Návrhové vzory

- V doterajších príkladoch sa vyskytovali viaceré návrhové vzory
- Prečo je dobré poznať návrhové vzory?
  - Ušetria čas
  - Zvyšujú kvalitu
  - Zlepšujú porozumenie pre vývojárov opravujúcich alebo rozširujúcich zdrojový kód
- História
  - Návrhové vzory získali popularitu po uvedení knihy „Elements of Reusable Object-Oriented Software“ v 1994 skupinou "Gang of Four" (GoF): Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides na konferencii OOPSLA v Portlande
- Čo je to návrhový vzor?
  - Návrhový vzor je popis alebo šablóna ako riešiť nejaký typ softvérového problému bez konkrétne špecifikácie aplikačných tried a objektov. Návrhový vzorec nie je hotový návrh, ale môže byť jednoducho transformovaný do kódu.
  - Doménou návrhových vzorcov sú abstraktné triedy, objekty a relácie medzi nimi .
  - Nie všetky softvérové vzory sú návrhové vzory.
    - Napríklad, algoritmy riešia výpočtové problémy a nie návrhové problémy.

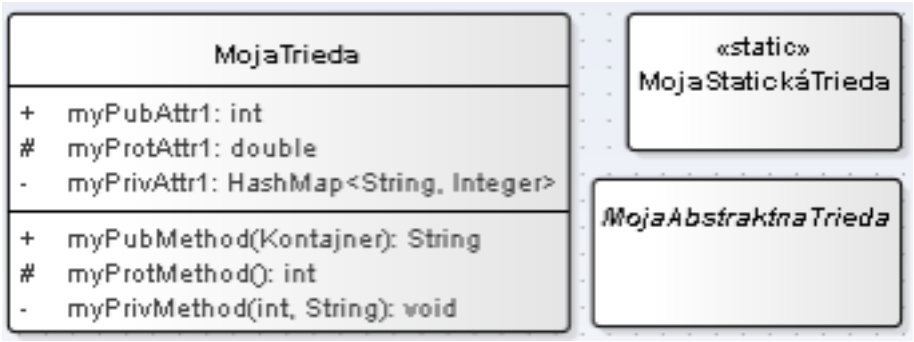
Typy návrhových vzorov

- Toto je zoznam návrhových vzorov podľa Gangu štyroch (Gang of Four – GOF):

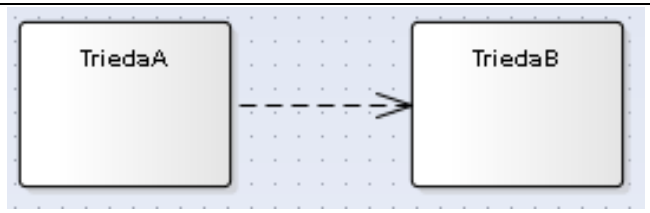
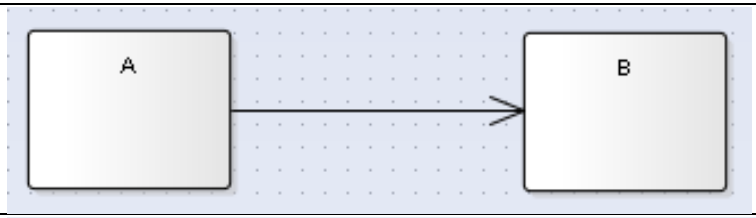
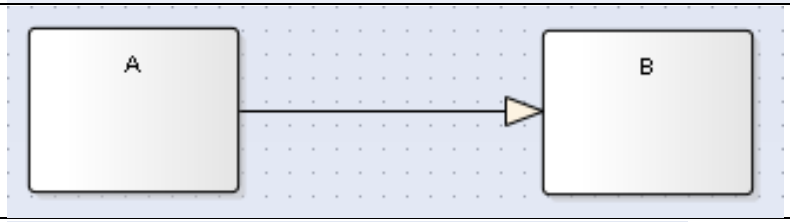
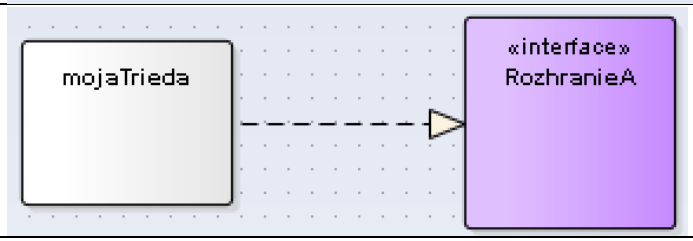
○ vzory na vytváranie (5) ▪ factory ▪ abstract factory ▪ builder ▪ prototype ▪ singleton	○ štrukturálne vzory (7) ▪ Adapter ▪ Bridge ▪ Composite ▪ Decorator ▪ Facade ▪ Flyweight ▪ Proxy	○ Vzory správania (8) ▪ Chain of reponsibility ▪ Command ▪ Iterator ▪ Mediator ▪ Memento ▪ Observer ▪ State ▪ Strategy

Poznámka k UML diagramom tried

- Základné informácie o triede (nomálna, statická, abstraktná, enum, rozhranie, ...) sú v hlavičke
- V tele triedy sa zobrazujú atribúty aj metódy, ich klasifikátory (public, protected, private ako: +#-)
  - Naviac pre Atribúty: typ
  - Naviac pre Metódy: parametre a návratová hod.



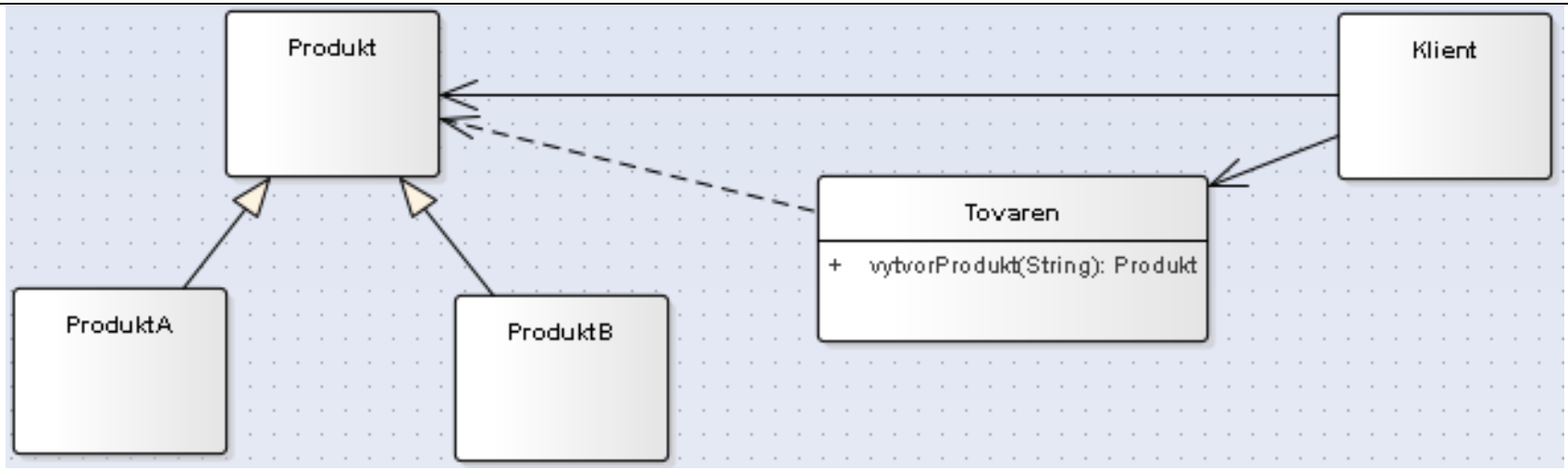
Opakovanie UML

Závislosť		Metóda triedy A používa objekt triedy B v ľubovoľnom mieste implementácia, nie však ako atribút
Jednosmerná asociácia		Označuje, že z objektov triedy A je spojenie na objekt(y) triedy B
Agregácia		Modeluje vzťah celok-súčasť („má“)
Kompozícia		Modeluje vzťah celok-súčasť („skladá sa z“)
Generalizácia		Zovšeobecnenie (**generalizácia)** prvku A na prvok B.
Implementácia rozhrania		mojaTrieda implementuje RozhranieA

