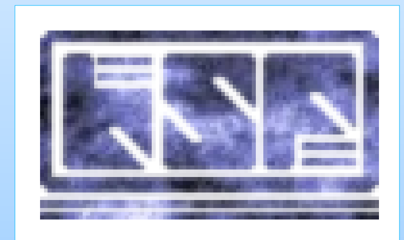
Dependency injection

s nástrojem

PicoContainer

Ing. Roman Nováček, HSF s.r.o.
<novacek@hsf.cz>

4. října 2006



Cíl přednášky

Vysvětlit princip
Inversion of Control (IoC)

Ukázat použití nástroje
PicoContainer

Obsah přednášky

1. úvod
2. Inversion of Control
3. základní terminologie
4. typy IoC komponent
5. Dependency Injection
6. PicoContainer – úvod
7. základní dovednosti
8. speciální funkce
9. kontejner a automatizované testy
10. závěr

Základní terminologie

POJO (Plain Old Java Object) = obyčejná Java třída neimplementující životní cyklus nebo označ. rozhraní

Javabean / Bean = má veřejný implicitní konstruktor, settery / gettery pro nastavení / získání stavu, může reprezentovat data i funkcionalitu

Data Object / Entity Bean / Data Bean / PODO = reprezentuje pouze data, často přetěžuje equals, hashCode a je serializovatelný

Component / Service = reprezentuje funkcionalitu, činnost bývá specifikována rozhraním

IoC Component = neimplementuje logiku pro nastavení stavu (spoléhá na kontejner)

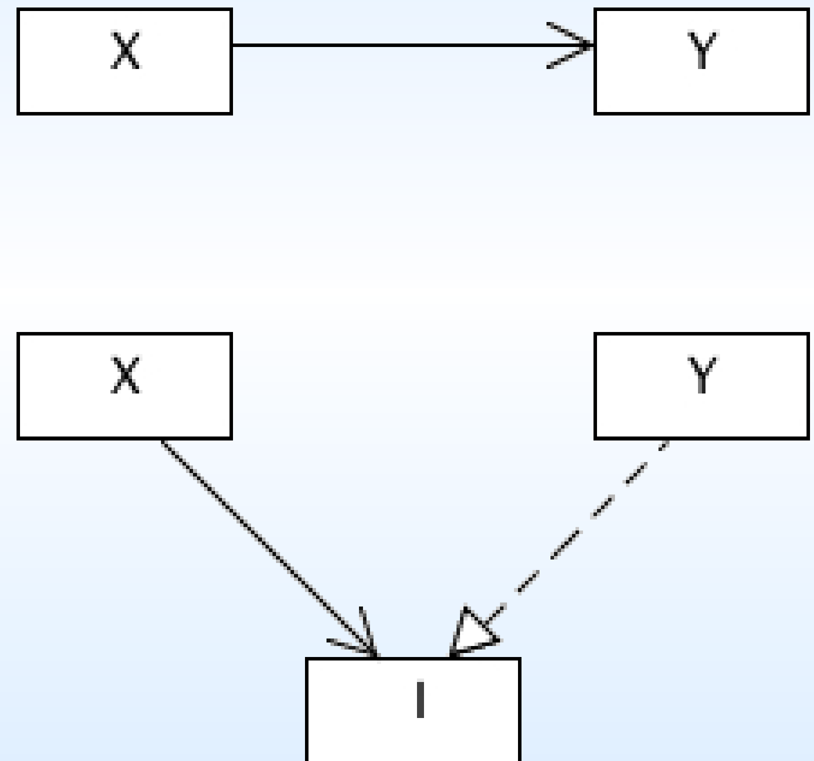
Inversion of Control (IoC)

- důležitý prostředek OOP pro snížení závislostí
- též známé jako *Dependency Inversion Principle* (Martin, 2002)
- další synonymum je *Hollywoodský princip* (Don't call us we'll call you)

Princip IoC (1)

třída X je závislá na třídě Y jestliže:

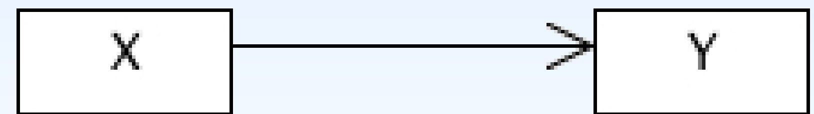
- X vlastní Y (kompozice)
- X je Y (dědičnost)
- X závisí na třídě Z jenž je závislá na Y (nepřímá závislost)



jestliže je X závislá na Y a zároveň Y je závislá na X = **kruhová závislost**

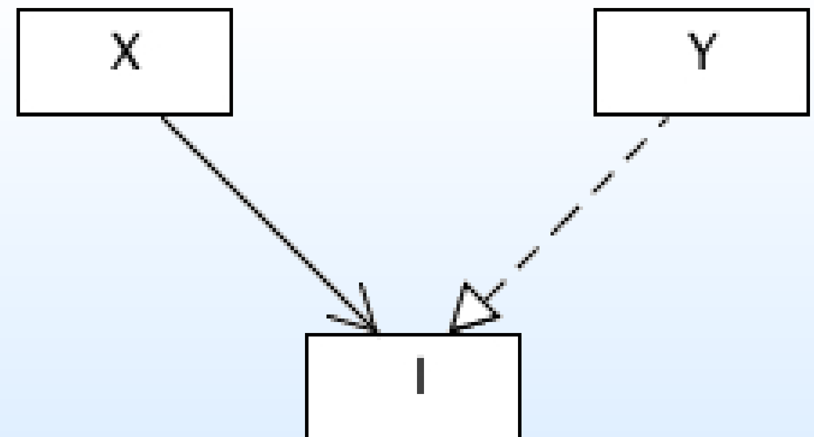
Princip IoC (2)

závislost lze *invertovat* zavedením rozhraní



- X a Y jsou závislé na I
- X nezávisí na Y

odstranění této závislosti se říká *inversion of control (dependency inversion)*



- transformace ruší i všechny nepřímé závislosti X z třídy Y

Typy Inversion of Control

I. **Dependency Injection**
   1. Constructor DI (PicoContainer, Spring, EJB 3.1?)
   2. Setter DI (Spring, PicoContainer, EJB 3.0)
   3. Interface Driven Setter DI (XWork, WebWork2)
   4. Field DI (Plexus)

II. **Dependency Lookup** (koncept vyhledávání objektů ve jmenných registrech – JNDI)

Contextualized Dependency Lookup

```
public interface Orange {
    String doSomething();
}

public class AppleImpl implements Apple, DependencyProvider {

  private Orange orange;

  public void doLookup(DependencyProvider dp) throws Exception {
      this.orange = (Orange) dp.lookup("Orange");
  }

  public String service() {
      return orange.doSomething();
  }
}
```

Dependency Injection (DI)

- je to návrhový vzor (model architektury)
- někdy (nesprávně) označován jako IoC
- ve skutečnosti to je forma implementace IoC
- základním prvkem architektury jsou *komponenty* (závislé objekty obsahující funkcionalitu)
- DI je vzor ve kterém je odpovědnost za vytváření komponent a jejich provázání **odstraněna** z objektů samotných a je umístěna do *Factory*
- DI představuje obrácený princip vytváření a provázání objektů

Dependency Injection – pokračování

- zajišťuje volné vazby mezi objekty
- vede ke snadno testovatelným objektům (test-driven development, mock objekty)
- vyhýbáme se závislostem na implementacích spolupracujících tříd (závislost na rozhraní)

základní typy Dependency Injection:

1. Constructor DI (PicoContainer, Spring)
2. Setter DI (Spring, PicoContainer)
3. Interface Driven Setter DI (XWork)

Setter Dependency Injection

```
public interface Orange {
    String doSomething();
}

public class AppleImpl implements Apple {

    private Orange orange;

    public void setOrange(Orange orange) {
        this.orange = orange;
    }

    public String service() {
        return orange.doSomething();
    }
}
```

Constructor Dependency Injection

```
public interface Orange {
    String doSomething();
}

public class AppleImpl implements Apple {

    private Orange orange;

    public AppleImpl(Orange orange) {
        this.orange = orange;
    }

    public String service() {
        return orange.doSomething();
    }
}
```

Jak komponenta *ne*vypadá?

```
public class AppleImpl implements Apple {

    private Orange orange;

    public AppleImpl() {
        this.orange = new OrangeImpl();
    }
    // other methods
}

public class AppleImpl implements Apple {

    private Orange orange = OrangeFactory.getOrange();

    public AppleImpl() {
    }
    // other methods
}
```

Co je PicoContainer?

= *lehký kontejner* na komponenty

- open source framework usnadňující Dependency Injection
- aktivně jej vyvíjí od roku 2003 skupina asi 12 vývojářů, z nichž většina pracuje ve firmě ThoughtWorks
- nejnovější verze 1.2 je z 17. ledna 2006

Co PicoContainer dělá?

- **Dependecy Injection**
- je neintruzivní (komponenty nemusí nic implementovat)
- podporuje životní cyklus komponent
- umožňuje snadnou konfigurovatelnost komponent a tím zároveň i snadnou testovatelnost aplikace

Jak se PicoContainer používá?

