

Modely v JavaFx CRUD aplikácie a ich návrh

UINF/PAZ_{1c}

3.epizóda

PAZ

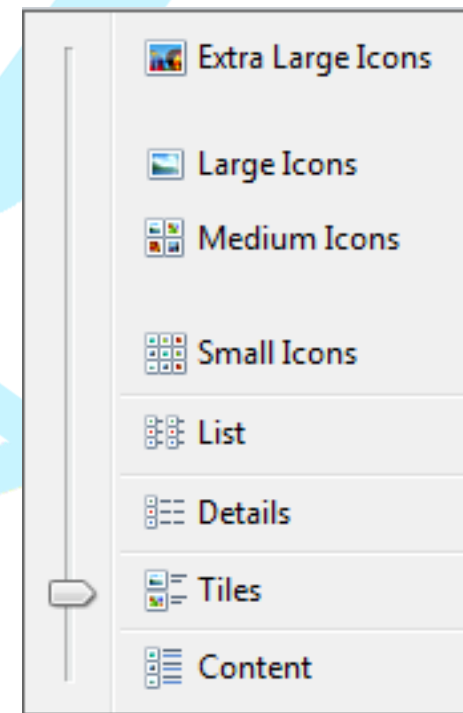
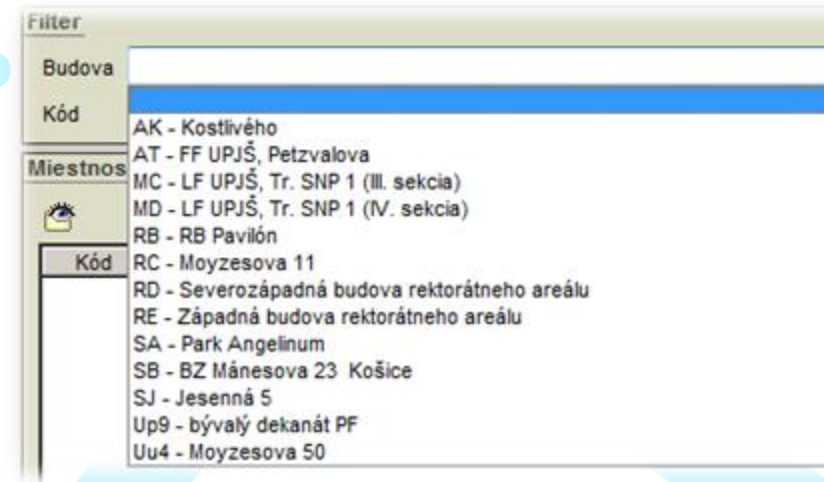


JavaFX – čo už vieme

- Javisko = okno v OS
- Scéna = vytvorená v java alebo v FXML súbore
 - FXML kreslíme cez Scene Builder
- Komponenty na scéne vedia odchytiť udalosti používateľa a informovať poslucháčov
 - Poslucháča pridáme tak, že na komponente zaregistrujeme objekt triedy čo implementuje interfejs EventHandler
- Premenné komponentov máme a spracovanie udalostí robíme **v kontroléri** – objekte, ktorý prepojíme so scénou

Zobrazenie dát

- jeden druh dát možno **zobraziť** viacerými spôsobmi
- akú **metaforu** zvoliť?
- čo je v danej situácii **dôležité** zobraziť?
- akým **estetickým** spôsobom?



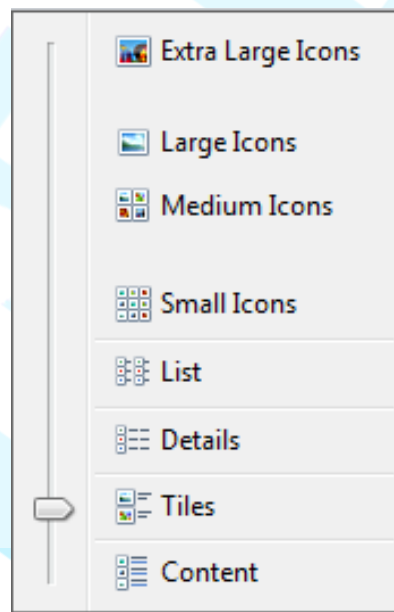
Cesta k návrhovému vzoru

Ako umožniť zobrazenie jedných dát viacerými spôsobmi?