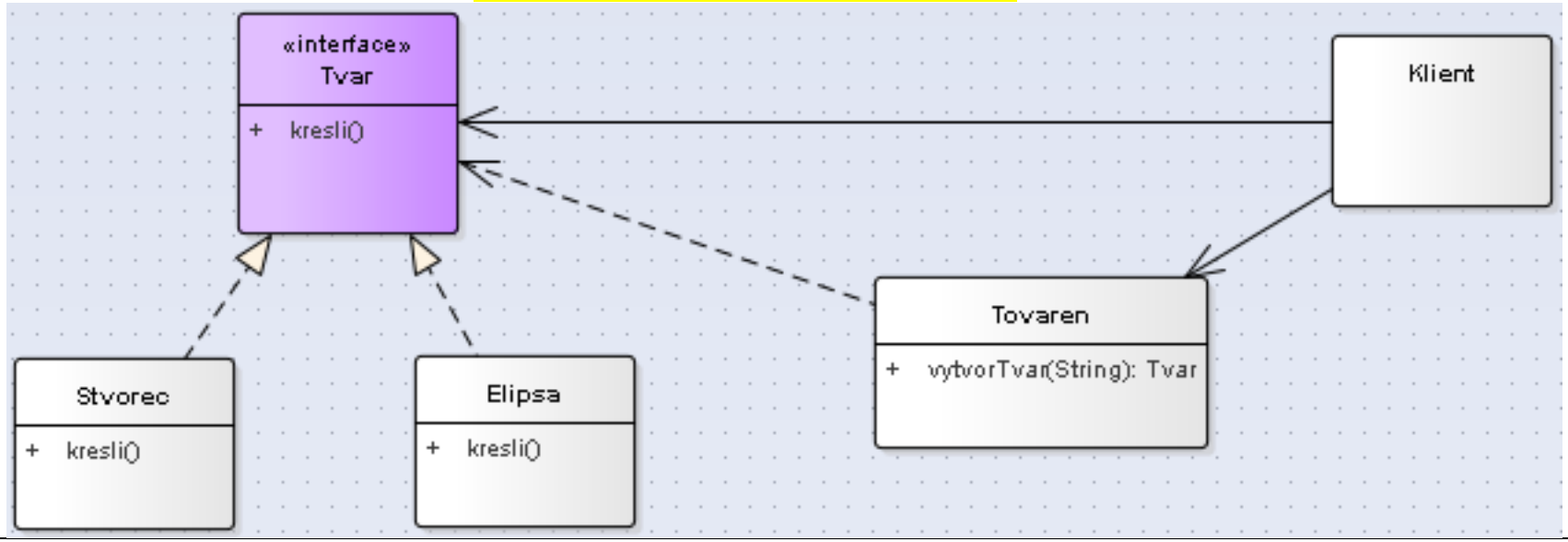
Vzor: „Factory“ (Továreň)

- Jeden z najpoužívanejších vzorov v Jave
- Vytvárame objekty bez toho, aby sme vystavili logiku, ktorá objekty vytvára
- Napr: Továreň vyrába viac druhov produktov. Klient povie aký produkt chce a továreň produkt vyrobí.
  - Definuje rozhranie na vytvorenie objektu (produktu)
  - továreň rozhodne ktorá trieda (výrobná linka), vytvára objekt (produkt) na základe informácií z rozhrania
- Verzia A:

```
public class TestTovarne {
    public static void main(String[] args) {
        Tovaren tv = new Tovaren();
        Produkt prod;
        prod = tv.vytvorProdukt("A");
        prod = tv.vytvorProdukt("B");
…
```



- Verzia B (použije sa rozhranie) [Príklad Prednaska05_Factory1]



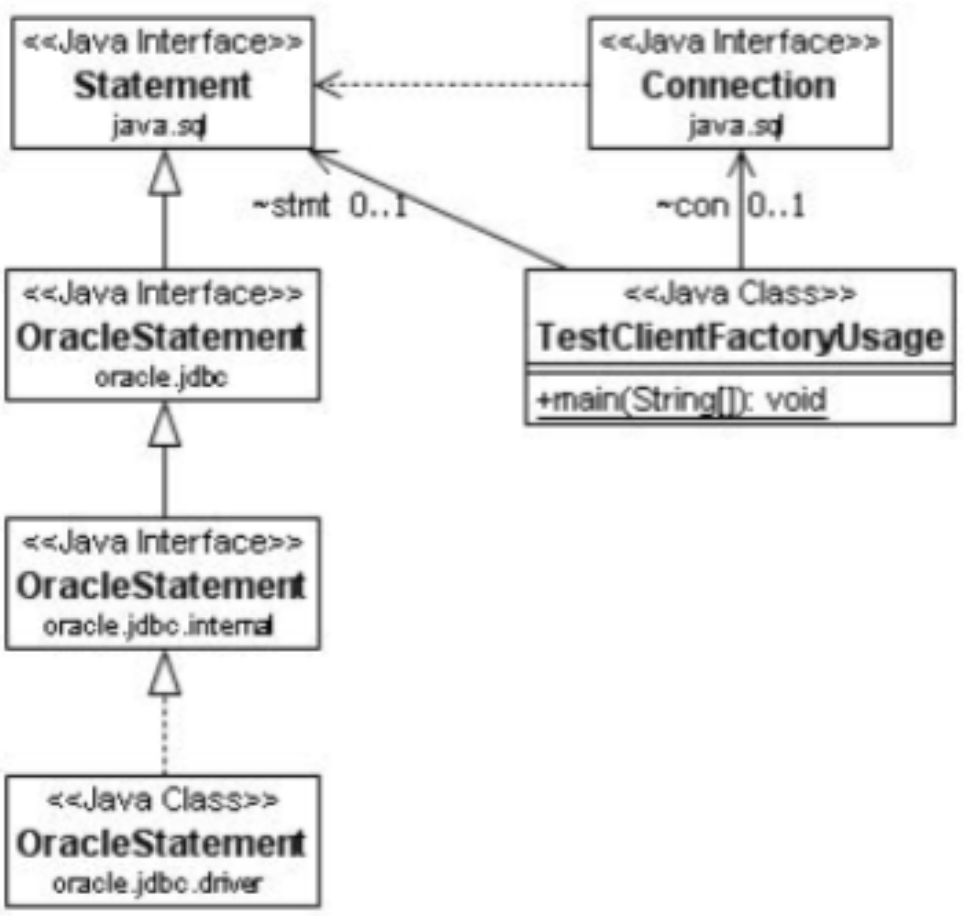

```
public class TestTovarne {
    public static void main(String[] args) {
        Tovaren tv = new Tovaren();
        Tvar mojtvar;
        mojtvar = tv.vytvorTvar("ELIPSA");
        mojtvar.kresli();
        mojtvar = tv.vytvorTvar("STVOREC");
        mojtvar.kresli();
    …
```

```
public class ShapeFactory {
   public Tvar vytvorTvar(String shapeType){
      if(shapeType.equalsIgnoreCase("ELIPSA")){
         return new Elipsa();
      }…
```

Vzor factory - príklad využitia pri pripájaní na server

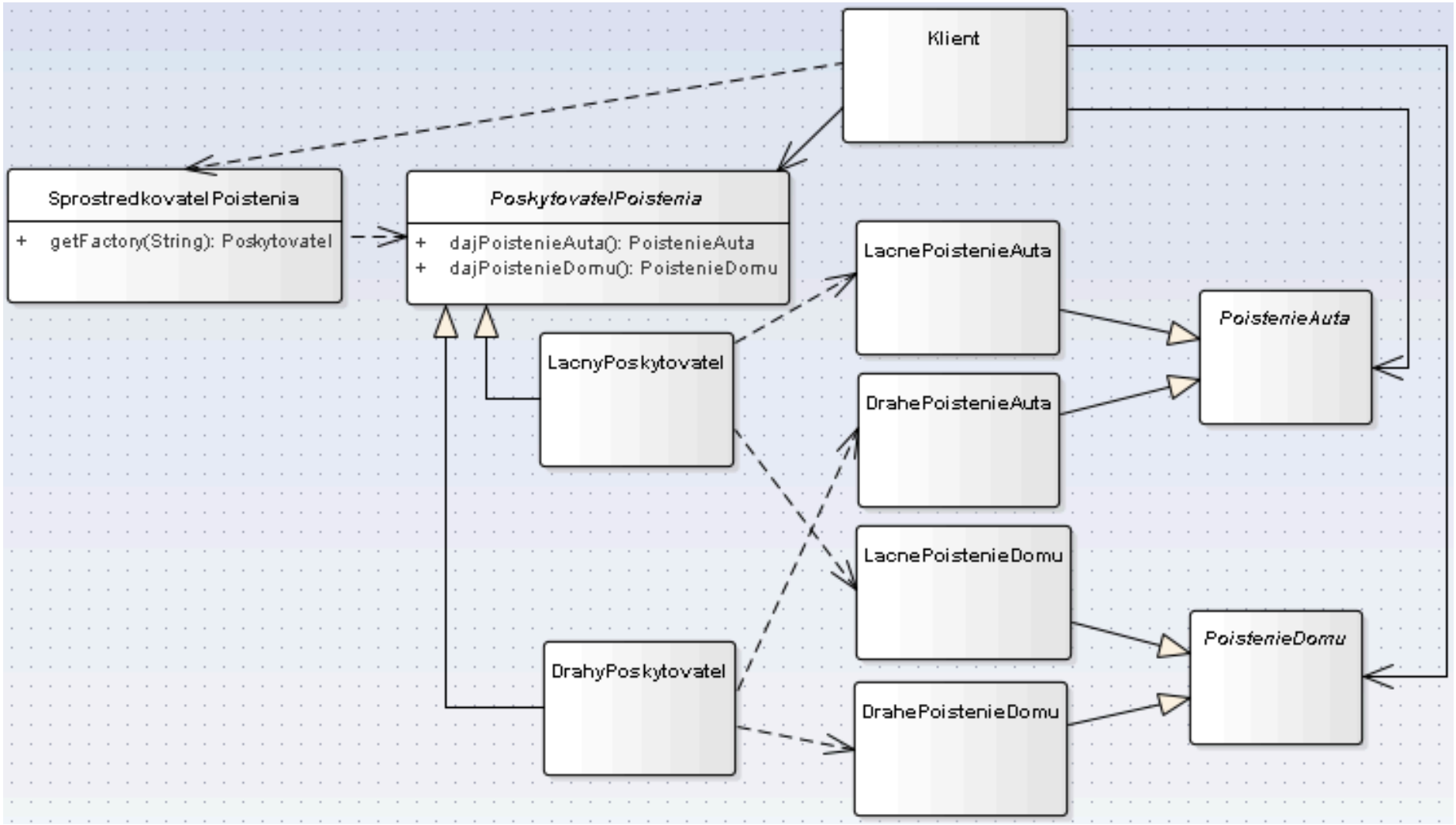
- Connection = továreň
- Statement = produkt

```
import java.sql.*;
public class TestClientFactoryUsage {
 static Connection con;
 static Statement stmt;
 public static void main(String[] args){
  try {
    Class.forName(
      "oracle.jdbc.driver.OracleDriver");
      con = DriverManager.getConnection(
         "myServer", "user",
         "password");
      stmt = con.createStatement();
    } catch(Exception e) {}
  }
}
```

