Dependency injection

Václav Purchart
Department of Computer Science and Engineering
Czech Technical University
Prague, Czech Republic
purchva1@fel.cvut.cz

Alena Varkočková
Department of Computer Science and Engineering
Czech Technical University
Prague, Czech Republic
varkoale@fel.cvut.cz

***Abstrakt*—Tady bude abstract Tady bude abstract**

***Keywords*—Keywords: Dependency injection, Inversion of control, Dependency injection container**

I. Úvod

Návrhový vzor Dependency Injection (DI) slouží pro snížení závislostí mezi jednotlivými částmi systému. Jeho provedení vychází z obecnějšího návrhového vzoru Inversion of Control (IoC) a tyto dva pojmy jsou často zaměňovány. IoC je obecný princip, kde upřednostňujeme kontrolu zvenčí daného objektu nad situací, kdy si objekt říká o věci sám z vnitřku – máme tak nad objektem větší kontrolu a také tento návrh umožňuje lepší znovupoužitelnost. DI je pojem, který poprvé představil Martin Fowler ve svém článku [zdroj], kde popisuje dvě různé implementace IoC pro snížení závislostí mezi komponentami systému – Dependency Injection a Service Locator. Tento článek částečně vychází z tohoto díla.

II. Princip Dependency injection

Základní myšlenkou DI je oddělení místa použití konkrétní implementace a místa, kde se tato implementace instaluje. Nejdříve si představíme vzorový příklad, který budeme následně upravovat. Vzorový příklad je napsát v Javě, ale tyto pravidla jsou obecně použitelná I pro ostatní objektově orientované jazyky. Máme třídu MovieLister, která slouží pro získání filmů a její konkrétní metodu moviesDirectedBy, která slouží pro získání pouze těch filmů, které mají daného režiséra.

```
class MovieLister...
    public Movie[] moviesDirectedBy(String arg) {
        List allMovies = finder.findAll();
        for (Iterator it = allMovies.iterator(); it.
             hasNext();) {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(arg)) it.
                remove();
        }
        return (Movie[]) allMovies.toArray(new Movie[
             allMovies.size()]);
    }
```

Listing 1. MovieLister naivní implementace

Toto je samozřejmě velmi naivní implementace, ale pro účely našeho příkladu bude postačovat. Metoda uvnitř využívá proměnnou finder, v které je uložena instance jiné třídy, která zajišťuje načítání filmů z úložiště. Nyní se pojďme podívat na způsoby, jak můžeme tuto proměnnou naplnit. Nejnaivnější způsob by byl vytvořit konkrétní instanci přímo při vytváření objektu MovieLister:

```
class MovieLister...
  private ColonDelimitedMovieFinder finder;
  public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt")
        ;
  }
```

Listing 2. Možné naplnění proměnné finder

V tomto provedení ale můžeme do proměnné finder vložit vždy pouze jednu konkréní implementaci, proto zřejmým prvním krokem bude zavedení interface MovieFinder:

```
public interface MovieFinder {
    List findAll();
}
class MovieLister...
  private MovieFinder finder;
  public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt")
        ;
  }
```

Listing 3. Naplnění proměnné finder s využitím interface

Nyní již můžeme do proměnné finder vložit jakoukoli implementaci rozhraní MovieFinder, ale vzhledem k zápisu našeho kódu je do něj opět vždy vkládána jedna konkrétní implentace, takže třída MovieLister má stále dvě závisloti, viz Fig. **??**. Také v současnosti vyžadujeme, aby se soubor, ze kterého budeme filmy načítat, vždy jmenoval movies1.txt, což je velmi silné omezení. Mohli bychom jeho nastavení umožňovat skrze třídu MovieLister, ale to by pak opět podporovalo

svázání s danou konkrétní implementací MovieFinder (ostatní implementace vůbec nemusí pracovat se soubory, ale místo toho například s databází).