- komponenty jsou obyčejné Java třídy
- kontejner se pro vkládání komponent chová jako inteligentní hash mapa (zná typy komponent)
- pokud je vyžadován životní cyklus komponent, je možné implementovat jednoduchý interface z API nebo lze použít vlastní životní cyklus

Vkládání komponent

- metoda **`registerComponentImplementation`**
  1. `Object componentKey` ... klíč komponenty (nepovinný)
  2. `Class componentImplementation` ... třída implementace
  3. `Parameter[] parameters` ... parametry konstruktoru (nepovinný)
- metoda **`registerComponentInstance`**
  1. `Object componentKey` ... klíč komponenty (nepovinný)
  2. `Object componentInstance` ... instance komponenty
- metoda **`registerComponent`**
  1. `ComponentAdapter componentAdapter`

Vyjímání komponent:

- `unregisterComponent(Object componentKey);`
- `unregisterComponentByInstance(Object componentInstance);`

Vkládání komponent – příklad

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

- metoda **registerComponentImplementation**:

```
pico.registerComponentImplementation(OrangeImpl.class);
pico.registerComponentImplementation(Orange.class, OrangeImpl.class);
```

- metoda **registerComponentInstance**:

```
pico.registerComponentInstance(new AppleImpl());
pico.registerComponentInstance(Apple.class, new AppleImpl());
```

- metoda **registerComponent**:

```
pico.registerComponent(
    new CachingComponentAdapter(
        new ConstructorInjectionComponentAdapter(
                Juicer.class, Juicer.class, parameters)));
```

Získávání instancí

- samotná instance:

```
Apple apple = pico.getComponentInstance(Apple.class);
```

- kolekce objektů:

```
List<Apple> apples =
   pico.getComponentInstancesOfType(Apple.class);
```

- všechny registrované instance:

```
List components = pico.getComponentInstances();
```

Adaptéry komponent

- PicoContainer pro každou registrovanou komponentu vytváří adaptér, který poskytuje její instanci
- základní typy adaptérů:
  - **`ConstructorInjectionComponentAdapter`**
  - `SetterInjectionComponentAdapter`
- pomocné adaptéry:
  - `CachingComponentAdapter`
  - `SynchronizedComponentAdapter`
  - `ImplementationHidingComponentAdapter`

```
ConstructorInjectionComponentAdapter(
   Object componentKey,
   Class componentImplementation,
   Parameter[] parameters,
   boolean allowNonPublicClasses,
   ComponentMonitor componentMonitor,
   LifecycleStrategy lifecycleStrategy
);
```

Konfigurace komponent

```
public class Foo {
    public Foo(DependantComp dComp, String foo, Integer bar) {
    }
}

Parameter[] fooParams = new Parameter[] {
    new ComponentParameter(),
    new ConstantParameter("foo"),
    new ConstantParameter(Integer.valueOf(BAR_VALUE))
};

MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponent(DefaultDependantComp.class);
pico.registerComponent(Foo.class, fooParams);
```

Konfigurační pseudo-komponenta

```
public class Foo {
    public Foo(DependantComp dComp, FooConfig fooConfig) {
    }
}

public class FooConfig {
    FooConfig(String name, int bar);
    String getFooName();
    int getBarNumber();
}

MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(DefaultDependantComp.class);
pico.registerComponentImplementation(Foo.class);
pico.registerComponentInstance(new FooConfig(...));
```

PicoContainer - slogan

Očekával jsem nový přístup
a vše co jsem dostal
byl prachobyčejný konstruktor!

Životní cyklus komponent

```
public class Peach implements Startable, Disposable {

    public Peach() {
    }

    public void start() {}
    public void stop() {}
    public void dispose() {}
}


MutablePicoContainer pico = new DefaultPicoContainer();
pico.start();
pico.stop();
pico.dispose();
```

Modifikace životního cyklu

```
PicoContainer pico = ...

QuantumLeapable quantumLeapable = (QuantumLeapable)
    Multicaster.object(pico, true, new StandardProxyFactory());

// This will call leap() on all QuantumLeapable components inside
pico.quantumLeapable.leap();
```

Kolekce a mapy jako parametry komponent (1)

```
public class Bowl {
    private final TreeMap fishes;
    private final Map cods;

    public Bowl(TreeMap fishes, Map cods) {
        this.fishes = fishes;
        this.cods = cods;
    }

    public Map getFishes() {
        return fishes;
    }

    public Map getCods() {
        return cods;
    }
}
```