- **oddelíme** dáta od spôsobu ich zobrazenia
- +
- určíme objekt zodpovedný za komunikáciu medzi **zobrazovačom** a **nosičom dát**

Model = ČO zobrazujeme, aké dáta

- **model** = objekt, ktorý poskytuje komponentu dáta na zobrazenie
- viacero komponentov môže zobrazovať dáta z modelu odlišným spôsobom (v zozname, tabuľke, ...)



view = AKO zobrazujeme

- **view** = objekt, ktorý namaľuje dáta používateľovi
- obvykle komponent, widget, ovládací prvok

Name	Type	Total Size	Free Space
Hard Disk Drives (2)			
Local Disk (C:)	Local Disk	237 GB	50,2 GB
Local Disk (D:)	Local Disk	237 GB	50,2 GB
Devices with Removable Storage (2)			
DVD RW Drive (E:)	CD Drive		
SD Disk (G:)	Removable Disk	30,0 GB	390 MB
Network Location (2)			
Data (\\192.168.1.10...	Network Drive	1,33 TB	398 GB

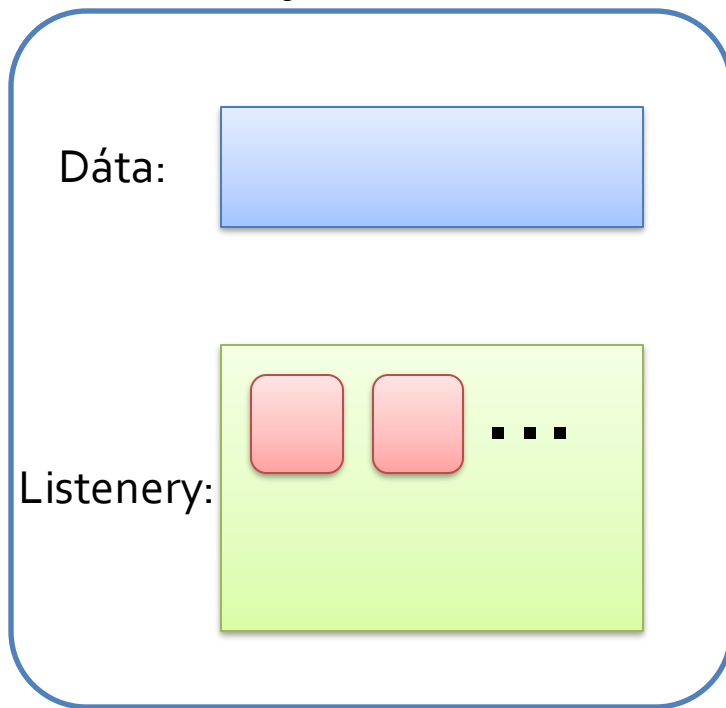
Hráme sa s dátami v komponentoch

```
public class EntranceController {  
  
    private final ObservableList<User> users =  
        FXCollections.observableArrayList();  
  
    @FXML  
    private ListView<User> userListView;  
  
    @FXML  
    void initialize() {  
        userListView.setItems(users);  
    }  
}
```

To čo je?
A prečo?

Observable/property objekty

Observable objekt



- Observable objektom vieme pridať listenery implementujúce rozhranie **EventHandler**
- Ak sa zmenia dáta observable objektu, listenery sú „informované“
 - Observable objekt vytvorí zápis o zmene do objektu typu `ActionEvent` a zavolá všetkým listenerom metódu **`handle(ActionEvent event)`**

ObservableList

- modelom komponentu **ListView** je trieda implementujúca **ObservableList**
- Keď sa v ňom niečo zmení
(pridanie/vymazanie/zmena prvku) **vyvolá udalosť** „zmenili sa mi dáta“

ObservableList

- Listener-om pre tento model je aj samotný ListView
- ListView sa po každej zmene modelu prekreslí s novými dátami.
- **Takže ak chcem zmeniť dáta komponentu, už nevolám jeho metódy, ale modifikujem jeho model**