Nyní je na čase využít Inversion of Control – pak již nebude třída MovieLister sama rozhodovat o tom jakou implementaci použije, ale bude tuto informaci vyžadovat po okolí. Pomůže nám to take vyřešit problém s předáváním názvu souboru. V našem případě se zabýváme Dependency Injection, takže zvolíme tento typ IoC. Pro implementaci DI můžeme používat následující tři způsoby:

- Constructor Injection
- Setter Injection
- Interface Injection

Všechny tři typy Dependency Injection mají společné to, že díky nim už nejsou „klientské“ třídy závislé na žádné konkrétní implementaci, ale pouze na obecném interface a čekají, že všechny potřebné parametry dostanou zvenku, dostáváme se tedy k situaci, kterou ilustruje Fig. **??**

A. Constructor Injection

Při použití Constructor Injection definujeme závislosti třídy v konstruktoru a tak vyžadujeme, aby bylo předáno vše, co třída potřebuje ke svému životu. Naše třídy tedy budou vypadat nějak takto:

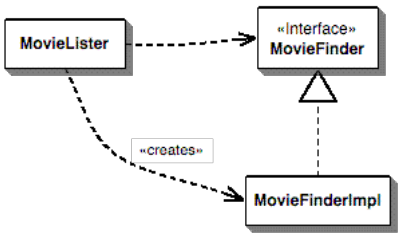
```
class MovieLister...
    public MovieLister(MovieFinder finder) {
        this.finder = finder;
    }

class ColonMovieFinder...
    public ColonDelimitedMovieFinder(String filename) {
        this.filename = filename;
    }
```

Listing 4. MovieLister s použitím Constructor Injection

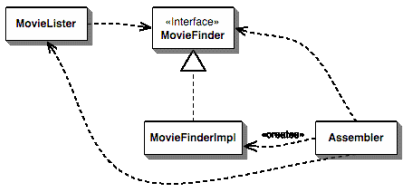
Nyní pokaždé, když budeme chtít instalovat třídu MovieLister, tak jí budeme muset dopředu předat konkrétní implementaci, kterou chceme zrovna použít. Jak je vidět, tak nyní jsme dosáhli oddělení místa použití a samotného skládání objektů, tento kód se tedy může objevit například v nějaké factory metodě:

```
public MovieLister static createMovieLister() {
    MovieFinder finder = (MovieFinder) new
         ColonDelimitedMovieFinder("movies1.txt");
    MovieLister lister = new MovieLister(finder);
    return lister;
}
```

Listing 5. Příklad použití ve factory metodě



Obrázek 1. Vzniklé závislosti při naivní implementaci



Obrázek 2. Závislosti při použití Dependency Injection

B. Setter Injection

Při použití Setter Injection nadefinujeme pro všechny závislosti, které potřebujeme dovnitř třídy předat, settery:

```
class MovieLister...
    private MovieFinder finder;
  public void setFinder(MovieFinder finder) {
    this.finder = finder;
  }

class ColonMovieFinder...
    private String filename;
    public void setFilename(String filename) {
        this.filename = filename;
    }
```

Listing 6. MovieLister a Setter Injection

Kód, který nám připraví MovieLister tedy bude tentokrát vypadat takto:

```
public MovieLister static createMovieLister() {
    ColonDelimitedMovieFinder finder = new
         ColonDelimitedMovieFinder();
    finder.setFilename("movies1.txt");
    MovieLister lister = new MovieLister();
    lister.setFinder(finder);
    return lister;
}
```

Listing 7. MovieLister Setter Injection a factory metoda

```
public interface InjectFinder {
    void injectFinder(MovieFinder finder);
}
```

Listing 8. Interface pro interface injection

C. Interface Injection

U Interface Injection budeme podobně jako u Setter Inection definovat metody pro předání parametrů do našeho objektu, tentokrát ale jejich jméno určíme pomocí interface:

```
class MovieLister implements InjectFinder...
    private MovieFinder finder;
    public void injectFinder(MovieFinder finder) {
        this.finder = finder;
    }
```

Listing 9. Třída MovieLister implementující definované rozhraní

Třída MovieLister pak musí toto rozhraní implementovat:

```
public interface InjectFinder {
    void injectFinder(MovieFinder finder);
}
```

