odstiňuje vývojáře od samotné databáze.

Databázová entita je označena anotací *Entity*. Samotné atributy, které jsou mapovány na sloupce tabulky, jsou označeny anotací *Column*. Entita uživatele v mém projektu vypadá přibližně následovně.

```
1  @Entity
2  public class User implements Serializable {
3
4      @Id
5      @GeneratedValue(generator = "increment")
6      @GenericGenerator(name = "increment", strategy = "increment")
7      private Long id;
8
9      @Column
```

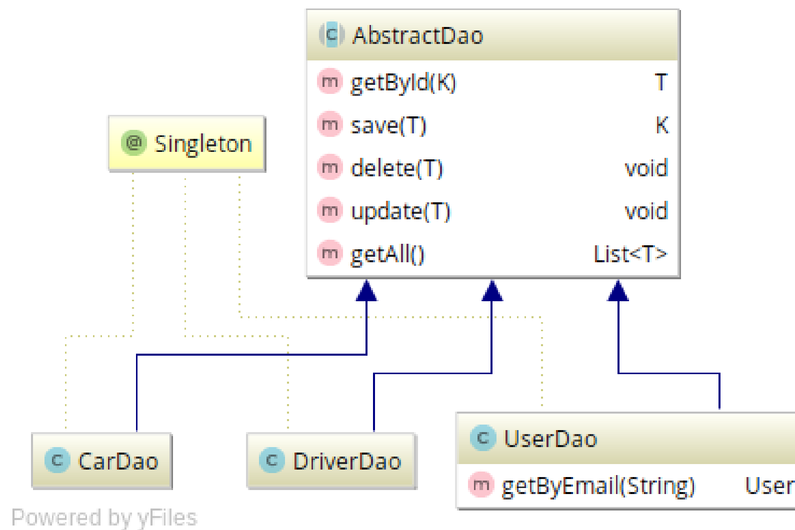
```
10      private String name;
11
12      @Column
13      private String email;
14
15      @Column
16      private String password;
17
18      @Column
19      private Role role;
20
21      public User() {
22          // prazdny konstruktor
23      }
24
25      // gettery a settery
26  }
```

Součástí projektu je konfigurační XML soubor, který se nachází v adresáři *resources* a jmenuje se *hibernate.cfg.xml*. Obsahuje informace potřebné pro připojení do databáze, jako je například databázový ovladač, název databáze nebo databázový uživatel a heslo.

Důležitou součástí je XML direktiva mapující třídy na databázové tabulky. Zajímavá je také definice vlastnosti *hbm2ddl.auto*, pomocí které je možné nastavit, že se databáze vytvoří při spuštění aplikace, pokud již neexistuje.

```
1  <!-- vytvori databazy pokud neexistuje-->
2  <property name="hbm2ddl.auto">update</property>
3
4  <!-- mapovani entit-->
5  <mapping class="entity.User"/>
6  <mapping class="entity.Driver"/>
7  <mapping class="entity.Car"/>
```

Nad entitami jsou definovány takzvané DAO třídy. Vzhledem k tomu, že existují operace společné pro všechny entity, je výhodné vytvořit abstraktního předka. V mém přídě se jedná o generickou třídu *AbstractDao*, která poskytuje základní CRUD operace.



Obrázek 6.5: Diagram DAO tříd ukázkové aplikace

6.5 Vlastní DI framework

Můj DI framework slouží k propojení komponentových tříd. Ukázková aplikace je ve třech různých verzích. První verze používá výchozí hodnoty DI kontejneru, tedy injektáž pomocí setterů a automatické vyhledávání tříd reflexí. Další dvě jsou ukázkou použití *constructor injection* a *field injection*.

Vstupním bodem aplikace je třída *App* s metodou *main*. V této metodě je vytvořen DI kontejner, z kterého je získána instance třídy reprezentující hlavní terminálové okno a na ní je zavolána hlavní programová smyčka pod metodou *mainLoop*.

Po vytvoření kontejneru je ještě pro testovací účely volána statická metoda *initDatabaseData*, které se jako parametr předává DI kontejner. V ní jsou z kontejneru získány DAO objekty, pomocí kterých jsou uloženi uživatelé.

```
1 public static void main(String[] args) {
2     DIContainer container = new DIContainer();
3 
4     initDatabaseData(container);
5 
6     MainBoard mainBoard = container.getInstance(MainBoard.class);
7     mainBoard.mainLoop();
8 }
```

Knihovna je součástí projektu v adresáři *lib*. Tento adresář je v souboru *build.gradle* nastaven jako další zdroj externích knihoven neboli repozitářů. V následující ukázce je vidět příslušné nastavení pro nástroj Gradle.

```
repositories {
    repositories {
        flatDir {
            dirs 'lib'
        }
    }

    mavenCentral()
}
```

Závěr

Téma své práce jsem si vymyslel a zvolil sám, z důvodu, že jsem se s DI frameworky při vývoji softwaru již setkával, ale zároveň pro mě vždy byly velkou neznámou. Díky této práci jsem prostudoval problematiku DI a díky vlastní implementaci jsem pochopil, jak přibližně DI frameworky fungují.

Na začátku své práce jsem představil problematiku spojenou s vytvářením a předáváním závislostí při vývoji softwaru. Na příkladech jsem ukázal, jaké návrhové vzory se používaly před rozšířením DI.

V následující kapitole jsem představil a blíže popsal tři v současné chvíli pravděpodobně nejpoužívanější frameworky na platformě Java. Konkrétně se jedná o Google Guice, IoC v rámci Spring frameworku a Weld, který je referenční implementací CDI Java EE.