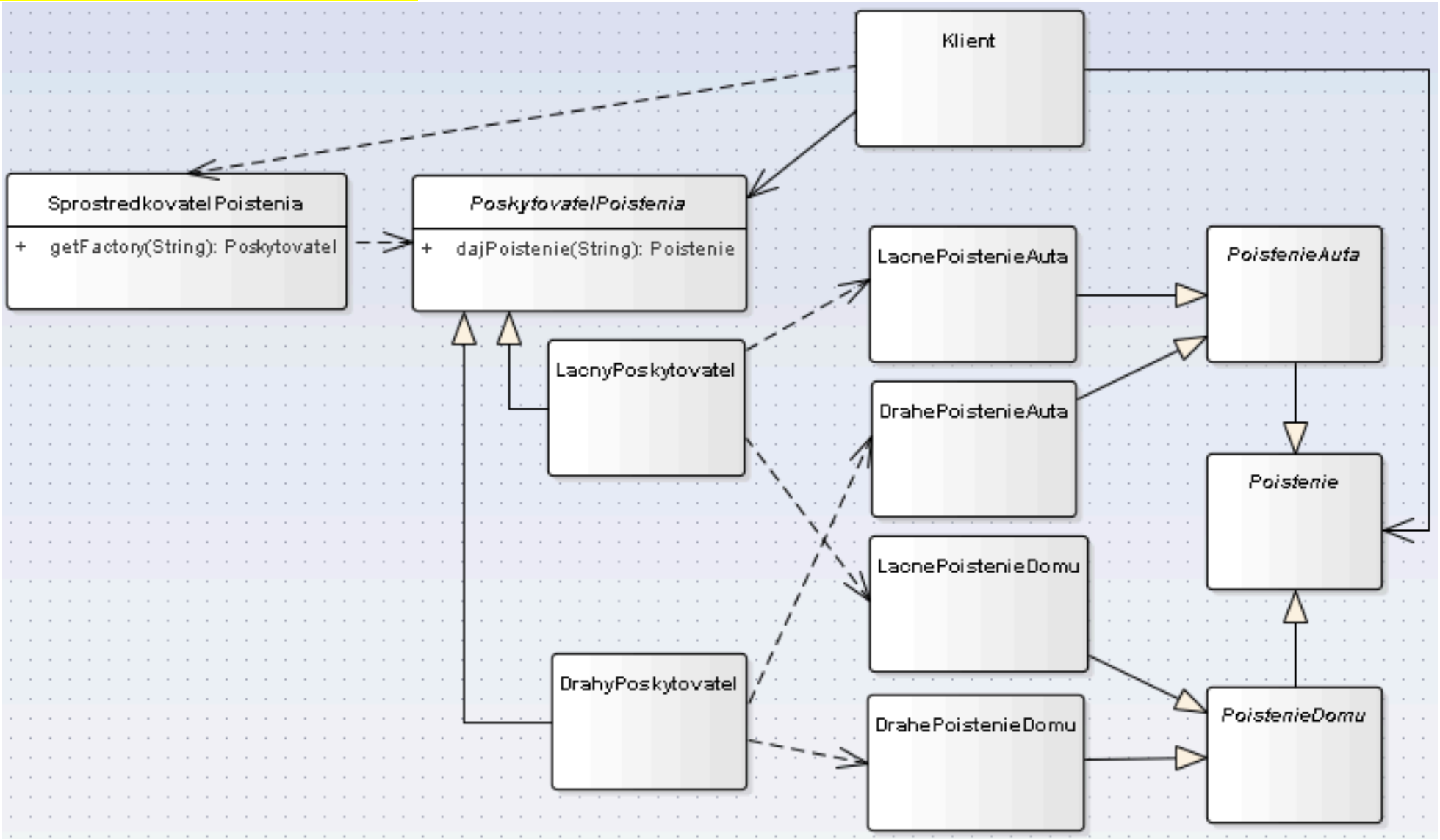


Vzor: „Abstract factory“

- Jedná sa o rozšírenie vzoru továreň o 1 novú úroveň
- Poskytuje abstraktnú supertováreň, ktorá vytvára konkrétne továrne, ktoré ďalej vytvárajú produkty z nejakej oblasti (napr. jedna továreň autá, druhá bicykle ...)
- Napr. Abstraktná továreň=PoskytovatelPoistenia:



[Príklad: Prednaska05_Afactory2]



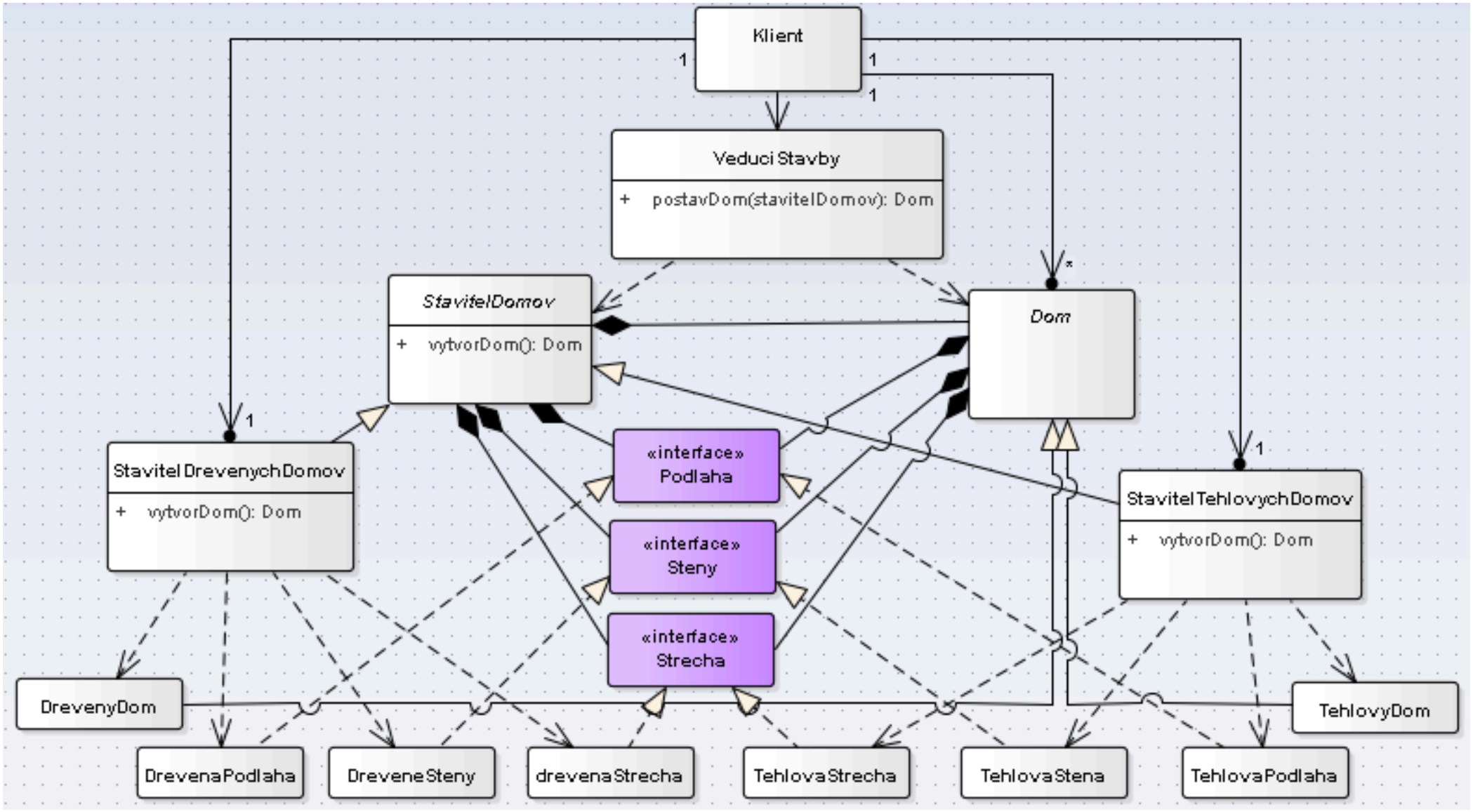
Vzor builder (staviteľ)