Listing 10. Interface pro interface injection

Identicky bude vypadat vložení filename do naší implementace MovieFinder.

```
public interface InjectFinderFilename {
    void injectFilename (String filename);
}
class ColonDelimitedMovieFinder implements MovieFinder,
     InjectFinderFilename......
    private String filename;
    public void injectFilename(String filename) {
        this.filename = filename;
    }
```

Listing 11. Implementace MovieLister u InterfaceInjection

III. Dependency Injection kontejner

V ukázkách jednotlivých typů DI jsem objekty vždy nakonec sestavil vlastnoručně, tento kód je ale ve obdobný pro všechny případy a tak tuto práci u DI můžeme přenechat tzv. kontejneru, kterému pouze poskytneme konfiguraci. Kontejner poté funguje vlastně tak, že sestaví a instaluje celý strom závislostí aplikace ještě předtím než se samotná aplikace spustí, můžeme si to představit na následujícím příkladu aplikace, mějme nějaký Controller, který bude používat nám již známou třídu MovieLister. Konstruktory jednotlivých tříd tedy budou vypadat nějak takto:

```
Controller (MovieLister lister) {...}
MovieLister(MovieFinder finder) {...}
MovieFinder(String filename) {...}
```

Listing 12. Konstruktory tříd k příkladu

Pro to, abychom mohli spustit (tzn. nejdříve instancovat) náš Controller, tak již musíme předat všechny potřebné parametry – takže budeme vlastně postupovat opačně, zjišťovat závislosti a postupně sestavovat strom. Nejdříve tedy musíme instancovat MovieFinder, kterému předáme do parametru String. Movie Finder pak můžeme vložit do MovieListeru a ten teprve nakonec do Controlleru. Toto je samozřejmě velmi jednoduchý případ, pokud si ho ale představíme na složitější aplikaci, tak samozřejmě vzniklý strom bude daleko rozvětvenější.

A. Dopady Dependency Injection na kód

Dependency Injection by dle mého názoru měl být pouze způsob, jak poskládat dohromady předem připravené třídy a tyto třídy by pokud možno uvnitř nemělo být potřeba měnit (někdy to zařídit ani nemůžeme – především u cizího kódu) a třídy by tak neměly o případném použití nějakého Dependency Injection kontejneru vůbec vědět. Když se podíváme na jednotlivé typy DI, tak největší nároky na podobu dané třídy klade Interface Injection, kde musíme vytvářet rozhraní a ta implementovat. U Setter Injection zase porušujeme čistý návrh objektu tím, že zveřejňujeme settery i pro parametry, které by měly zůstat po počáteční inicializaci neměnné – k tomu se nejlépe hodí Constructor Injection. Je zde velmi názorně patrné, které závislosti potřebuje třída pro svůj běh a tak nemáme vlastně možnost třídu instalovat v nekonzistentní podobě. To zároveň nejlépe odpovídá požadavku na co nejmenší přizpůsobování se DI – třídu, která používá Constructor Injection bychom měli být schopní volně používat i bez jakéhokoli kontejneru a vždy mít zajištěnu její konzistentnost. Naopak je zde několik problémů s využitím Constructor Injection:

- Pokud máme v konstruktoru hodně parametrů, tak může být méně přehledný, zvlášť výrazně je toto vidět u parametrů, které jsou nějakého skalárního typu (String/int), zde se pak můžeme orientovat pouze podle pořadí (u setterů vždy vidíme jméno toho, co nastavujeme).
- Pokud máme více způsobů, jak sestavit daný objekt, tak můžeme poskytnout více konstruktorů, ty se ale můžou lišit pouze počtem a typy parametrů, takže nemusí vždy postačovat – pro ten případ pak můžeme použít factory metody nebo třídy.
- Pokud máme více konstruktorů a ještě použijeme dědičnost, pak musíme vždy ošetřit všechny možnosti konstrukce a přesměrovat je na konstruktory rodiče, což může vést k ještě většímu počtu konstruktorů.

