Objektové modelování, AD7B36OMO

Návrhové vzory
Tvorba objektů

Část B

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

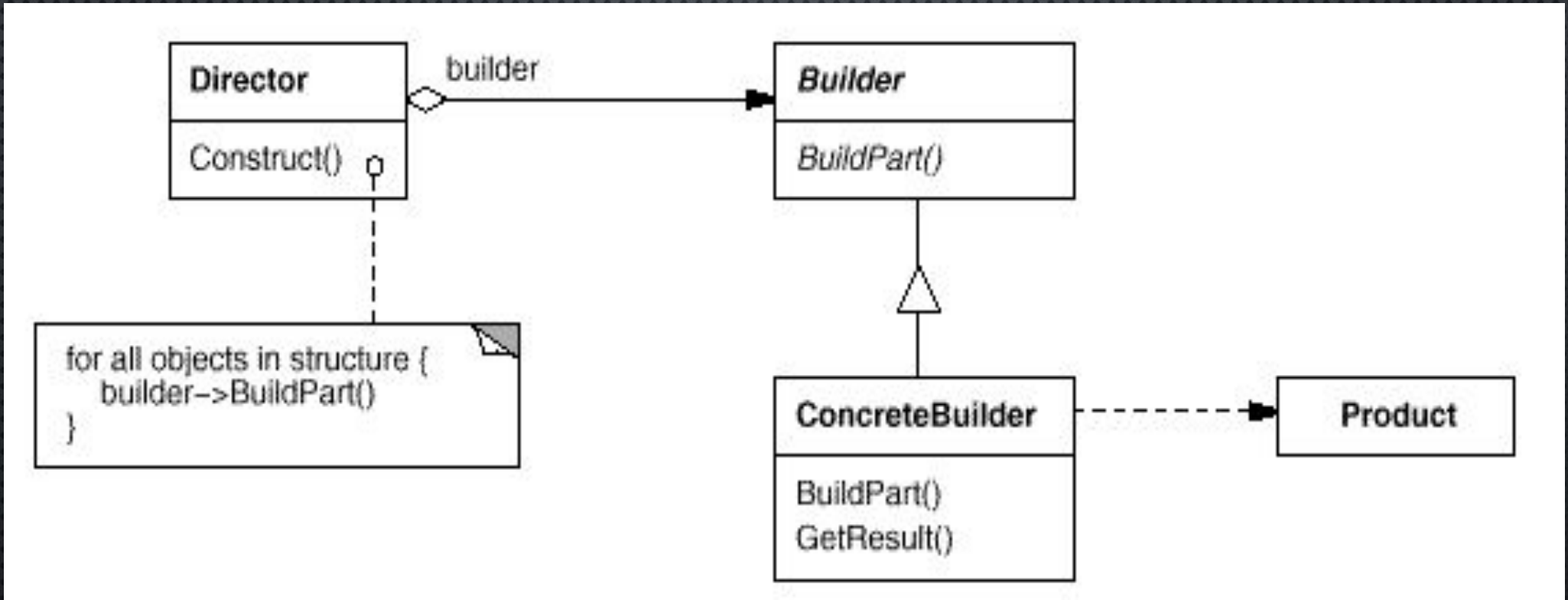
https://edux.feld.cvut.cz/courses/AD7B36OMO

Přednáška je vytvořena na základě původních přednášek autorů: Karel Richta;Pavel Strnad, 2014

VZOR: BUILDER

- Deklarace záměru:
  - oddělení konstrukce složitého objektu od jeho reprezentace
  - pokud má být algoritmus pro vytváření složitého objektu nezávislý na vytváření částí
  - pokud lze objekt reprezentovat různými způsoby
- Motivační příklad:
  - editor dokumentu v RTF by měl umět pracovat s různými reprezentacemi textu (ASCII, TeX, …)

STRUKTURA VZORU BUILDER



Obrázek převzat z [GoF]

PŘÍKLAD: STAVBA DOMU

BUDOVA

Zdroj: http://voho.cz/wiki/informatika/oop/navrhovy-vzor/builder/

- Začneme jednoduchou třídou představující budovu. Bude mít tři atributy: podlahu (floor), zdi (walls) a střechu (roof). Budova bude naším produktem. Různé budovy se od sebe budou lišit podlahou, zdmi a střechou. Důležité je, že klient o postavení budovy požádá řídící objekt (stavbyvedoucího) a o proces jejího vystavění se nestará.

```
public class Building {
  private String floor;
  private String walls;
  private String roof;
  public void setFloor(final String pFloor) { this.floor = pFloor; }
  public void setWalls(final String pWall) { this.walls = pWall; }
  public void setRoof(final String pRoof) { this.roof = pRoof; }
  public void print() { System.out.println(String.format("%s, %s, %s",
          this.floor, this.walls, this.roof));
  }
}
```

PŘÍKLAD: STAVBA DOMU (POKR.)