- Uľahčuje tvorbu zložitých objektov, ktoré sa skladajú z jednoduchších objektov
  - Napr. Dom sa skladá zo sien, podláh, strechy
- Oddeľuje konštrukciu od reprezentácie, t.j. ten istý konštrukčný proces môže byť použitý na vytvorenie výsledného objektu z iných objektov
  - Napr. celý Dom môže byť vybudovaný z drevených objektov, alebo z tehlových objektov

Príklad napr. [Christiansson, B., et. All: GoF Design Patterns - with examples using Java and UML2, online]

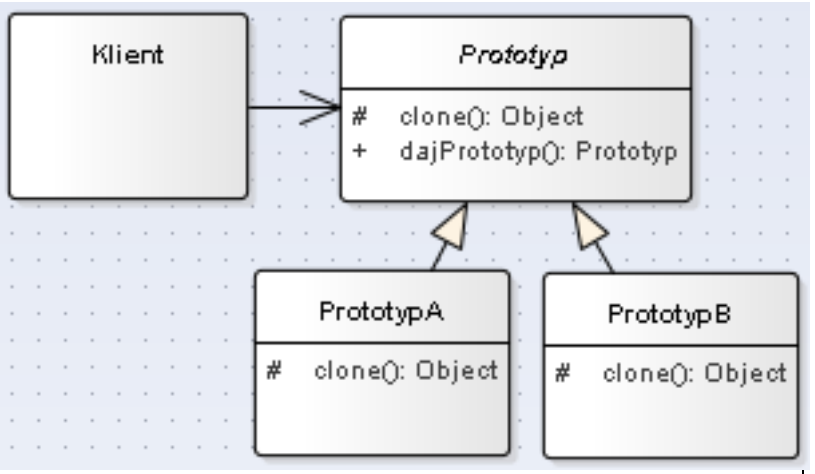
```
public class KlientKtoryChceDom {
   public static void main(String[] args) {
      VeduciStavby mojVeduci = new VeduciStavby();

      StavitelDrevenychDomov mojStavitel1 = new StavitelDrevenychDomov ();
      StavitelTehlovychDomov mojStavitel1 = new StavitelTehlovychDomov ();

      // Postav drevený dom
      Dom mojDrevenyDom = mojVeduci.postavDom(StavitelDrevenychDomov);

      // Postav tehlový dom
      Dom mojTehlovyDom = mojVeduci.postavDom(StavitelTehlovychDomov);
   }
}
…
```

Vzor builder (staviteľ) – UML model príkladu



Vzor Prototyp

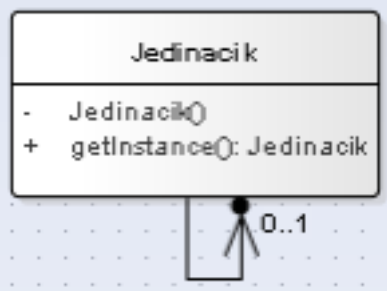
- používa sa na vytvorenie nových inštancií prostredníctvom klonovania existujúcich inštancií
- po vytvorení prototypu sú nové objekty vytvárané kopírovaním tohto prototypu.
- [Príklad: Prednaska05_Prototyp]




```
public class MojPrototyp {
   public static void main(String[] args) throws CloneNotSupportedException {
        // TODO code application logic here
        Produkt kniha0 = new Kniha ("KA", 0, 100);
        Produkt clonProd1 = kniha0.dajKopiu();
        clonProd1.ID=1;
        ((Kniha)clonProd1).pocetStran=200;
        kniha0.tlac();
        clonProd1.tlac();

        Produkt  nahr0 = new Nahravka ("NA", 10, 300);
        Produkt  clonProd2 = nahr0.dajKopiu();
        clonProd2.ID=11;
        ((Nahravka)clonProd2).dlzka=500;
        nahr0.tlac();
        clonProd2.tlac();
    }
}
```

Vzor Singleton (jedináčik)

- poskytuje možnosť vytvárať iba jedinú inštanciu zo zvolenej triedy a poskytuje odkaz na príslušný objekt
  - konštruktor je privátny
  - namiesto konštruktora sa používa public metóda, ktorá ošetruje prístup, napr. „getInstance“.




```
public class FileLogger {
    private static FileLogger logger;
    // zakázané používanie konštruktora zvonku
    private FileLogger() { }
    // vytvorenie/prístup k inštancii je pomocou inej metódy
    public static FileLogger getFileLogger() {
        if (logger == null) logger = new FileLogger();
        return logger;
    }
    // samotná logovacia funkcia spravidla býva označená ako synchronizovaná,
    // t.j. nemôžu ju používať dve vlákna naraz
    public synchronised void log (String logovaciaSprava) {
        ...
    }
}
```

Vzor Adaptér

- používa sa, keď je treba aby trieda začala „podporovať“ dané rozhranie
  - dorobíme vtedy adaptér, ktorý rozšíri pôvodnú triedu („adaptee“)a doimplementuje potrebné rozhranie
  - v príklade Adaptér (`Adapter`) poskytuje nové štandardné rozhranie (`StandardneRozhranie`) na zasielanie správ a pri implementácii využíva metódu na zasielanie z pôvodnej triedy (`sendMessage`).
- Resp. adaptér nemusí „adaptee“ rozširovať (extends), môže mať naňho asociáciu (mAdapteeRef)

```
public interface StandardneRozhranie {
   public Boolean sendMessage(String msgPayload);
}
```

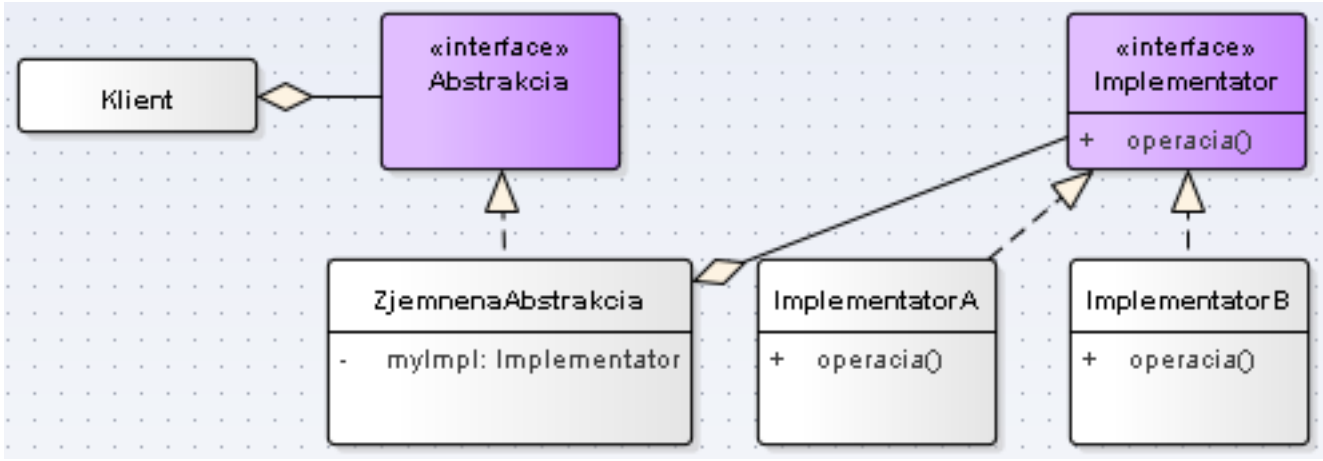
```
public class TriedaNaPosielanieSprav {
   Boolean protected poslispravu (String msgHrd, String msgPayload) {
      ...
   }
   ...
}
```

```
public class Adapter extends ZdrojovaTrieda implements CieloveRozhranie{
   Boolean sendMessage(String msgPayload) {
      ... tu robime potrebne upravy ...
      return posliSpravu(String msgHrd, String msgPayload);
   }
}
```

```
public class Klient{
   public static void main(String[] args) {
   Adapter ma =new Adapter();
   ma.sendMessage(“Telo spravy”)
   ...
}
```

Vzor Most (bridge)