Každý komponent potrebuje model

komponent : ListView<User>

model: ObservableList<User>

users: List<User>

jozko: User

jozka: User

fero: User

vykresľuje

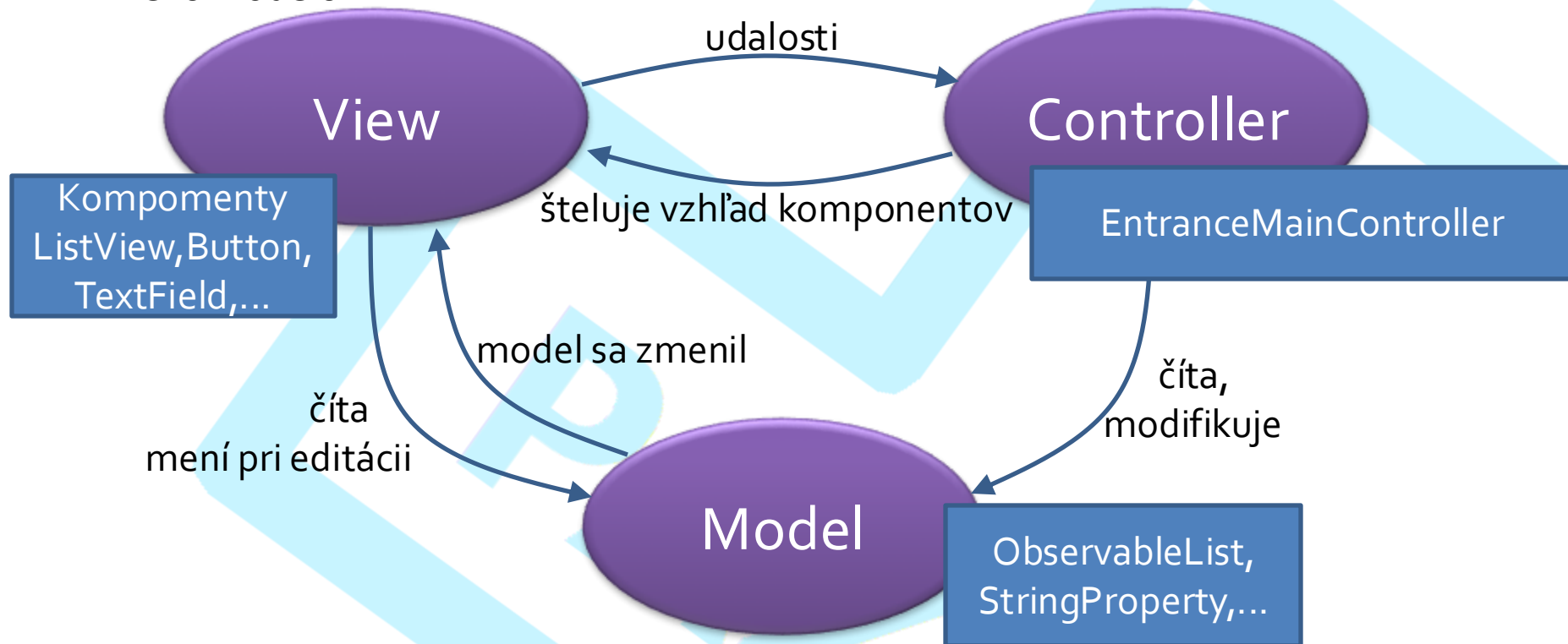
The screenshot shows a web application window titled "Prístupový systém". It contains two input fields: "ID čipu:" and "Meno:", followed by a "Pridať" button. Below the inputs is a table displaying a list of users.

User [chipId=123456, meno=Jano, active=true]
User [chipId=654321, meno=Anka, active=true]

Model – View – Controller (MVC)

- Odchytáva používateľské akcie
- Prekresľuje GUI, keď sa dozvie zmenu modelu

- Spracováva vstup od používateľa
- Mení alebo vymieňa model



- Zachováva zobrazovaný obsah
- Poskytuje dáta pre View, keď ich potrebuje

TextField

- Základný vstupno-výstupný komponent
- Jeho model môžeme meniť z kódu aj priamou editáciou

TextField

- V našom prípade máme 2 textové políčka
 - Kód čipu, Meno
 - Každý má svoj vlastný (skrytý) model reťazcového typu konkrétne `StringProperty` – nevieme ho nahradiť svojím
 - Vieme ho čítať cez `textField.getText()` a nastavovať cez `textField.setText()`
 - Ak zmeníme hodnotu cez `setText()`, listenery sú informované, a teda aj grafická reprezentácia `TextField-u` - textové okienko (view)
 - V obsluhu tlačidla „Pridať“ môžeme obe hodnoty vytiahnuť, vytvoriť a vyskladať objekt typu používateľ a uložiť ho

Model modelov

- Skutočným spoločným modelom oboch textových políček z nášho pohľadu je používateľ
 - Kód čipu a meno sú iba jeho atribúty
- Vyrobneme si UserFxModel
 - Jeho atribúty budú modely pre komponenty
 - `javafx.beans.property.StringProperty`
 - Ak mu niekto zmení hodnotu cez setter, komponent sa prekreslí
 - Takýchto XxxProperty je celá kopa – podľa vnútorného typu – `boolean`, `double`, `float`, `integer`, `long`, `list`, `map`, `set`, `object`
 - Vieme ich prepojiť s modelom `TextFieldu`