Navzdory zmíněným rizikům použití Constructor Injection bych ji stále stejně jako Martin Fowler preferoval – jak jsem již zmínil, tak její používání má nejmenší vliv na přirozené psaní tříd a také nám vždy zajišťuje konzistenci. Výběr není většinou tak obtížný, protože tvůrci jednotlivých kontejnerů a frameworků v mnoha případech podporují více typů DI, takže je možné v průběhu vývoje bez problémů přestoupit z Constructor Injection např. na Setter Injection, ostatně dalším dobrým důvodem pro podporu více typů DI je možnost integrace cizího kódu do naší aplikace, kde čím více typů je podporováno, tím je větší šance, že nenarazíme na situaci, kterou by daný kontejner nezvládl. Pro začlenění cizího kódu můžeme také využít návrhových vzorů jako např. Adapter nebo Bridge, které vhodně přizpůsobí rozhraní potřebě kontejneru.

B. Injectables vs. Newables

TODO

C. Konfigurace

Konfiguraci kontejneru – tzn. místo, kde určujeme, která konkrétní implementace se na daném místě použije, může mít v základu dvě podoby. První možností je tuto konfiguraci provést přímo pomocí programového kódu, nebo můžeme konfiguraci načítat z konfiguračního souboru a pak jej zpracovávat. Výhodou konfiguračních souborů by mělo být, že by je měl být schopný upravovat i někdo, kdo není programátor, což se ale asi moc často stávat nebude. Výhodou však může být to, že nemusíme znovu kompilovat nějaký kód, pokud je potřeba pouze upravit konfiguraci. [je to opravdu tak? – můj odhad] Výhodnější může být zapsat konfiguraci pomocí kódu, pokud je konfigurace kratší – pak je zbytečné vytvářet konfigurační soubor. Nebo naopak pokud je konfigurace velmi komplexní a obsahuje nějakou podmiňovací logiku, pak není příliš vhodné takto „programovat“ v konfiguračním souboru. Ve výsledku tato volba připadá na konkrétní situaci a preference programátorů. Ostatně pokud kontejner podporuje nastavení pomocí programového kódu, tak není výrazný problém si dopsat vlastní podporu pro nějaký typ konfiguračního souboru – například pokud bychom místo formátu XML chtěli použít YAML.

D. Autowiring

Autowiring se nazývá mechanismus, který by měl ulehčit konfiguraci DI kontejnerů. V různých implementacích nabízí různé možnosti, ale několik jich má společných – především ušetřit programátorovi psaní. Obecně Autowiring funguje tak, že se sám kontejner, případně jeho konfigurace snaží uhodnout určité parametry:

- U konstruktorů může Autowiring pomocí reflexe zjistit, které typy předáváme a pokud se tento typ vyskytuje v konstruktoru pouze jednou a zároveň má v systému jedinou implementaci, tak ji automaticky vybere.
- Podobně se může chovat u properties dané třídy, kde může opět zjistit typ dané proměnné a podle něj uhodnout implementaci.
- U properties se také někdy využívá odhadování podle jména proměnné, kdy pak systém hledá třídu, která má stejné jméno jako daná property.

Autowiring se tedy snaží znovupoužít informace, které jsme již uvedli při psaní kódu a pokud na jejich základě je schopný říci, jakou implementaci vložit, tak tuto informaci nemusíme duplikovat v konfiguraci. Samozřejmě nám ale nepomůže v případech, kdy potřebujeme do třídy injectiovat např. parametry typu string – tam není, kde by danou hodnotu systém mohl zjistit a musíme ji zapsat sami.

E. Konkrétní příklady

K dispozici je hned několik hotových kontejnerů a to pro všechny různé jazyky a jejich použití a implementace jsou odlišné. Liší se například i v konfiguraci, která je buď napsaná klasicky v kódu (v Javě, PHP atd.) případně v XML. Výběr mezi těmito dvěma typy je hodně o osobních preferencích. XML konfigurace ale může být výhodná například v případě, kdy chceme umožnit administrátorům/uživatelům měnit základní funkcionalitu napojením jiných implementací daných služeb a je pro ně daleko příjemnější upravovat XML soubor, než přímo zdrojový kód.