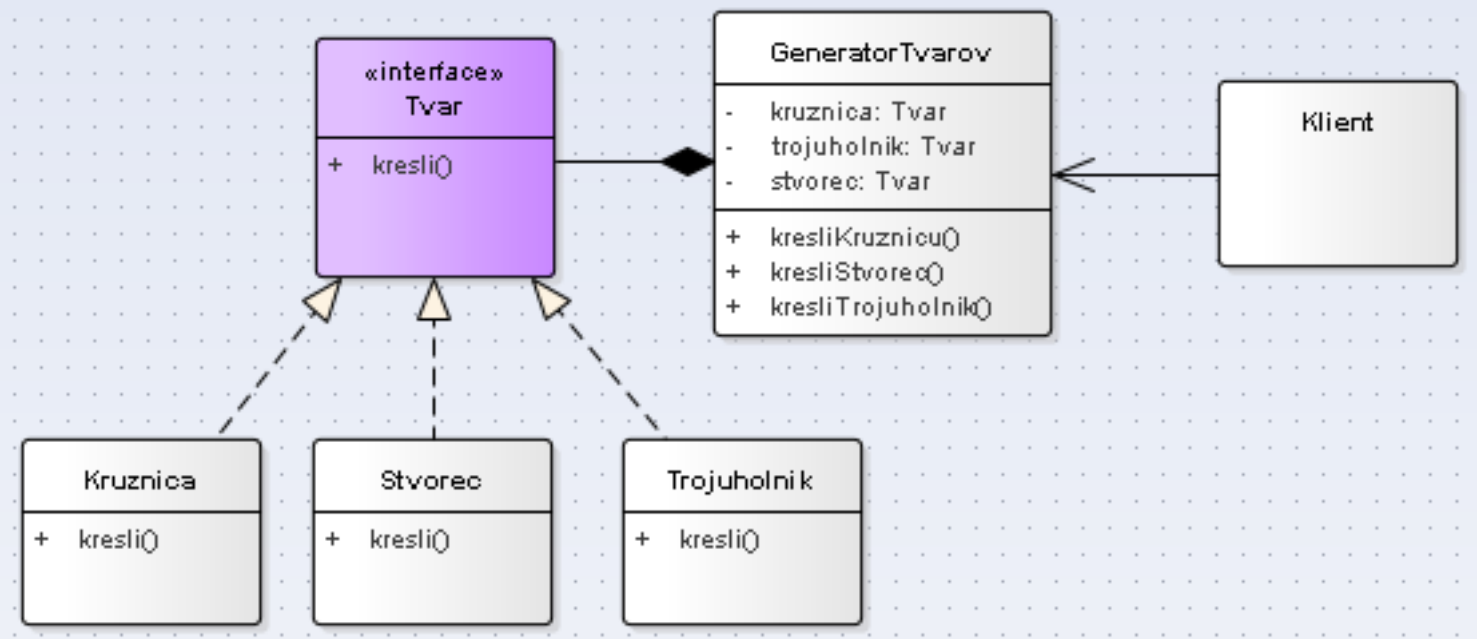
- Oddeľuje Abstrakciu od implementácie (implementátora)
- Implementácia môže byť zmenená počas behu
- Abstrakcia a implementácia môžu byť rozširované a vyskladávané zvlášť
- Príklad: Abstrakcia=Tvar, ZjemnenaAbstrakcia=kružnica, Implementator=DrawingAPI




```
public class Most {
   public static void main(String[] args)    {
   Tvar[] tvary = new Tvar[2];
   tvary [0] = new Kruznica(1, 2, 3, new DrawingAPI1());
   tvary [1] = new Kruznica(5, 7, 11, new DrawingAPI2());
   for (Tvar mojTvar : tvary) {
      mojTvar.kresli ();
   }
}
public interface Tvar {
    public void kresli();
}
public interface DrawingAPI {
    public void kresliKruznicu(double x, double y, double priemer);
}
```

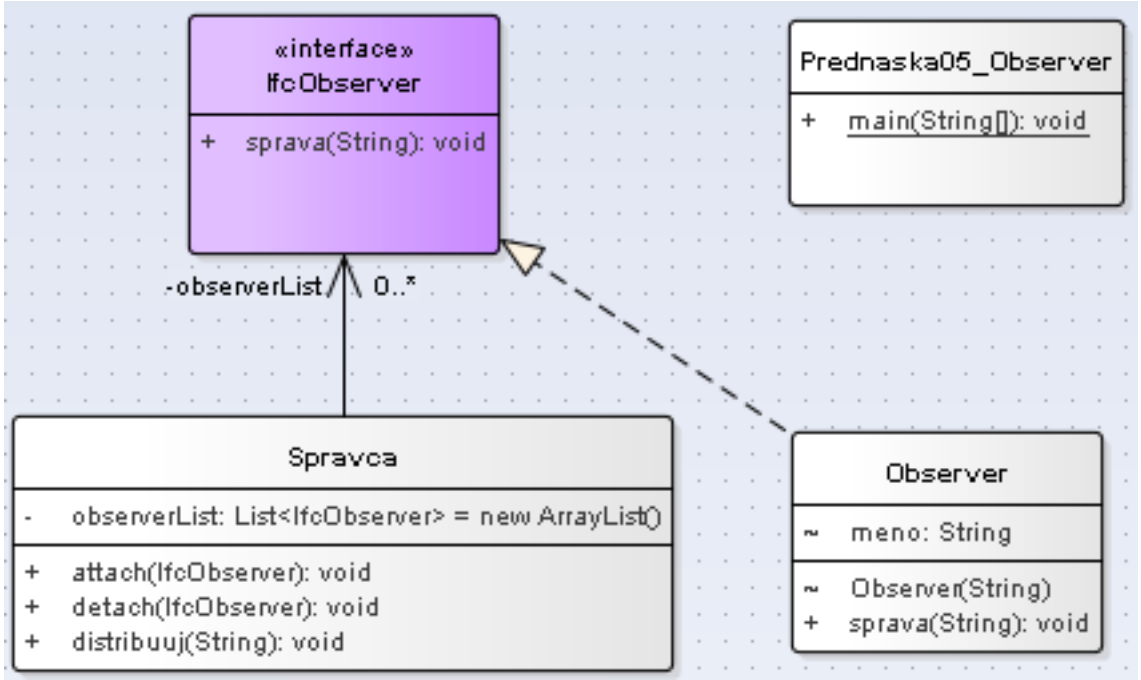
Vzor Fasáda

- Vzor Fasáda ukrýva komplexitu systému a poskytuje jednoduché rozhranie na obsluhu systému
- Je poskytnutá jedna trieda, ktorá ponúka zjednodušené metódy a následne volá ďalšie metódy vnútri systému
- Príklad: Fasáda=GenerátorTvarov



Vzor pozorovateľ

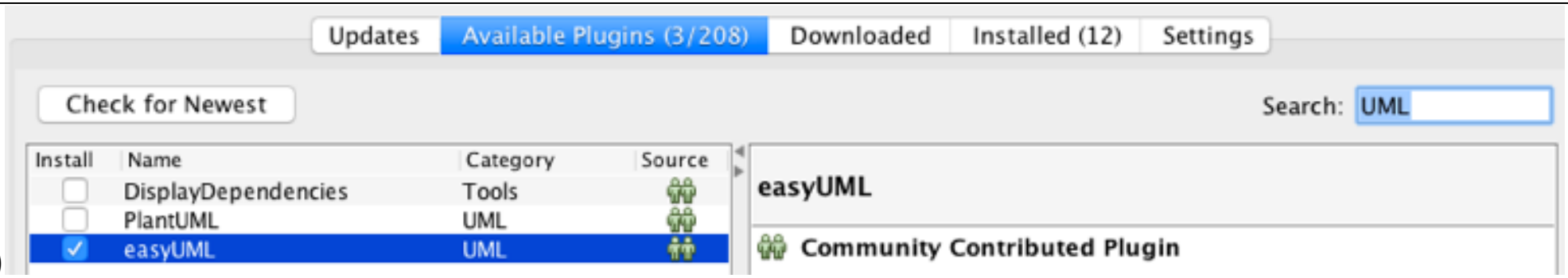
- Sprístupňuje funkcionalitu publish/subscribe, t.j. publikovanie/prihlásenie sa k odberu
- Objekty sa môžu pripájať odpájať od odberu dynamicky
- Príklad: [Prednaska05_Observer]:



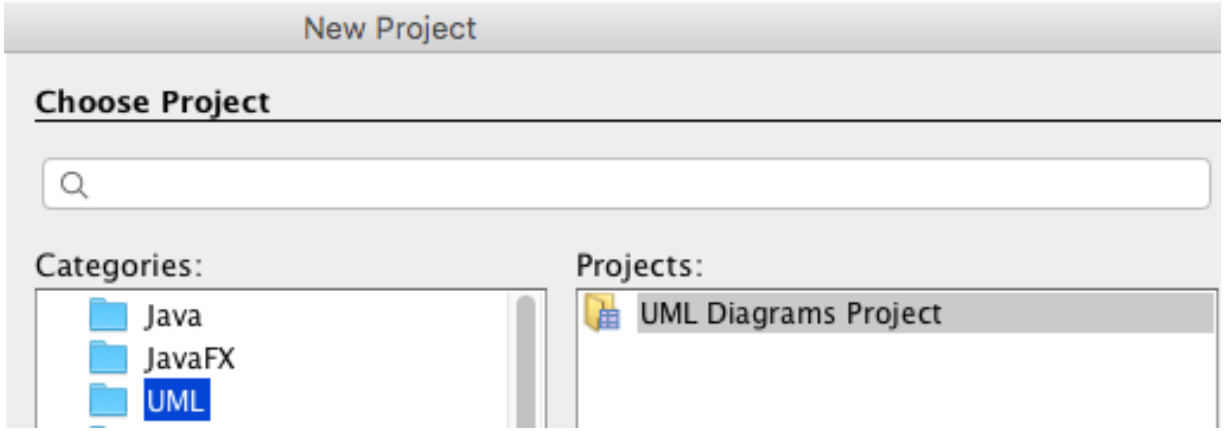

```
…
public static void main(String[] args) {
  Spravca mng = new Spravca ();
  mng.attach(new Observer ("TASR"));
  mng.attach(new Observer ("NBC"));
  mng.attach(new Observer ("CBS"));
  Observer alj = new Observer ("Al Jazeera");
  mng.attach(alj);
  mng.distribuuj("Potopa");
  mng.detach(alj);
  mng.distribuuj("Korupcia");
  mng.distribuuj("Utecenci");
}
…
```