ROZHRANÍ STAVITELE

- Stavitelé v našem příkladu budou mít několik společných metod, a to na zahájení nové stavby, položení podlahy, stavbu zdí, stavbu střechy a získání výsledné budovy. Proto vytvoříme rozhraní, které budou všichni stavitelé implementovat.

```
public interface Builder {
  void startNew();      /* Zahájí novou stavbu. */
  void buildFloor();    /* Položí podlahu. */
  void buildWalls();    /* Postaví zdi. */
  void buildRoof();     /* Postaví střechu. */
  Building getResult(); /* Vrátí výslednou budovu. */
}
```

PŘÍKLAD: STAVBA DOMU (POKR.)

STAVITELÉ

- Pro náš příklad vytvoříme dvě různé implementace stavitelů: jeden bude stavět budovy levné, druhý luxusní. Klient si pak jednoho z nich vybere a tuto volbu sdělí stavbyvedoucímu. Oba stavitelé budou v našem případě výkonnými objekty, protože zajišťují konkrétní úkony při stavbě.

```
public class CheapBuilder implements Builder {
  private Building result;
  public void startNew() { this.result = new Building(); }
  public void buildFloor() { this.result.setFloor("laminate floor"); }
  public void buildWalls() { this.result.setWalls("panel walls"); }
  public void buildRoof() { this.result.setRoof("wooden roof"); }
  public Building getResult() { return this.result; }
}
```

PŘÍKLAD: STAVBA DOMU (POKR.)

STAVITELÉ - POKRAČOVÁNÍ

```
public class LuxuryBuilder implements Builder {
  private Building result;
  public void startNew() { this.result = new Building(); }
  public void buildFloor() { this.result.setFloor("wooden floor"); }
  public void buildWalls() { this.result.setWalls("brick walls"); }
  public void buildRoof() { this.result.setRoof("shindel roof"); }
  public Building getResult() { return this.result; }
}
```

PŘÍKLAD: STAVBA DOMU (POKR.)

STAVBYVEDOUCÍ

- V roli řídícího objektu zde bude vystupovat stavbyvedoucí. Ten dostane k dispozici stavitele (builder), s jehož pomocí budovu postupně vystaví. Začne podlahou, pokračuje stěnami a nakonec postaví střechu. Po dokončení všech operací vrátí klientovi výslednou budovu.

```
public class Director {
  /* Postaví budovu s použitím zadaného stavitele (parametr builder),
       vrátí výslednou budovu */
   public Building build(final Builder builder) {
     builder.startNew(); // zahájit stavbu nové budovy
     builder.buildFloor(); // položit podlahu
     builder.buildWalls(); // postavit zdi
     builder.buildRoof(); // postavit střechu
     return builder.getResult(); // vrátit výsledek
   }
}
```

PŘÍKLAD: STAVBA DOMU (POKR.)