Kolekce a mapy jako parametry komponent (2)

```java
public interface Fish {
}

public class Cod implements Fish {
}

public class Shark implements Fish {
}

pico.registerComponentImplementation("Shark", Shark.class);
pico.registerComponentImplementation("Cod", Cod.class);
pico.registerComponentImplementation(Bowl.class, Bowl.class,
    new Parameter[]{
        new ComponentParameter(Fish.class, false),
        new ComponentParameter(Cod.class, false)
});
```

Monitor komponenty – ukázka

```
public class MyMonitor extends DefaultComponentMonitor {
  private boolan instatiated = false;
  public void instantiated(Constructor const, long duration) {
      instantiated = true;
  }
  public boolan wasInstantiated() {
      return instantiated;
  }
}

serverConnectorMonitor = new InstantiatedMonitor();

pico.registerComponent(
    new CachingComponentAdapter(
        new ConstructorInjectionComponentAdapter(
            ServerConnector.class, ServerConnectorImpl.class,
            null, false,
            serverConnectorMonitor)));
```

PicoContainer a testy

```
public void testCocktailWithVodkaIsAlcoholic() {

  DefaultPicoContainer container = new DefaultPicoContainer();
  container.registerComponentImplementation(Banana.class);
  container.registerComponentImplementation(Vanilla.class);
  container.registerComponentImplementation(Vodka.class);
  container.registerComponentImplementation(Cocktail.class);

  Cocktail cocktail = (Cocktail)
     container.getComponentInstance(Cocktail.class);

  assertTrue(cocktail.isAlcoholic());
}
```

Co je Mock objekt?

= speciální verze objektu určena pro testování

- má stejné chování jako skutečný objekt, ale je to jen **atrapa** bez vazeb na další objekty
- před vlastním provedením testu se definuje ***očekávané*** chovaní všech zástupných (Mock) objektů, které se podílejí na testované funkčnosti
- vhodné zejména pro *test-driven development*

PicoContainer a Mock objekty (1)

```
public void testCocktailWithVodkaIsAlcoholic() {

  Banana banana = createMockBanana();
  Vanilla vanilla = createMockVanilla();
  Vodka vodka = createMockVodka();

  // set expectations on banana, vanilla and vodka here

  DefaultPicoContainer container = new DefaultPicoContainer();
  container.registerComponentInstance(Banana.class, banana);
  container.registerComponentInstance(Vanilla.class, vanilla);
  container.registerComponentInstance(Vodka.class, vodka);
  container.registerComponentInstance(Cocktail.class);

  Cocktail cocktail = (Cocktail)
   container.getComponentInstance(Cocktail.class);

  assertTrue(cocktail.isAlcoholic());
}
```

PicoContainer a Mock objekty (2)

```
public void testCocktailWithVodkaIsAlcoholic() {

  Banana banana = createMockBanana();
  Vanilla vanilla = createMockVanilla();
  Vodka vodka = createMockVodka();

  // set expectations on banana, vanilla and vodka here

  Cocktail cocktail = new Cocktail(banana, vanilla, vodka);
  assertTrue(cocktail.isAlcoholic());

  // verify expectations on banana, vanilla and vodka here
}
```

Programování do rozhraní

- pomáhá minimalizovat vazby částí systému
- usnadňuje spolupráci vývojářů (rozhraní říká vše co potřebuje uživatel vědět)
- vazby přes interface jsou obecnější
- použití rozhraní je nejvhodnější způsob vytváření Mock objektů
- rozhraní jsou vhodné pro kontejnery na principu *Inversion of Control* (PicoContainer, Spring)
- **nevýhody:** více kódu (2x více souborů), kód je duplicitní (porušuje OnceAndOnlyOnce), problém při ladění (znesnadňuje trasování)

Shrnutí

- Dependency Injection framework pomáhá vytvářet čistější kód
- vyvíjíme-li aplikaci skládající se z více komponent je velmi vhodné nasadit kontejner na principu *Inversion of Control* (snadná záměna netestovaných částí systému za Mock objekty)

Odkazy

Trocha teorie

- http://en.wikipedia.org/wiki/Inversion_of_control
- http://en.wikipedia.org/wiki/Dependency_injection
- http://www.martinfowler.com/articles/injection.html

PicoContainer

- http://www.picocontainer.org/
- http://www.picocontainer.org/Five+minute+introduction
- http://docs.codehaus.org/display/PICO/User+Documentation

Další lehké kontejnery

- http://plexus.codehaus.org
- http://www.springframework.org/
- http://jakarta.apache.org/hivemind/
- http://excalibur.apache.org/
- http://wiki.apache.org/excalibur/WhichContainer
- http://www.opensymphony.com/xwork/
- http://java-source.net/open-source/containers

Konec