Ako využiť UML v Netbeans na diagramy tried

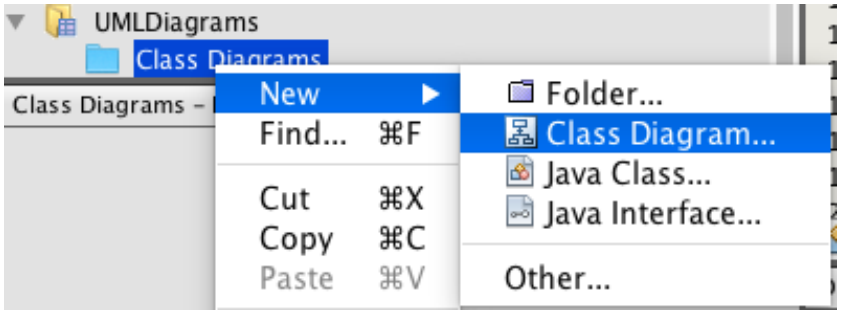
(1)



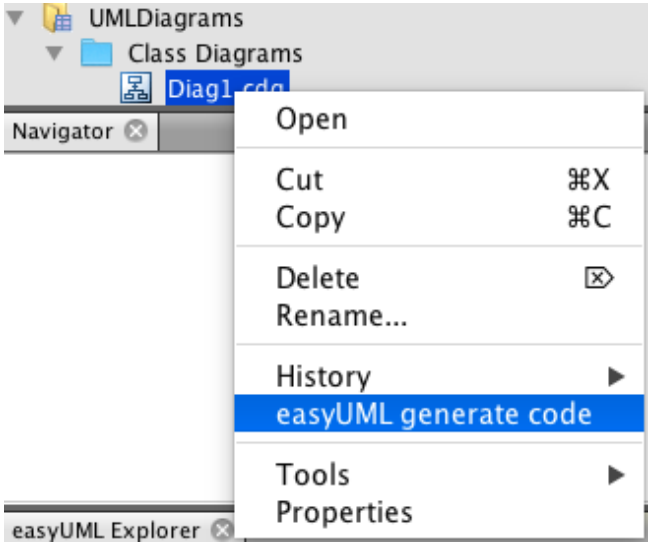
(2)



(3)

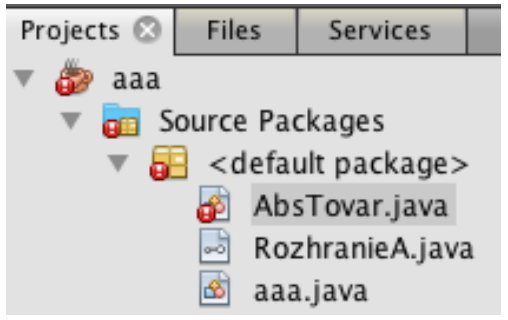


(4) Editovanie CallDiagramu ...

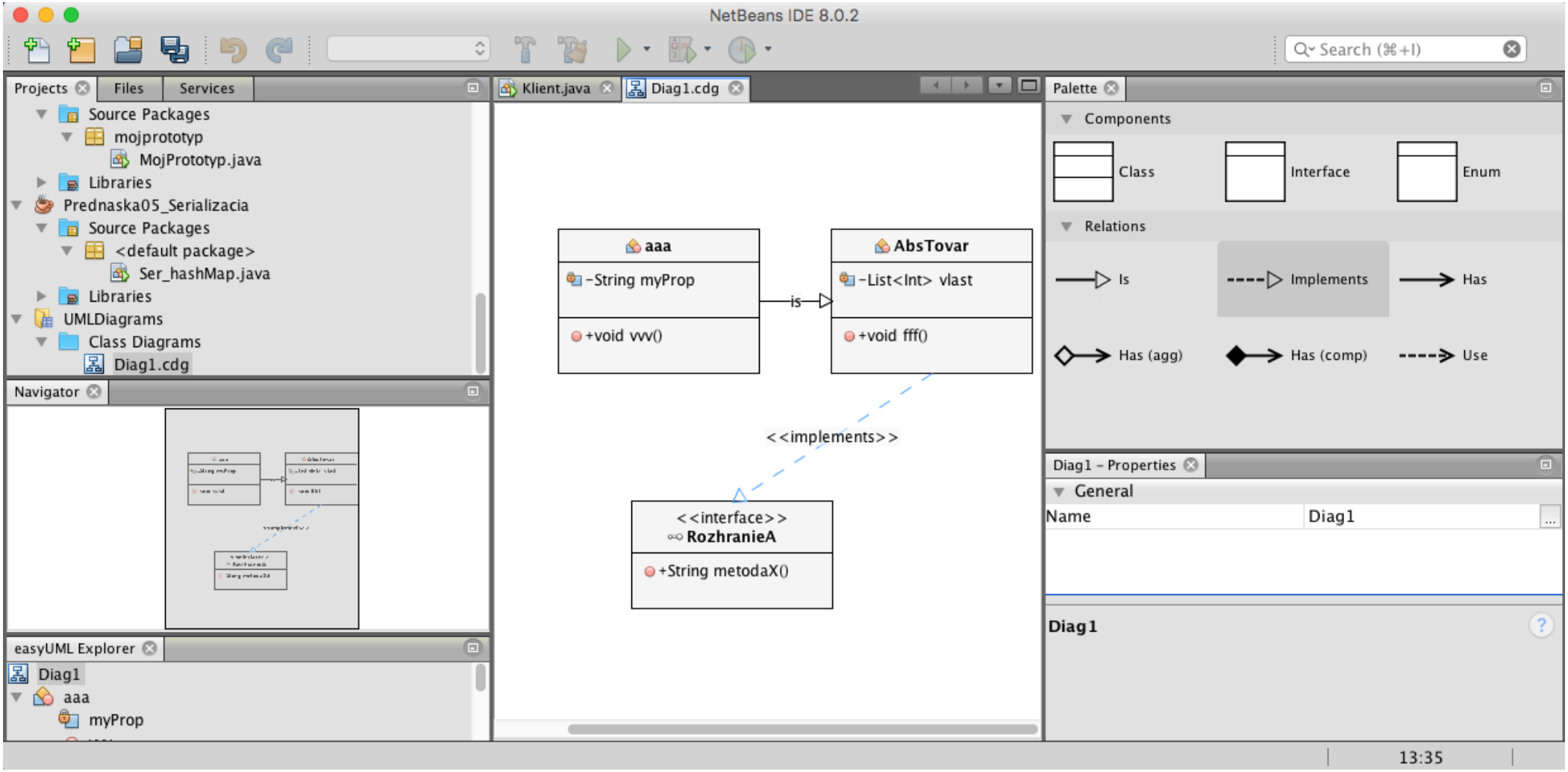


(5)

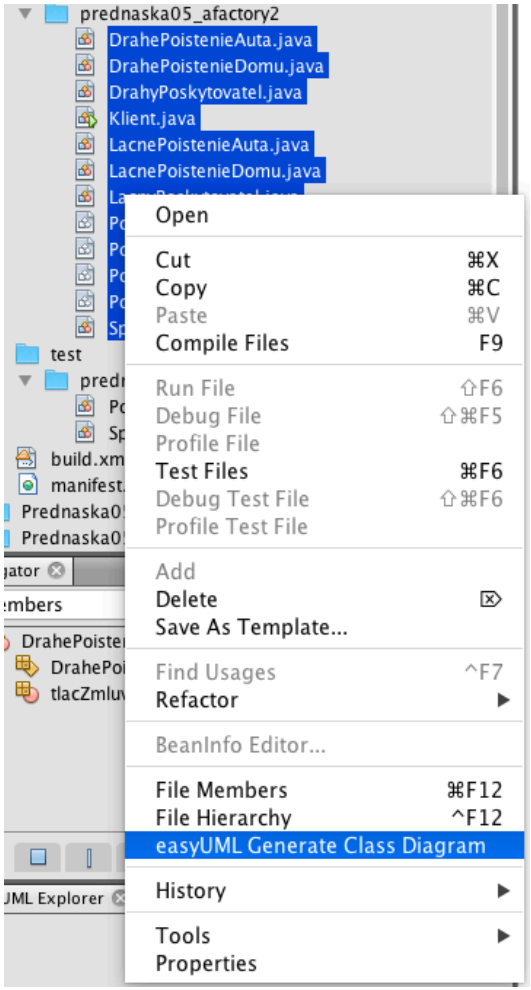
(6)