TESTER

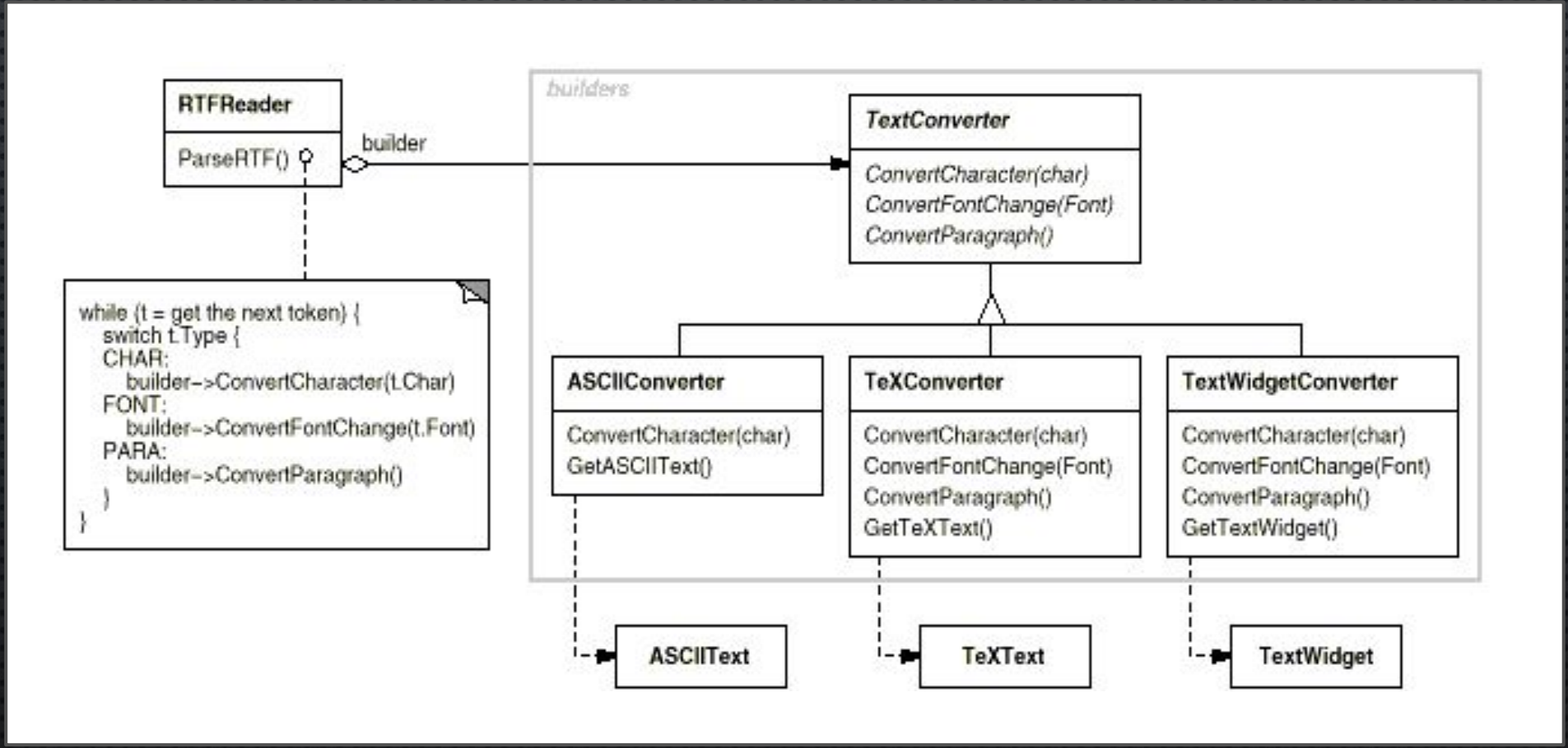
- VZOR POUŽITÍ

```
final Director director = new Director();

// levná budova
final Building cheapBuilding = director.build(new CheapBuilder());
cheapBuilding.print();

// luxusní budova
final Building luxuryBuilding = director.build(new LuxuryBuilder());
luxuryBuilding.print();
```

PŘÍKLAD POUŽITÍ VZORU BUILDER



Obrázek převzat z [GoF]

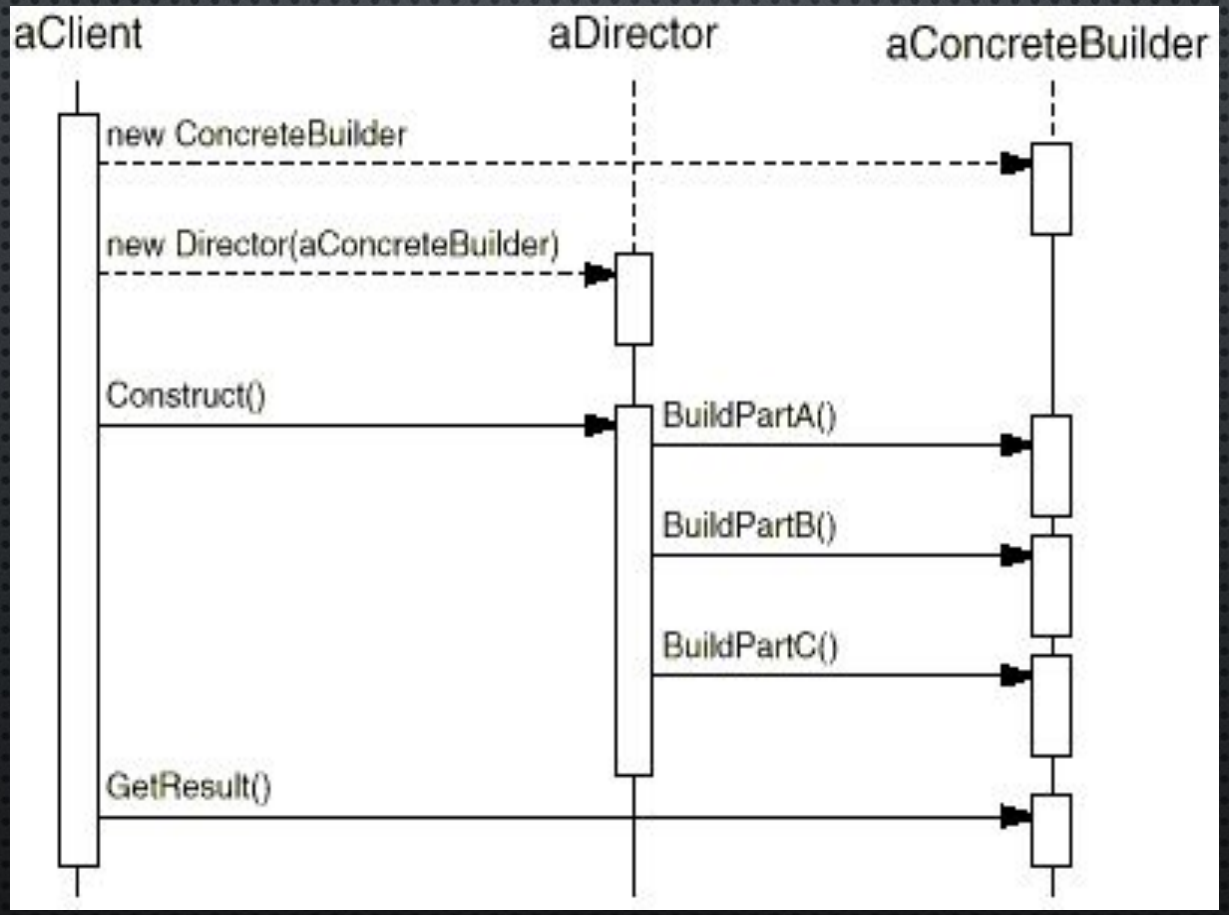
PARTICIPANTI

- Builder (TextConverter)
  - specifikuje abstraktní interface pro vytváření částí objektů typu Product.
- ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)
  - kontruuje a sestavuje části – implementuje interface třídy Builder
  - udržuje si vytvořenou reprezentaci
  - poskytuje služby pro vyzvednutí produktu (GetASCIIText, GetTextWidget, ...)

PARTICIPANTI (POKRAČ.)

- Director (RTFReader)
  - konstruuje objekty pomocí služeb třídy Builder
- Product (ASCIIText, TeXText, TextWidget)
  - reprezentuje složitý objekt, který je konstruován, třída ConcreteBuilder vytváří konkrétní vnitřní reprezentaci
  - zahrnuje třídy pro definici částí a služby pro konstrukci celku

SPOLUPRÁCE OBJEKTŮ PŘI POUŽITÍ BUILDER



Obrázek převzat z [GoF]

BUILDER

- Kdy použít
  - Pokud se vyskytuje konstruktor s mnoha volitenými parametry
  - Pokud kód produkuje různé reprezentace jednoho produktu – stejné kroky vedoucí k cíli, které se liší v detailu
  - Když pracujeme se kompozitními stromovými strukturami
- Výhody
  - Umožňuje postavit produkt krok po kroku
  - Umožní stejnému kódu postavit rozdílné produkty
  - Izoluje složitý kód pro vytváření od business logiky produktu
- Nevýhody
  - Zvyšuje členitost řešení dalšími třídami

INVERSION OF CONTROL (IOC)

- Návrhový princip, ve kterém kód programu je řízený z obecného frameworku
- Jedná se inverzní přístup oproti původnímu přístupu
- Původní klasický přístup – programový kód volá znovupoužitelné knihovny starající se o obecné úlohy
- IoC – obecný framework řídí chod programu a volá specificky vytvořený kód pro řešení konktrétních úloh

Dependency injection (vkládání závislostí)

- **Speciální případ IoC**

- **Definice wikipedia: Vkládání závislostí** (anglicky **Dependency injection** (DI)) je v objektově orientovaném programování technika pro vkládání závislostí mezi jednotlivými komponentami programu tak, aby jedna komponenta mohla používat druhou, aniž by na ni měla v době sestavování programu referenci. Dependency injection lze chápat jako novější název pro Inversion of Control (IoC), ale v užším významu. Jedná se o konkrétní techniku IoC.
- Martin Fowler, http://www.martinfowler.com/articles/injection.html

PRINCIP DEPENDENCE INJECTION

- Pokud chce náš objekt používat služby jiného objektu bez použití DI, pak zodpovídá za celý Jeho životní cyklus, tedy inicializaci apod.
- Při aplikaci DI je ale náš objekt odstíněn od této správy, tu zařizuje dependenc provider. Náš objekt tedy potřebuje už jen referenci na dependence providera.
- DI zahrnuje nejméně tři objekty, které spolu musí spolupracovat. **Konzument**, tedy objekt požadující služby. **Poskytovatel** služby - Poskytovatele těchto služebmu dodá **DI provider**
- DI Provider zodpovídá za celý životní cyklus poskytovatele.
- DI Provider může být implementován několika způsoby, např. jako lokátor služeb, abstraktní továrna, tovární metoda, nebo pomocí nějakého z řady z frameworků, například Spring nebo CDI.

Způsoby vkládání

- Martin Fowler poukazuje na tři různé vzory, jak lze objektu přidat externí referenci jiného objektu.
  - Vkládání rozhraním - externí modul, který je do objektu přidán, implementuje rozhraní, jež objekt očekává v době sestavení programu.
  - Vkládání setter metodou - objekt má setter metodu, pomocí níž lze závislost injektovat.
  - Injekce konstruktorem - závislost je do objektu injektována v parametru konstruktoru již při jeho zrodu.

Výhody a nevýhody DI

- **Výhody**
  - Provider může poskytovat řadu komponent k použití bez zásahu do kódu programu
  - O přetypovávání se stará IoC kontejner
  - Závislosti komponent jsou definovány explicitně, tedy jednodušší orientace v závislostech
  - IoC kontejner lze ve většině případů konfigurovat mimo kód, jednotlivé komponenty je možné využít v jiných programech bez přepisování kódu
- **Nevýhody**
  - Menší přehlednost kódu
  - Náročnější správa kódu

DI FRAMEWORKY

- Spring framework
- Java EE WebBeans(Contexts and Dependency Injection - CDI)
- Google Guice
- PicoContainer
- JBoss Microcontainer

SPRING FRAMEWORK

- Populární open-source framework (označován také jako kontejner) pro vývoj J2EE aplikací
- První verze byla napsána Rodem Johnsonem, který ji vydal v rámci publikace své knihy Expert One-on-One J2EE Design and Development v říjnu 2002. Rod Johnson se ve své knize zabývá vývojem J2EE aplikací a věnuje pozornost problémům, se kterými se programátoři setkávají. V knize je průběžně prezentován kód frameworku, který se nazývá Interface21, a měl by vývoj J2EE aplikací usnadnit. Za pomoci Juergena Hoellera je později framework rozšířen a pod názvem Spring Framework uvolněn jako open-source.

SPRING

- Základním důvodem vzniku Spring Frameworku je usnadnění vývoje enterprise aplikací:
  - Odstranění těsných programových vazeb jednotlivých POJO objektů a vrstev pomocí návrhového vzoru Inversion of Control.
  - Možnost volby implementace (EJB, POJO) business vrstvy pro aplikační architekturu a ne naopak (aby architektura předepisovala implementaci)
  - Řešení různých aplikačních domén bez nutnosti použití EJB, například transakční zpracování, podpora pro remoting business vrstvy formou webových služeb či RMI.
  - Podpora implementace komponent pro přístup k datům, ať již formou přímého JDBC či ORM (object-relation mapping) technologií a nástrojů jako je Hibernate, TopLink, iBatis nebo JDO.
  - Odstranění závislosti na roztroušených konfiguracích a pracného dohledávání jejich významu.
  - Abstrakce vedoucí ke zjednodušenému používání dalších části J2EE, jako například JMS, JMX, JavaMail, JDBC, JCA nebo JNDI.
  - Usnadnění používání a psaní unit testů.
  - Správa a konfigurační management business komponent.

SPRING 3 PŘÍKLAD

- ANOTACE JAVAX.INJECT.NAMED – TAKTO OZNAČENOU BEAN SE SPRING BUDE SNAŽIT INJECTOVAT
- ANOTACE JAVAX.INJECT.INJECT – OZNAČUJE MÍSTO, KDE MÁ SPRING INJECTOVAT BEAN

```
@Named
public class VolvoCarFactory implements CarFactory {
```

```
@Inject CarFactory carFactory;
```