Dále jsem se zabýval dobou inicializace Spring frameworku a Weld CDI v závislosti na počtu tříd, které jsou na sobě lineárně závislé. Z grafů se ukázalo, že v tomto konkrétním případě se jako lepší řešení jeví Weld.

V následující kapitole jsem představil vlastní implementaci lehkého DI frameworku, který je myšlený pro použití menších desktopových, případně serverových aplikací. Implementace frameworku je v jazyce Java. Popsal jsem jeho rozhraní a některé zajímavé implementační skutečnosti, jako je například tvorba dynamických proxy tříd. V této kapitole jsem také uvedl možnosti dalšího rozšíření DI frameworku.

V poslední kapitole jsem navrhl a vytvořil ukázkovou aplikaci, na které jsem demonstroval použití vlastního DI kontejneru. Jedná se o konzolovou aplikaci pro registr řidičů a automobilů. Součástí aplikace je perzistentní vrstva operující nad ORM frameworkem Hibernate.

Literatura

[1] Prasanna, D. R.: *Dependency injection*. London: Pearson Education [distributor], 2009, ISBN 9781933988559.

[2] Forman, I. R.: *Java reflection in action*. London: Pearson Education, 2005, ISBN 9781932394184.

[3] Spring Framework: *Dokumentace Spring: The IoC container [online]*. [cit. 2017-03-02] Dostupné z: `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html`.

[4] Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern. leden 2004, [cit. 2017-03-02] Dostupné z: `https://martinfowler.com/articles/injection.html`.

[5] JBoss: *Dokumentace JBoss: What Is Aspect-Oriented Programming? [online]*. [cit. 2017-04-05] Dostupné z: `https://docs.jboss.org/aop/1.0/aspect-framework/userguide/en/html/what.html`.

[6] Oracle: *Dokumentace Google Guice [online]*. [cit. 2017-04-13] Dostupné z: `https://github.com/google/guice/wiki`.

[7] Oracle: *JavaBeans Spec [online]*. [cit. 2017-05-01] Dostupné z: `http://www.oracle.com/technetwork/articles/javaee/spec-136004.html`.

[8] Oracle: *Introduction to Contexts and Dependency Injection for Java EE [online]*. [cit. 2017-04-04] Dostupné z: `http://docs.oracle.com/javaee/7/tutorial/cdi-basic.htm`.

[9] Walls, C.: *Spring in action*. Fourth edition. vydání, ISBN 161729120x.

[10] Koirala, S.: Dagger vs. Guice. prosinec 2013, [cit. 2017-05-06] Dostupné z: `https://www.codeproject.com/Articles/592372/Dependency-Injection-DI-vs-Inversion-of-Control-IO`.

[11] Shores, R.: Oracle Buys Sun. duben 2009, [cit. 2017-02-13] Dostupné z: http://www.oracle.com/us/corporate/press/018363.

[12] *Java Community Process [online]*. Dostupné z: https://www.jcp.org/en/home/index.

[13] *Java Community Process - JSR 330 [online]*. Dostupné z: https://www.jcp.org/en/jsr/detail?id=330.

[14] *Apache Avalon [online]*. [cit. 2017-03-30] Dostupné z: http://avalon.apache.org/closed.html.

[15] Lee, B.: Thanks for the Jolt! březen 2008, [cit. 2017-04-11] Dostupné z: https://opensource.googleblog.com/2008/03/thanks-for-jolt.html.

[16] *Weld specification [online]*. Dostupné z: http://weld.cdi-spec.org.

[17] Gupta, D.: Dagger vs. Guice. září 2015, [cit. 2017-04-30] Dostupné z: https://dig.floatingsun.net/dagger-vs-guice-8c9fbae4712e.

[18] *Repozitář Spring Frameworku [online]*. Dostupné z: https://github.com/spring-projects/spring-framework.

[19] Gupta, A.: *Java EE 7 essentials*. První vydání, ISBN 1449370179.

[20] *Dokumentace jazyka Groovy [online]*. Dostupné z: http://groovy-lang.org/.

[21] *Dokumentace Gradle [online]*. Dostupné z: https://gradle.org/.

[22] *Centrální repozitář knihoven [online]*. Dostupné z: http://mvnrepository.com/.

[23] *Apache Maven [online]*. Dostupné z: https://maven.apache.org/.

[24] *Dokumentace RuntimeException [online]*. [cit. 2017-04-25] Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html.

[25] *Dokumentace Spring Data [online]*. [cit. 2017-05-06] Dostupné z: http://projects.spring.io/spring-data/.

[26] *Dokumentace Dynamic Proxy [online]*. [cit. 2017-05-05] Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html.

Příloha A

Seznam použitých zkratek

OOP Object-oriented programming

DI Dependency injection

DSL Domain-specific language

IoC Inversion of control

GUI Graphical user interface

XML Extensible markup language

POJO Plain old Java object

EE Enterprise Edition

SE Standard Edition

EJB Enterprise JavaBeans

JNDI Java Naming and Directory Interface

JSR Java Specification Requests

JAR Java ARchive

ORM Object-relational mapping

DAO Data access object

HTTP Hypertext Transfer Protocol

CRUD Create, Read, Update, Delete

Příloha B

Obsah přiloženého CD

```
│ readme.txt ........................ popis obsahu CD + návod k použití
├── di-app ........................ adresář se zdrojovým kódem DI kontejneru
├── di-butta-app ......... adresář se zdrojovým kódem ukázkových aplikací
│   ├── constructor ........................ s použitím constructor injection
│   ├── field ........................................ s použitím field injection
│   └── constructor ............................ s použitím setter injection
└── text ............................ zdrojová forma práce ve formátu LaTeX
    └── thesis.pdf ............................ text práce ve formátu PDF
```