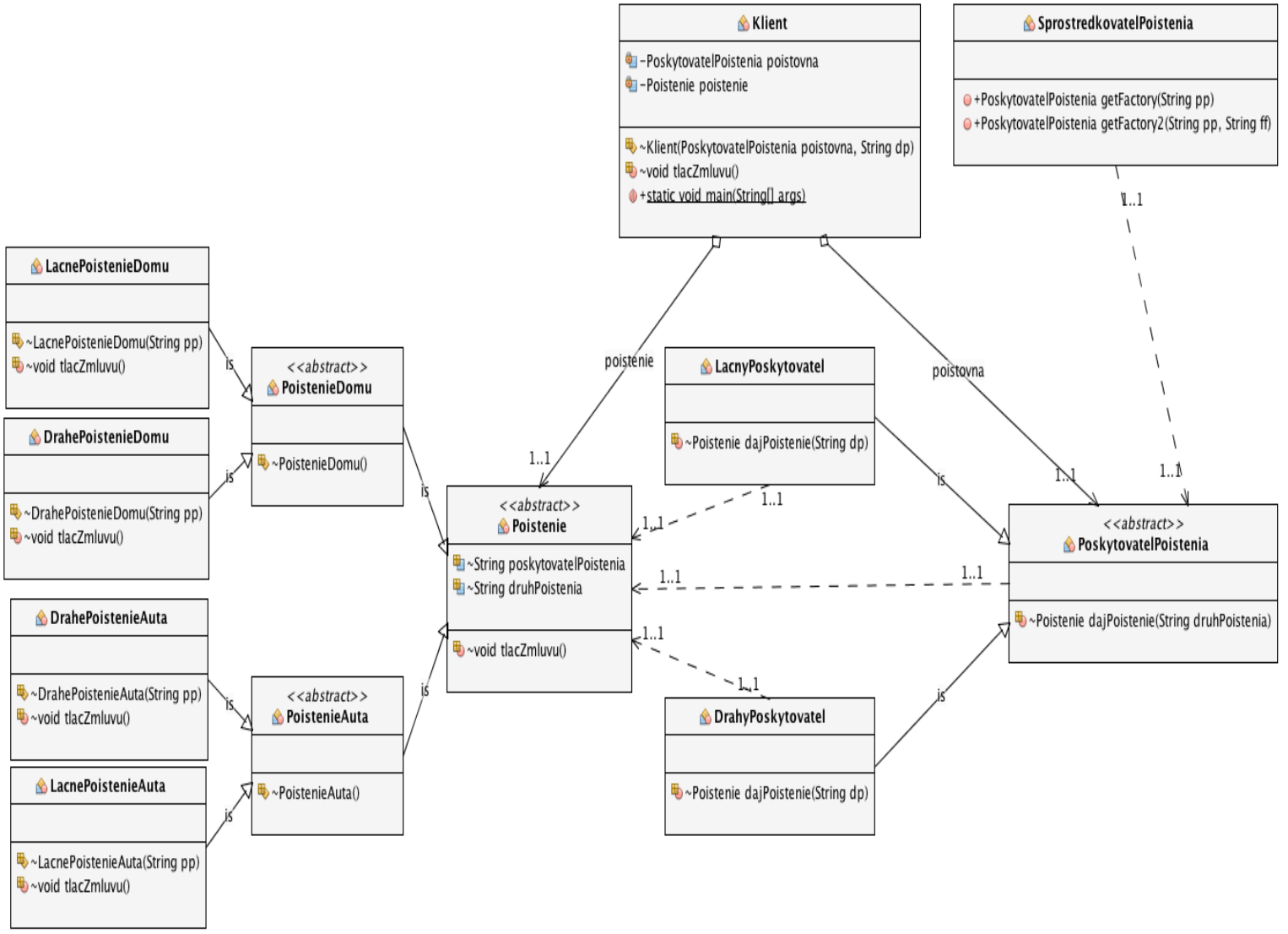


UML v Netbeans2 ...

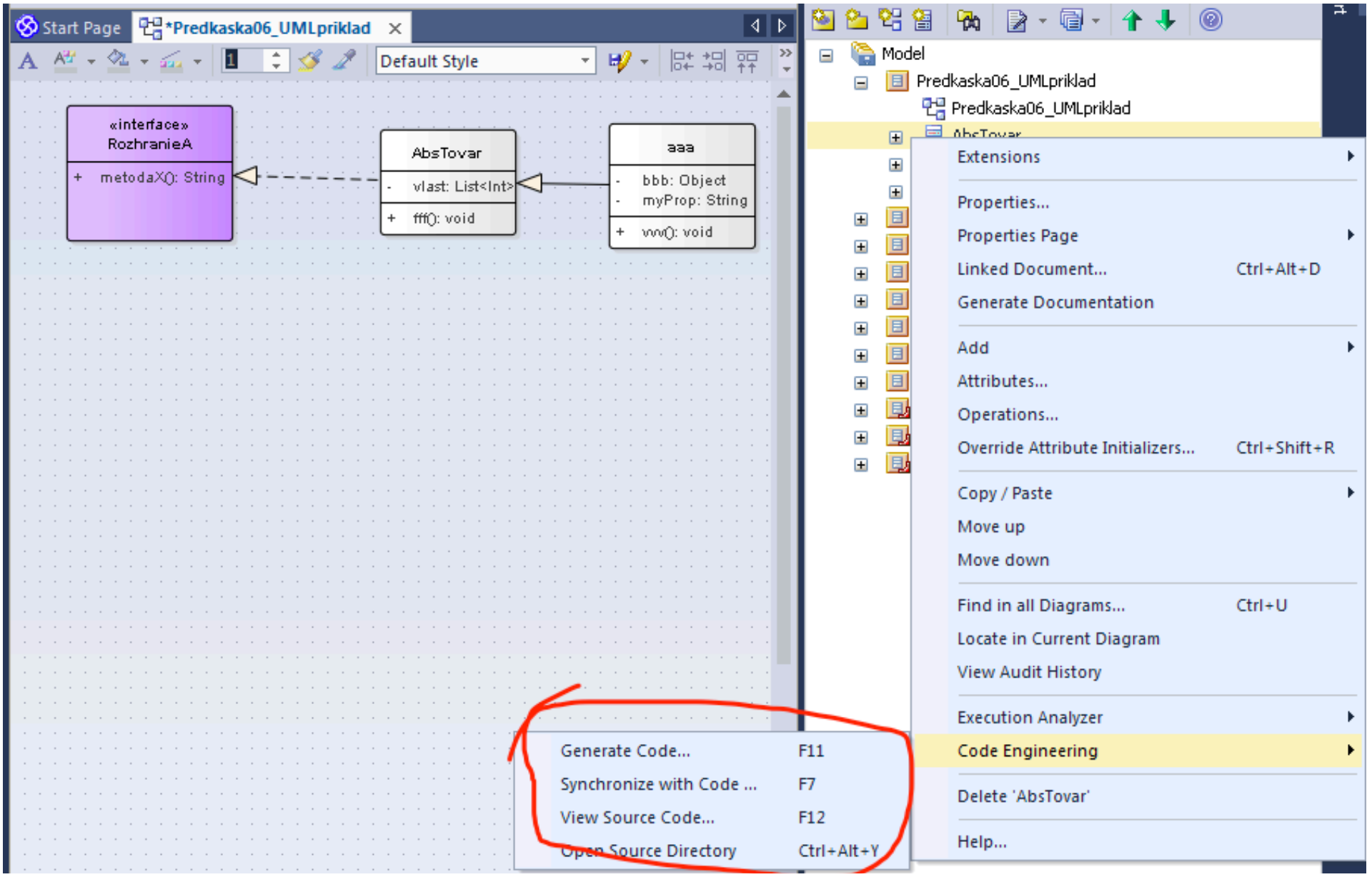


Netbeans – generovanie UML diagramov z kódu (príklad: Abstract factory)





UML v Enterprise Architectovi (EA)



Hashmapa (HM)

- nazýva sa aj hešovacia tabuľka je dátová štruktúra, pomocou ktorej sa implementuje **asociatívne pole**. t.j. štruktúra, ktorá mapuje **kľúče** na **hodnoty**.
- hešovacia tabuľka používa hešovaciu funkciu na vypočítanie INDEXu do poľa, kde sa dá nájsť HODNOTA:

Index=hashFnc(KLUC, VELKOST_POLA)	Index=hashFnc(KLUC)%VELKOST_POLA

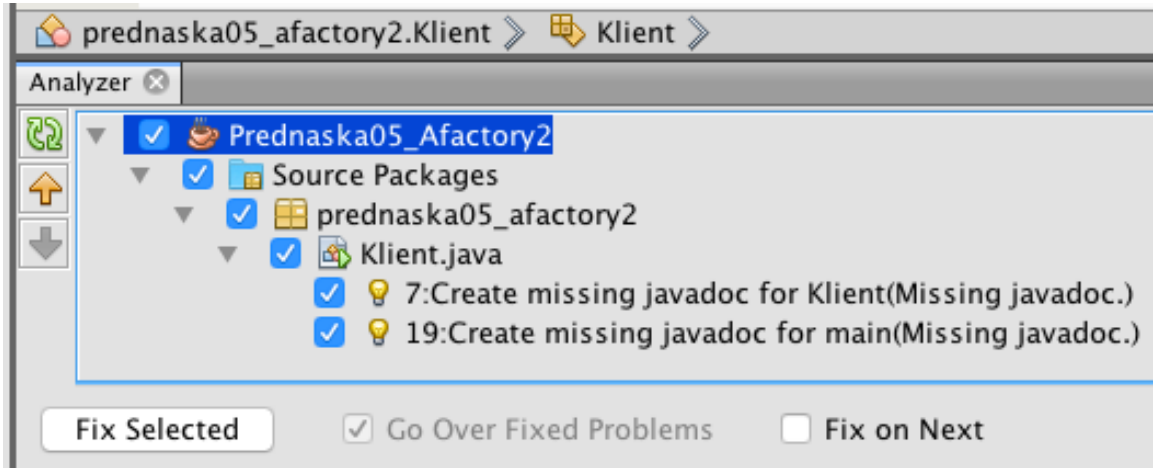
- kolízia = rôzne kľúče vedú k tomu istému indexu
  - kolíziám sa nedá vyhnúť, pravdepodobnosť, že aspoň dva indexy sú v kolízii je minimálne $p(pk, vp) \approx 1 - e^{-pk(pk-1)/2vp}$, kde pk=počet kľúčov, vp=veľkosť poľa, t.j. ak máme napr. 1000 kľúčov a veľkosť poľa 1milión, potom p=0,3932, t.j. s cca 39% pravdepodobnosťou máme kolíziu
  - minimalizácia počtu kolízií → zvolením vhodnej hešovacej funkcie
  - riešenie kolízie → napr. triviálne je, že v každej HODNOTE je zoznam, z ktorého sa volí, zvyčajne má 0-1 záznamov, pri dobrej hešovacej funkcii má zriedkavo viac ako pár hodnôt

Java – serializácia

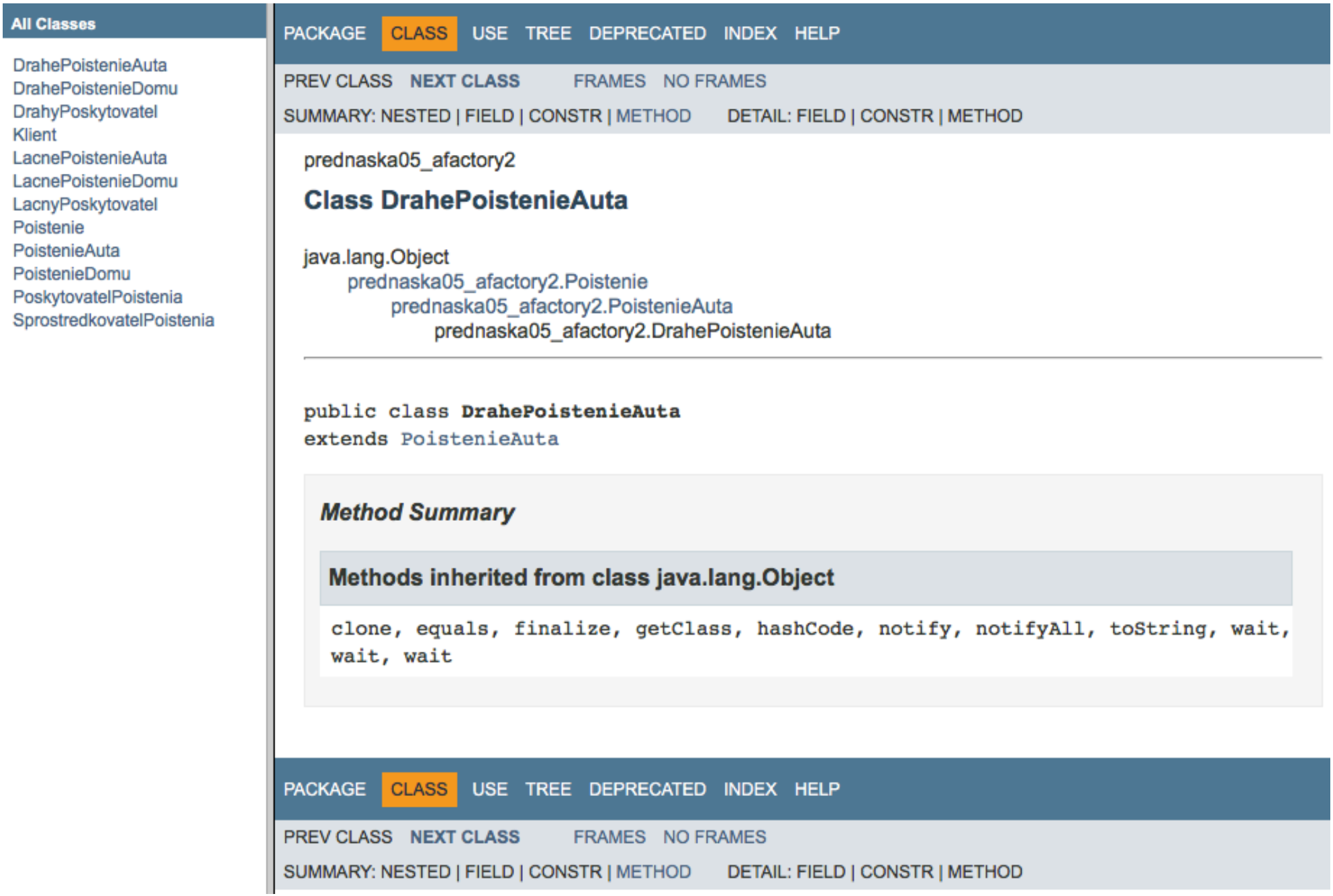
- serializácia je to proces, ktorý z ľubovoľného objektu vytvorí sekvenciu bytov, pričom túto sekvenciu dokážeme naspäť pretvoriť do objektu.
- v jave veľmi mocný nástroj
- daný objekt musí implementovať rozhranie Serializable.
- Príklad: [Prednaska05_Serializacia] - objekty sú LinkedHashMap ...

SW vývoj – dokumentácia kódu

- „Dobrý programátor píše programy, ktorým rozumie nielen počítač ale aj človek“
  - ak budete postupovať podľa myšlienky „poriadok je pre blbcov, inteligent zvláda chaos“ – skôr, či neskôr na to doplatíte
  - v programe sa potrebujete vyznať nielen teraz, ale aj o rok VY SAMI !
- Druhy komentárov
  - Riadkový komentár //
  - Obecný komentár /* tu môže byť viac riadkov textu ... */
  - Dokumentačný komentár
    - určený pre program „javadoc“, ktorý je súčasťou JDK
    - otvára sa postupnosťou znakov /**
    - Používaju sa rôzne doplnkové značky: @param, @return, ...
  - Netbeans pomáha pomocou dalších nástrojov:
    - Tools -> Analyse Javadoc
    - Run -> Generate Javadoc



Javadoc – demo vygenerovaná dokumentácia (bez pridaných značiek)



All Classes

DrahePoistenieAuta
DrahePoistenieDomu
DrahyPoskytovatel
Klient
LacnePoistenieAuta
LacnePoistenieDomu
LacnyPoskytovatel
Poistenie
PoistenieAuta
PoistenieDomu
PoskytovatelPoistenia
SprostredkovatelPoistenia

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

prednaska05_afactory2

Class DrahePoistenieAuta

java.lang.Object
  prednaska05_afactory2.Poistenie
    prednaska05_afactory2.PoistenieAuta
      prednaska05_afactory2.DrahePoistenieAuta

```
public class DrahePoistenieAuta
extends PoistenieAuta
```

Method Summary

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

JavaDoc značky

- @author – povinné pre triedy a rozhrania, za značkou sa zapisuje meno autora, autorov sa môže napísať viac, alebo pre každého sa použije nový riadok s novou značkou. Pole author sa negeneruje do dokumentácie, je viditeľné len v zdrojovom kóde.
- @version – povinné pre triedy a rozhrania, za značkou sa zapisuje číslo verzie, pre formát nie je žiadny predpis
- @param – pre metódy a konštruktory – uvádza sa presný názov parametru z hlavičky a zaň potom popis, pre každý parameter sa pridáva nový riadok.
- @return – pre metódy ktoré vracajú nejakú hodnotu, za značkou sa uvádza popis návratovej hodnoty
- @exception, resp throws – popisuje dôvody, prečo metóda vyhadzuje výnimku
- @see – ukazuje niekam (na iný balík, triedu, jej atribút, metódu)
- @since – popisuje sa od akej verzie je trieda, metóda dostupná
- @deprecated – zdôvodňuje od kedy a prečo je trieda, metóda nepodporovaná

Príklady:

```java
/**
 * Toto je nová pekná metóda, účelovo vytvorena len pre dokumenačné účely
 * @param pp parameter 1 - dummy
 * @param ff parameter 2 - dummy
 * @return vždy vracia null, lebo je to len demo :-)
 */
public PoskytovatelPoistenia getFactory2(String pp, String ff) {
    return null;
}
```

Method Summary

All Methods	Instance Methods	Concrete Methods

Modifier and Type	Method and Description
`PoskytovatelPoistenia`	`getFactory(java.lang.String pp)` Generovanie "továrne" poskytovateľom poistenia
`PoskytovatelPoistenia`	`getFactory2(java.lang.String pp, java.lang.String ff)` Toto je nová pekná metóda, účelovo vytvorena len pre dokumenačné účely

getFactory2

```
public PoskytovatelPoistenia getFactory2(java.lang.String pp,
                                         java.lang.String ff)
```

Toto je nová pekná metóda, účelovo vytvorena len pre dokumenačné účely

Parameters:

`pp` - parameter 1 - dummy

`ff` - parameter 2 - dummy

Returns:

vždy vracia null, lebo je to len demo :-)

Javadoc a @see

Zdrojový kód:

```
* @see LacnePoistenieAuta
* @see LacnePoistenieAuta#druhPoistenia
```

Dokumentácia:

See Also:

`LacnePoistenieAuta, Poistenie.druhPoistenia`

Kam môže see ukazovať:

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package
```