1) Spring: Při použití Dependency Injection kontejneru, který je součástí Spring frameworku je potřeba všechny třídy, které chceme instancovat nadefinovat v XML souboru, přičemž závislosti mezi službami jsou nadefinovány jako „property“ element. Tyto informace Spring poté použije k zavolání správné „setter“ metody – jedná se totiž o kontejner, který používá Setter injection. Příklad konfiguračního souboru:

```
<beans>
  <bean id="AirlineAgency" class="com.dnene.ditutorial.
       common.impl.SimpleAirlineAgency" singleton="true"/
       >
  <bean id="CabAgency" class="com.dnene.ditutorial.common
       .impl.SetterBasedCabAgency" singleton="true">
     <property name="airlineAgency">
      <ref bean="AirlineAgency"/>
     </property>
   </bean>
   <bean id="TripPlanner" class="com.dnene.ditutorial.
        common.impl.SetterBasedTripPlanner" singleton="
        true">
    <property name="airlineAgency">
      <ref bean="AirlineAgency"/>
    </property>
    <property name="cabAgency">
      <ref bean="CabAgency"/>
    </property>
  </bean>
</beans>
```

Listing 13. Konfigurační soubor Spring kontejneru

Při inicializaci kontejneru je potřeba poskytnout název konfiguračního souboru:

```
ClassPathResource res = new ClassPathResource("spring-
    beans.xml"); BeanFactory factory = new
    XmlBeanFactory(res);
```

Listing 14. Inicializace kontejneru

Jednotlivé služby pak v kódu získáváme prostřednictvím id atributí nadefinovaných v XML konfiguračním souboru. Stejně jako u jiných kontejnerů, i zde jsou třídy nainstancovány ve správném pořadí a závislosti jsou do nich vloženy prostřednictvím nadefinovaných setterů.

```
factory.getBean("TripPlanner");
```

Listing 15. Inicializace kontejneru

2) PicoContainer: Pro použití PicoContaineru je potřeba nejprve instancovat DefaultPicoContainer

```
pico = new DefaultPicoContainer();
```

Listing 16. Inicializace kontejneru

Následně je potřeba zaregistrovat jednotlivé třídy:

```
pico.registerComponentImplementation(SimpleAirlineAgency.
    class); pico.registerComponentImplementation(
    ConstructorBasedCabAgency.class); pico.
    registerComponentImplementation(
    ConstructorBasedTripPlanner.class);
```

Listing 17. Registrace tříd v kontejneru

PicoContainer je příkladem Constructor injection (podporuje i setter injection, ale constructor injection je u tohoto kontejneru preferovaná) a v kódu vypadá například takto:

```
public ConstructorBasedCabAgency(AirlineAgency
    airlineAgency) {  this.airlineAgency = airlineAgency
    ; }
```

Listing 18. Použití PicoContainer

Třída ConstructorBasedCabAgency je tedy závislá na rozhraní AirlineAgency a zároveň ConstructorBasedTripPlanner je závislý na CabAgency a také AirlineAgency. PicoContainer rozhoduje závislosti prohlížením jednotlivých konstruktorů, přičemž jednotlivé služby se u něj registrují prostřednictvím implementovaných tříd, klient si ovšem o službu říkají prostřednictvím názvu rozhraní. PicoContainer následně traverzuje celou implementační hierarchií rozhraní a vybere nejvhodnější třídu. U předešlého příkladu je tedy TripPlanner závislý na AirlineAgency a CabAgency a AirlineAgency a CabAgency jsou závislé na TripPlanner. PicoContainer tedy nejprve nainstancuje AirlineAgency, předá referenci konstruktoru CabAgency a poté předá CabAgency i AirlineAgency při vytváření TripPlanneru. Celou tuto sekvenci přitom vyvolá následující řádka kódu: pico.getComponentInstanceOfType(TripPlanner.class);

IV. Závěr

TODO

Reference

[1] Martin Fowler, http://martinfowler.com/articles/injection.html
[2] Spring, http://martinfowler.com/articles/injection.html
[3] Misko Hevery, http://misko.hevery.com/2008/07/08/how-to-think-about-the-new-operator/
[4] Misko Hevery, http://misko.hevery.com/2008/09/30/to-new-or-not-to-new/
[5] Fabien Potencier, http://www.slideshare.net/fabpot/dependency-injectionzendcon2010
[6] Dhananjay Nene, http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection