# Aplikačný rámec Spring

Róbert Novotný

robert.novotny@upjs.sk

# Krátka história Springu

- Rod Johnson
- *Expert One-on-One: J2EE Design and Development (2001)*
  - ako sa vysporiadať s problémami vývoja enterprise aplikácií?
  - dajú sa problémy riešené pomocou J2EE technológií implementovať inak?

# Spring – riešenie problémov J2EE

- Riešenie:

**Spring Framework**


- 2001 – prvá verejná verzia
- 2008 – verzia 2.5
- plán 2009 – verzia 3.0
  - čistá Java 5, prekopanie MVC

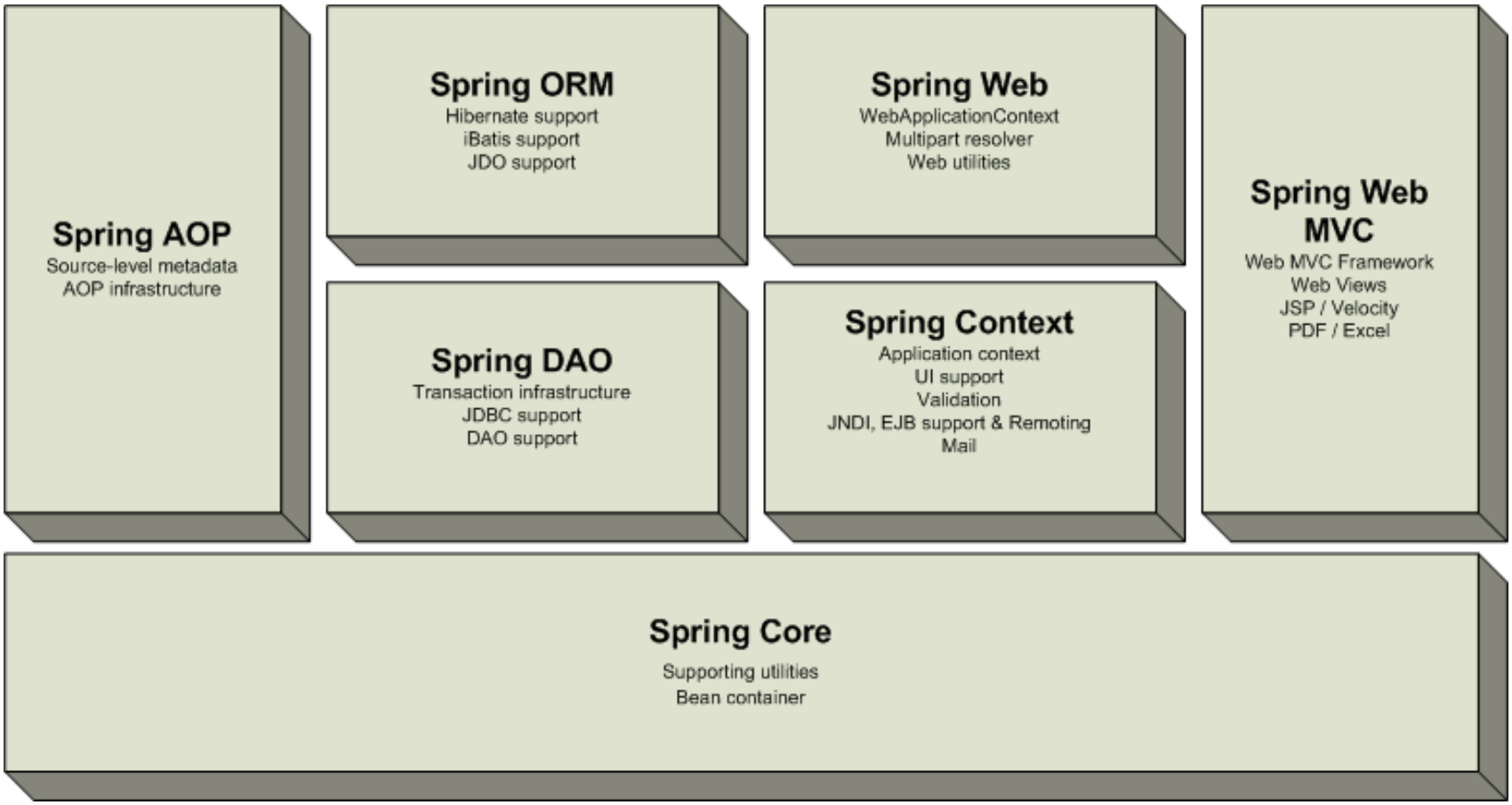
# Čo je vlastne Spring?

- **kontajner**
  - pre **komponenty** tvoriace systém
  - podpora konfigurácie a životného cyklu komponentov
- **aplikačný rámec**
  - vzťahy medzi komponentami
  - deklaratívna konfigurácia
  - ,,švajčiarsky armádny nôž"

# Čo je vlastne Spring?

- **podpora** *dependency injection*
  - asociácie a väzby medzi objektami sú konfigurované zvonku
- **podpora aspektovo orientovaného programovania**
  - nebudeme sa venovať
- **dôraz na malú veľkosť a neintruzívnosť**
  - prítomnosť Springu by mala byť v systéme čo najnenápadnejšia
  - pokiaľ možno, používať POJO

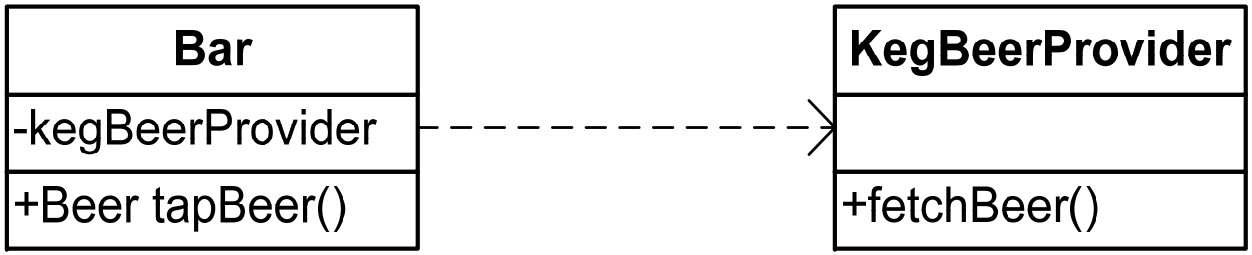
# Základné moduly Springu



# Návrhový vzor *dependency injection*


- pivný bar čapuje pivo
- komponenty systému:
  - pivné sudy ako poskytovateľ piva
  - *pípa* – priamy dodávateľ piva do pohárov

# Návrhový vzor *dependency injection*

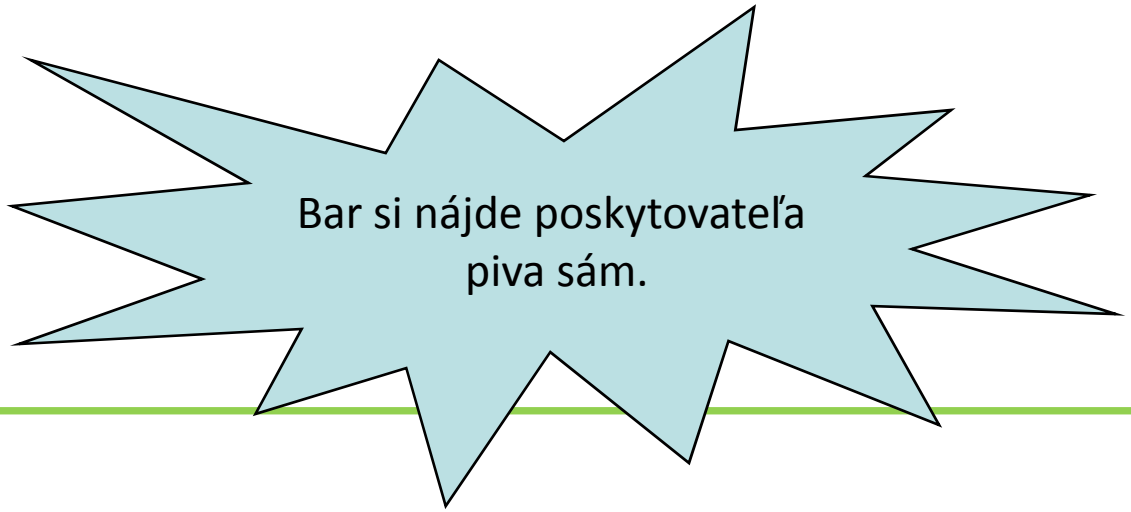



Bar potrebuje na načapovanie piva službu, ktorá mu bude dodávať pivo.

**V implementácii si ju môže nájsť sám.**

# Návrhový vzor *dependency injection*

```
public class Bar {
   private KegBeerProvider provider;
   public Bar() {
      provider = new KegBeerProvider();
   }

   public Beer tapBeer() {
      provider.fetchBeer();
      …
   }
   …
}
```



# Návrhový vzor *dependency injection*

- problémy:
  - krčma chce prejsť na pivo čapované z tanku (väčšia kvalita piva, netreba meniť sudy...)
  - všetky výskyty **KegBeerProvider** je treba nahradiť **TankBeerProvider**-om
- riešenie: *inversion of control*

**Bar si nebude hľadať „službu", čo mu bude dodávať pivo, sám, ale bude mu nastavená zvonku**

# Návrhový vzor *dependency injection*

```
public static void main(...) {
  KegBeerProvider provider = new KegBeerProvider();

  Bar bar = new Bar();
  bar.setProvider(provider);
}
```

- hollywoodsky princíp – nevolajte nás, my vám zavoláme.
- komponent (*bar*), ktorý potrebuje nejakú službu (*pivo*) si ju nenastaví sám, ale nastaví mu ju niekto iný

# Návrhový vzor *dependency injection*

```
public static void main(...) {
  KegBeerProvider provider = new KegBeerProvider();

  Bar bar = new Bar();
  bar.setProvider(provider);
}
```

- Čo ak chceme prejsť na pivo z tanku?

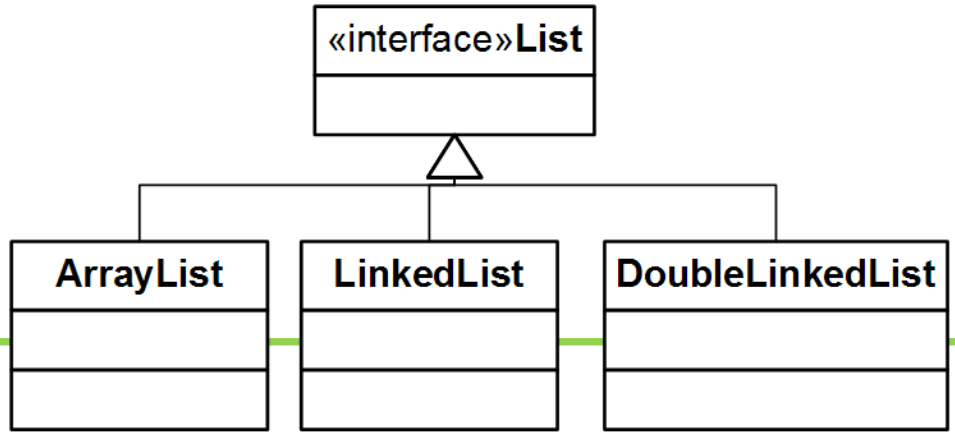
```
public static void main(...) {
  TankBeerProvider provider = new TankBeerProvider();

  Bar bar = new Bar();
  bar.setProvider(provider);
}
```

# Programovanie cez interfejsy

- Čo ak chceme prejsť na pivo z tanku?
- Zásada: pokiaľ je to možné, využívame interfejsy, nie konkrétne implementácie

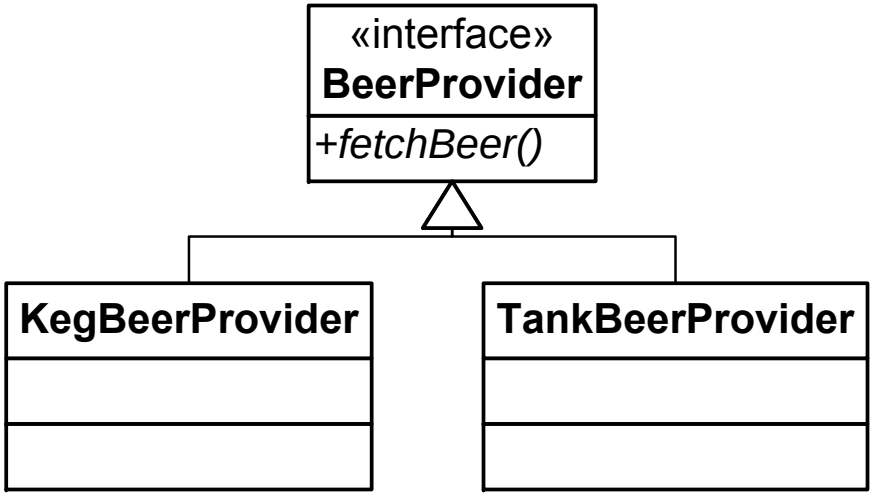
**program to interfaces**

```
List aList = new ArrayList();

AServiceInterface service
          = new ServiceImplementation();
```



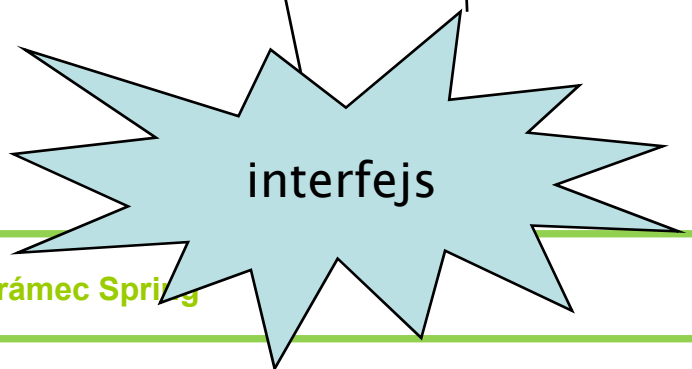
## program to interfaces

- vytvoríme interface *BeerProvider*



# Programovanie cez interfejsy

```
public class Bar {
   private BeerProvider provider;

   public void setBeerProvider(BeerProvider p) {
   …
   }
}

public static void main(...) {
   BeerProvider provider = new KegBeerProvider();

   Bar bar = new Bar();
   bar.setBeerProvider(provider);
}
```




# Spring - aplikačný kontext

- **aplikačný kontext**: reprezentuje množinu vzájomne prepojených komponentov manažovaných kontajnerom
- komponenty sú definované a špecifikované deklaratívne, typicky pomocou **XML**

# Spring - aplikačný kontext

```
<beans>
  <bean id="beerProvider"
        class="sk.beer.KegBeerProvider"
  />

  <bean id="bar" class="sk.beer.Bar">
    <property name="provider" ref="beerProvider" />
  </bean>
</beans>
```

- definujeme 2 komponenty a ich vzájomné prepojenie
- komponenty – odteraz nazývané beanami (*beans*)
- prepájanie - *wiring*

# Základné triedy pre aplikačný kontext

- org.springframework.beans.factory.BeanFactory
  - kontajner pre beany, podpora ich životného cyklu
  - toto je interfejs, existuje viacero užitočných implementácií
- org.springframework.context.ApplicationContext
  - konfigurácia kontajneru na základe XML a pod., načítavanie prostriedkov (konfiguračné súbory, .properties súbory...) z ľubovoľného zdroja, rozosielanie udalostí, internacionalizácia
- org.springframework.context.support.ClassPathXmlApplicationContext
  - načítavanie XML z CLASSPATH, naštartovanie kontajnera

# Príklad použitia kontajneru Spring

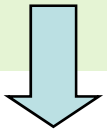
```
ClassPathXmlApplicationContext ctx =
   new ClassPathXmlApplicationContext("applicationContext.xml");

Bar bar = (Bar) ctx.getBean("beerProvider");

ctx.close(); //zatvorenie kontextu
```

- bean vytiahneme z kontextu pomocou **`getBean()`**
- Spring automaticky dodá do baru inštanciu **`beerProvider`**-a

# Spring - aplikačný kontext

- ak chceme prejsť na pivné tanky, stačí zmeniť názov triedy v XML aplikačného kontextu
- programujeme do interfaceov, zmenou triedy sa vymení implementácia
- v kóde netreba zmeniť nič

```
<bean id="beerProvider"
      class="sk.beer.KegBeerProvider"
/>
```



```
<bean id="beerProvider"
      class="sk.beer.TankBeerProvider"
/>
```

# Spring - aplikačný kontext

```
<beans>
  <bean id="beerProvider"
        class="sk.beer.KegBeerProvider"
  />
  <bean id="bar" class="sk.beer.Bar">
    <property name="provider" ref="beerProvider" />
  </bean>
</beans>
```

- Spring vytvorí pre každú triedu ***jeden*** *objekt* (zavolá implicitný konštruktor) a zaregistruje ich v kontajneri
- pre **bar** navyše zavolá **setProvider()**, kde do parametra dosadí inštanciu **KegBeerProvider**-a

# Wiring pevných konštánt

```
<bean id="bar" class="sk.beer.Bar">
    <property name="name" value="U Slováka" />
</bean>
```

- vlastnosti je možné konfigurovať aj pevnými hodnotami
- konfigurácia vykoná `setName("U Slováka")`
- prebieha automatická konverzia typov (reťazce na čísla, booleany...)
- analogická syntax:

```
<bean id="bar" class="sk.beer.Bar">
    <property name="name">
        <value>U Slováka</value>
    </property>
</bean>
```

# Wiring cez parametrické konštruktory

```
public class Bar {
  public Bar(String name, int capacity) {
  …
}
```

- inštancie môžu byť vytvárané aj s využitím parametrických konštruktorov

```
<bean id="bar" class="sk.beer.Bar">
    <constructor-arg value="U Slováka" />
    <constructor-arg value="250" />
</bean>
```

- Spring vytvorí inštanciu beanu bar pomocou vyššie uvedeného konštruktora

# Wiring beanov cez konštruktory

```
public class Bar {
  public Bar(String name, BeerProvider p) {
  …
}
```

- do konštruktorov možno vkladať aj existujúce beany

```
<bean id="bar" class="sk.beer.Bar">
    <constructor-arg value="U Slováka" />
    <constructor-arg ref="beerProvider" />
</bean>
```

- takýto *wiring*: *constructor injection*

# Nejednoznačnosti v konštruktoroch

```
public class Bar {
   public Bar(String name, int capacity) {
   …
   }
   public Bar(String name, String address} {
   …
   }
}
```

- konštruktor, ktorý sa zavolá, sa odvodí z typov argumentov. Niekedy však treba predísť nejednoznačnosti

```
<bean id="bar" class="sk.beer.Bar">
    <constructor-arg value="U Slováka" />
    <constructor-arg type="int" value="250" />
</bean>
```

# Autowiring

```
<beans>
  <bean id="beerProvider"
        class="sk.beer.KegBeerProvider"
  />
  <bean id="bar"
        class="sk.beer.Bar"
        autowire="byName"
  />
</beans>
```

- *autowiring* umožňuje automatické prepájanie beanov bez explicitnej deklarácie
- *byName* – prepájanie na základe zhody ID
- *byType* – zhoda dátových typov
- *constructor* – zhoda dátových typov v konštruktore

# Autowiring podľa mena - byName

- nastaví sa na beane, pre ktorý sa majú nastaviť závislosti automaticky
- prejdú sa všetky *settery* beanu a hľadá sa bean, ktorý má rovnaké ID ako názov *setteru*

```
public class Bar {
   ...
   public void setBeerProvider(BeerProvider p) {
   …
   }
}
```

- hľadá sa bean s ID **beerProvider**
- ak sa nenájde, závislosť nebude nastavená

# Autowiring podľa typu - byType

- prejdú sa všetky *settery* beanu a hľadá sa bean, ktorého trieda zodpovedá dátovému typu v setteri

```
public class Bar {
   ...
   public void setBeerProvider(BeerProvider p) {
   …
   }
}
```

- hľadá sa bean typu **BeerProvider** (čiže trieda implementujúca tento interface)
- ak sa nenájde, nič sa nedeje
- ak existuje viacero beanov s daným typom, hádže sa výnimka

# Autowiring konštruktora - constructor

- autowiring prebieha na parametroch konštruktora na základe typu

```
public class Bar {
   public Bar(BeerProvider p) {
   …
   }
}
```

- hľadá sa bean s ID **beerProvider**
- ten sa vloží do konštruktora ako v prípade `<constructor-arg ...>`
- v prípade nejednoznačnosti nastáva výnimka

# Globálny autowiring

- autowiring možno nastaviť aj na koreňovom elemente **<beans>**
- treba sa zamyslieť nad použitím autowiringu
  - veci sa dejú automaticky
  - vhodné pre malé projekty
  - ale v prípade veľkých projektov / viacerých vývojarov môžu nastať zmätky
    - čo sa autowiruje automaticky?
    - čo sa autowiruje manuálne?
- prípadne zvoliť vhodný konzistentný prístup

# Wiring kolekcií

- existuje špeciálna podpora pre konfiguráciu zoznamov, množín, máp...

```
public class Bar {
  void setMenu(List menuItems);
}
```

```
<bean id="bar" class="sk.beer.Bar">
    <property name="name">
        <list>
            <value>Hoegaarden</value>
            <value>Šariš</value>
        </list>
    </property>
</bean>
```

odkazy na beany pomocou
`<ref bean="idBeanu" />`

# Wiring množín

```
public class Bar {
  void setMenu(Set menuItems);
}
```

```
<bean id="bar" class="sk.beer.Bar">
    <property name="name">
        <set>
            <value>Hoegaarden</value>
            <value>Šariš</value>
        </set>
    </property>
</bean>
```

odkazy na beany pomocou
`<ref bean="idBeanu" />`

# Wiring java.util.Properties

```
public class Bar {
  void setMenuPrices(Properties prices);
}
```

```
<bean id="bar" class="sk.beer.Bar">
  <property name="name">
    <props>
      <prop key="hoegaarden">55</prop>
      <prop key="šariš">30</prop>
    </props>
  </property>
</bean>
```

# Vnorené beany

- v jednoduchých prípadoch je možné referencie medzi beanami uviesť aj priamo

```
<bean id="bar" class="sk.beer.Bar">
  <property name="beerProvider>
    <bean id="beerProvider"
          class="sk.beer.KegBeerProvider"
    />
  </property>
</bean>
```

deklarácia beanu priamo vo vnútri property

# Väzba s kontajnerom

Springovský kontajner poskytuje komponentom podporu pre

- životný cyklus
  - štart
  - stop
- prístup k útrobám Springu
  - prístup k objektu aplikačného kontextu
  - prístup k identifikátoru beanu v kontajneri
  - možnosť rozposielať udalosti

# Životný cyklus - štart

- kontajner môže upovedomiť bean o dokončení jeho inicializácie volaním jeho metódy
  - príklad: štart *embedded* databázy

```
<bean id="bar" class="sk.beer.Bar"
      init-method="open"
/>
```

- potom, čo je vytvorená a inicializovaná inštancia beanu, sa zavolá metóda open( )

# Životný cyklus - stop

- v prípade ukončenia kontajneru možno o tom upovedomiť beany
  - príklad: ak treba uvoľniť databázové pripojenia v beane

```
<bean id="bar" class="sk.beer.Bar"
      destroy-method="close"
/>
```

- pred ukončením kontajnera sa zavolá metóda `close()`
- kontajner treba ukončiť slušne

```
ctx.registerShutdownHook()
```

- zastaví kontajner pri ukončení JVM

# Viditeľnosť vnútorností

- niekedy (zriedka) chceme v beane vidieť do vnútorností Springu. Na to stačí implementovať v beane vhodný interfejs
  - znalosť identifikátora beanu:
    - org.springframework.beans.factory.*BeanNameAware*
    - metóda setBeanName(String)
  - prístup k objektu aplikačného kontextu
    - org.springframework.context.*ApplicationContextAware*
    - metóda setApplicationContext(ApplicationContext)
- trieda beanu je však úzko zviazaná s kontajnerom. Treba zvážiť!

# Zneužitie *dependency injection*

```
public class Bar implements ApplicationContextAware {
  private ApplicationContext ctx;

  private void getBeerProvider() {
    this.beerProvider
      = (BeerProvider) ctx.getBean("beerProvider");
  }
}
```

- takýto kód znamená skoro vždy nepochopenie *dependency injection*
- odkaz na **beerProvider**a môžeme predsa špecifikovať v popisovači aplikačného kontextu!

# Singletony a nie-singletony

- pre každý deklarovaný bean v aplikačnom kontexte vytvorí Spring *jedinú* inštanciu
- pri zavolaní getBean() získate od Springu stále jeden a ten istý objekt
- niekedy to však nie je žiadúce
  - príklad: ThrowAwayController z webovej vrstvy – pre každú požiadavku chceme získať novú inštanciu (inak by si klienti vzájomne prepisovali dáta)

```
<bean id="bar" class="sk.beer.Bar
      singleton="false"
/>
```