Gettery a settery pre property premenné

```
public class UserModel {  
    private final StringProperty name = new SimpleStringProperty();  
    ...  
    public String getName() {  
        return name.get();  
    }  
  
    public void setName(String name) {  
        this.name.set(name);  
    }  
  
    public StringProperty nameProperty() {  
        return name;  
    }  
}
```

Ten, kto počúva na udalosti zmeny hodnoty, bude informovaný

Pýta si kód, ktorý chce pridať nového listenera

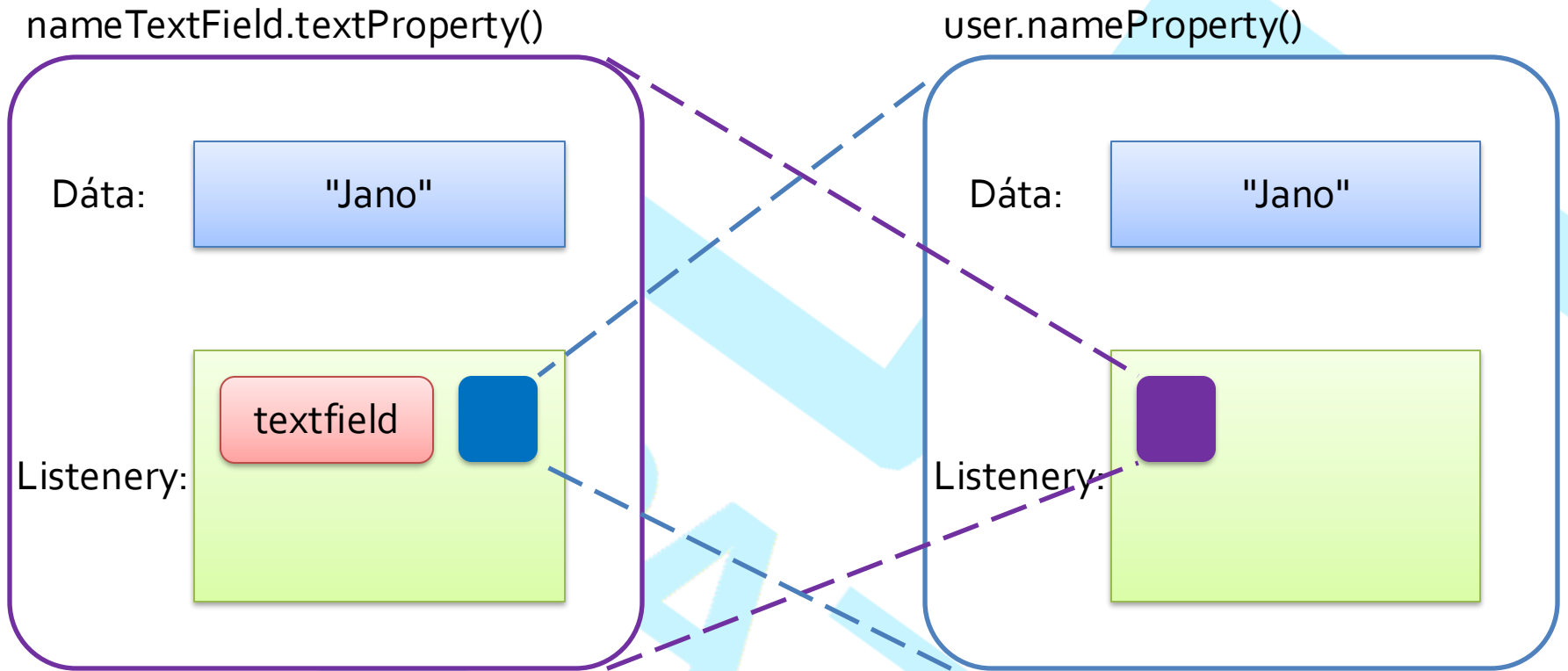
Prepojenie na model komponentu

```
public class UserModel {  
    private final StringProperty name = new SimpleStringProperty();  
    ...  
    public String getName() {  
        return name.get();  
    }  
  
    public void setName(String name) {  
        this.name.set(name);  
    }  
  
    public StringProperty nameProperty()  
        return name;  
    }  
}
```

Obojsmerné
informovanie o
udalostiach zmien

```
nameTextField.textProperty().bindBidirectional(user.nameProperty());
```

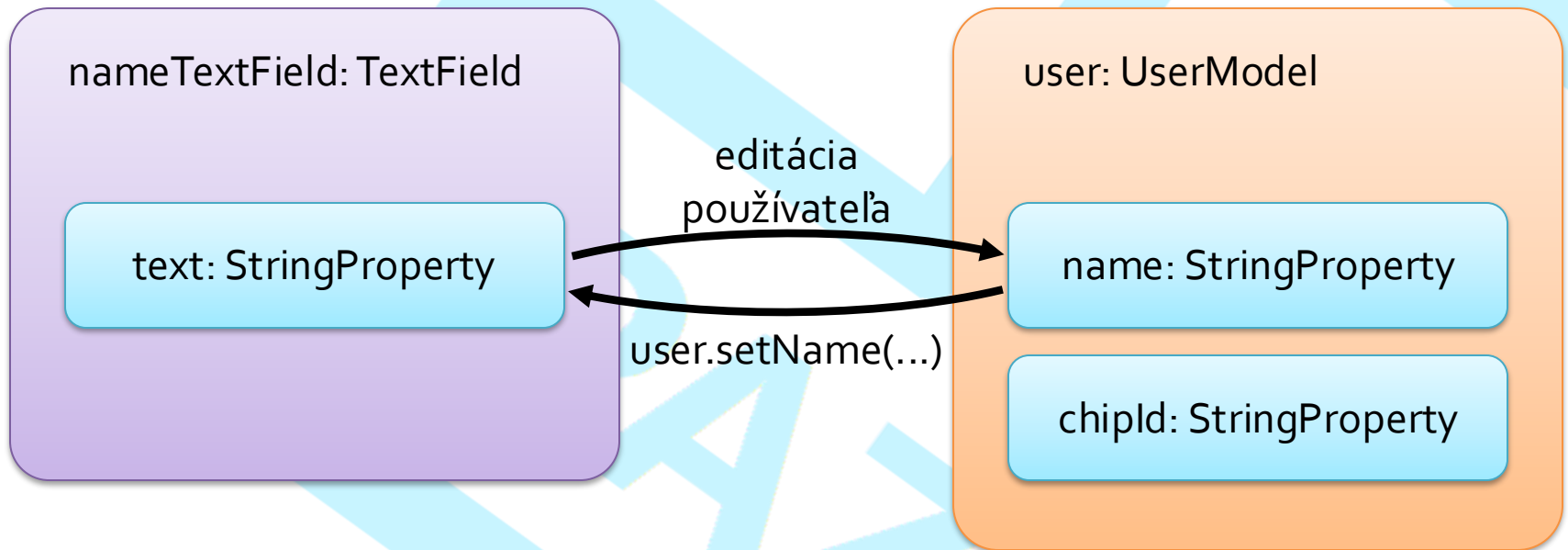
Obojsmerné prepojenie



```
nameTextField.textProperty().bindBidirectional(user.nameProperty());
```

Prepojenie na model komponentu

```
nameTextField.textProperty().bindBidirectional(user.nameProperty());
```



Kontrolér s modelom používateľa

```
public class EntranceController {
    private final ObservableList<User> users = ...;
    private UserModel userModel = new UserModel();
    ...
    @FXML
    void addUserButtonAction(ActionEvent event) {
        userListView.getItems().add(userModel.getUser());
    }

    @FXML
    void initialize() {
        nameTextField.textProperty()
            .bindBidirectional(userModel.nameProperty());
        doorTextField.textProperty()
            .bindBidirectional(userModel.doorProperty());
        ...
    }
}
```

```
public class UserModel {
    ...
    public User getUser() {
        var user = new User();
        user.setName(name);
        ...
        return user;
    }
}
```

Kontrolér s modelom používateľa

```
public class EntranceController {  
    @FXML  
    private ListView<User> userListView;  
    private UserModel userModel = new UserModel();  
    ...  
    @FXML  
    void addUserButtonAction(ActionEvent event) {  
        userListView.getItems().add(userModel.getUser());  
    }  
}
```

Toto nie je **univerzálne** správne riešenie.
Sú aj iné prístupy. Závisí od okolností.

Entity (ako User) by mali byť
serializovateľné, čo FX Properties nie sú.

Najväčšiemu množstvu chýb sa vyhneme,
ak entity (ako User) nastavujeme v modeli.

```
public class UserModel {  
    ...  
    public User getUser() {  
        var user = new User();  
        user.setName(name.get());  
        ...  
        return user;  
    }  
}
```

```
property());  
dProperty());
```



**ÚLOŽISKO /
PERZISTENTNÁ VRSTVA**

Use-case správcu prístupového systému

- **pridať** používateľa
 - vyplní ID čipu, meno, ...
- **vyhľadať** podľa rozličných kritérií
 - podľa mena, IDčka
 - podľa posledného prihlásenia,...
- **prehliadať** si evidovaných používateľov
 - zoradené podľa mena, času posledného prihlásenia
- **upraviť** používateľa
 - Zmeniť aktivnosť, zmeniť čip
- **vymazať** používateľa

Čo sú tvoje
dáta?

Ako sa nimi
žongluje v
systeme?

Kde ich máš
uložené?

Ako ich
zobrazuješ
používateľovi?



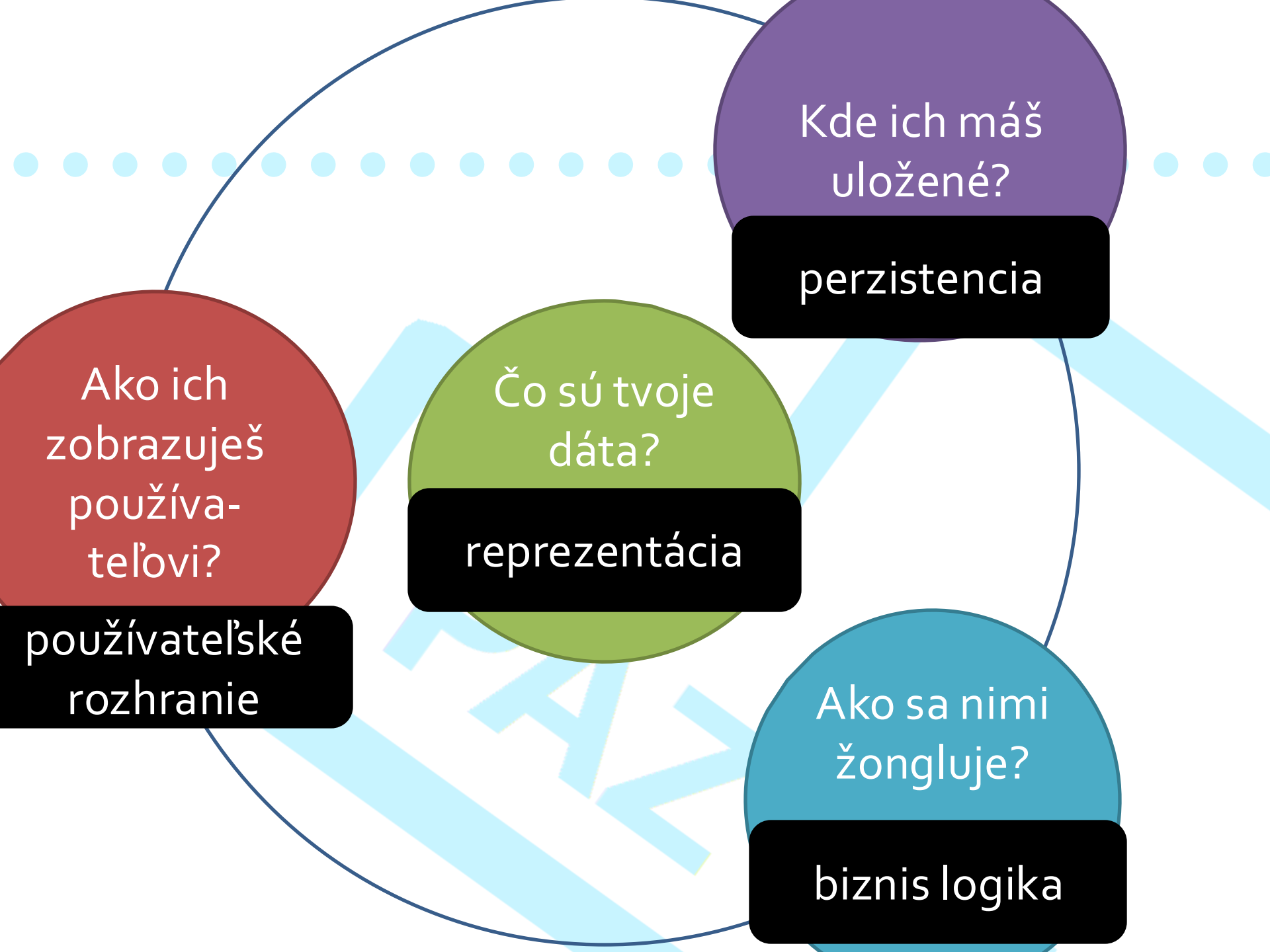
Čo sú tvoje
dáta?

Ako sa nimi
žongluje v
systéme?

Kde ich máš
uložené?

Ako ich
zobrazuješ
používate-
ľovi?

- pracujeme s používateľmi a ich čipmi
 - a s časmi použitia karty na otvorenie dverí
- chceme v nich vyhľadávať, overovať platnosť,...
- uložené v pamäti, neskôr v databáze
- zobrazované v okienkovej aplikácii, neskôr na webovej stránke



Čo sú tvoje dáta?

- systém sa točí okolo používateľov = **entít**
- typické otázky:
 - Ako **vyhľadám** konkrétnu entitu?
 - ako získam konkrétnu entitu z databázy?
 - Ako zistím, že konkrétna entita v databáze je tá istá ako v ramke?
 - Ako **zobrazím všetky** entity?
 - Ako môžem založiť **novú** entitu a udržiavať ju v dátovom úložisku?
 - dátové úložisko: databáza, súbor...

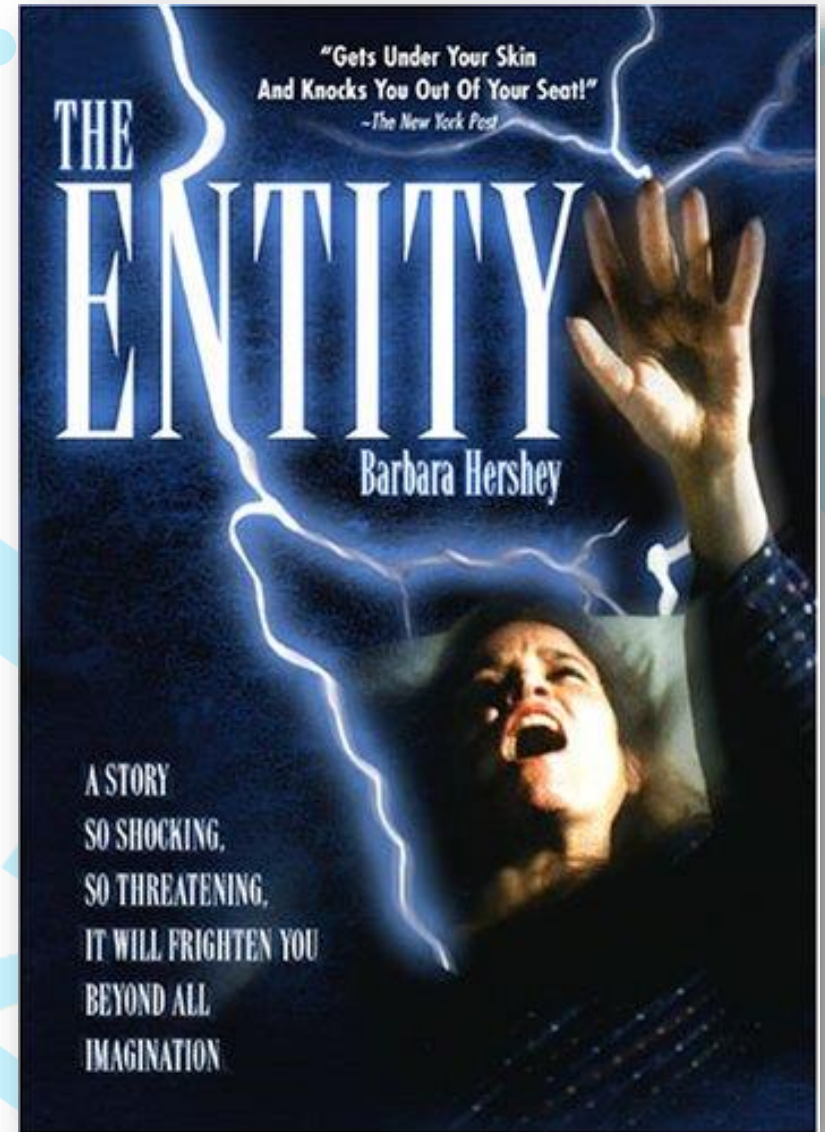
Entita: základný element domény

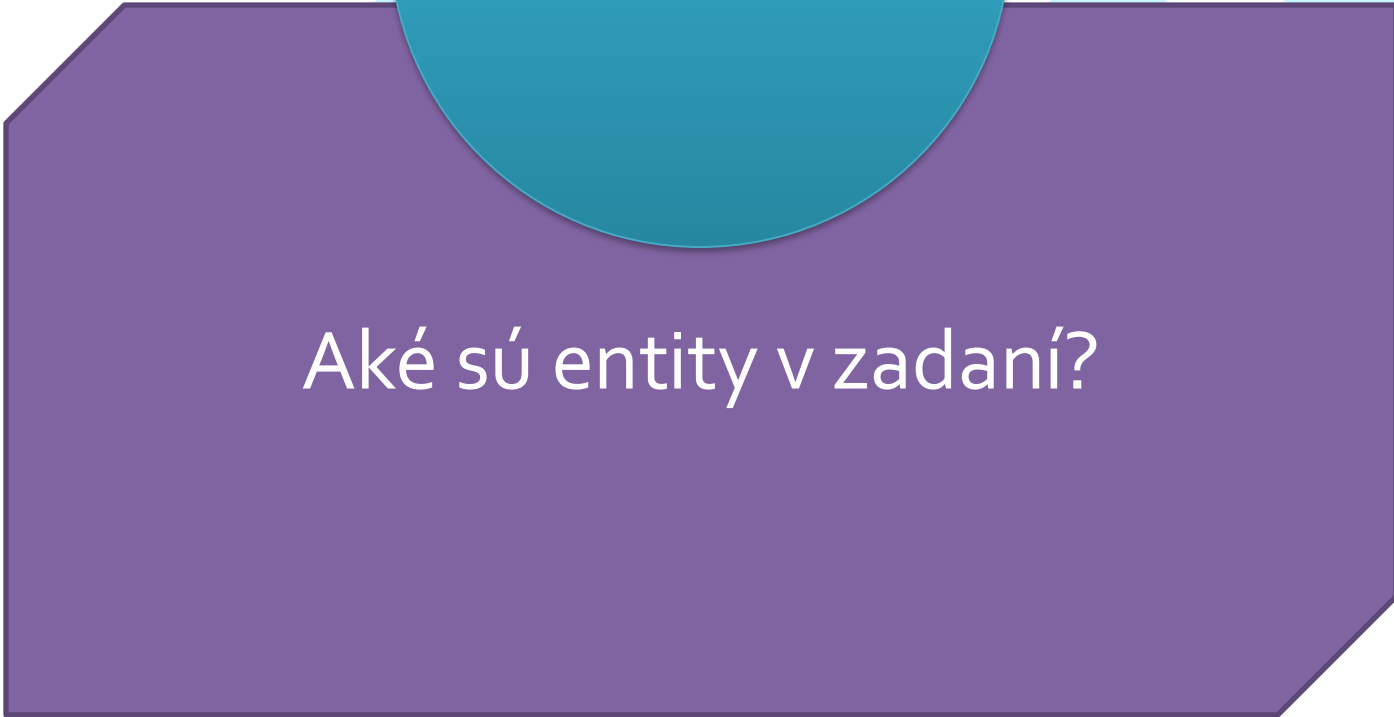
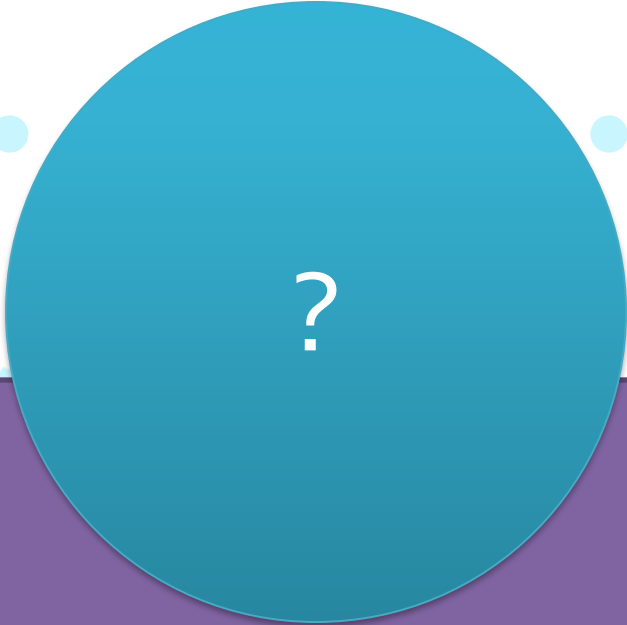
- **základný element** domény
 - **doména**: oblasť, ktorú sa snažíme modelovať v informačnom systéme
- obvykle zodpovedá základným objektom sveta v doméne
 - podstatné mená!
 - pri **prvotnej identifikácii** tried zvykneme identifikovať **práve entity**

študent
citát
automobil
záverečná práca
status pacient
identifikačný preukaz
tovar
faktúra
objednávka
zamestnanec

Entita: základný element domény

- nesie v sebe **dáta**
 - neraz: 1 entita = 1 objekt počas behu = 1 riadok v databázovej tabuľke
- má svoju **identitu**
- putuje **naprieč systémom**





?

Aké sú entity v zadaní?

Entita: **používateľ**

Schopnosti:

- overenie aktívnosti ?

???

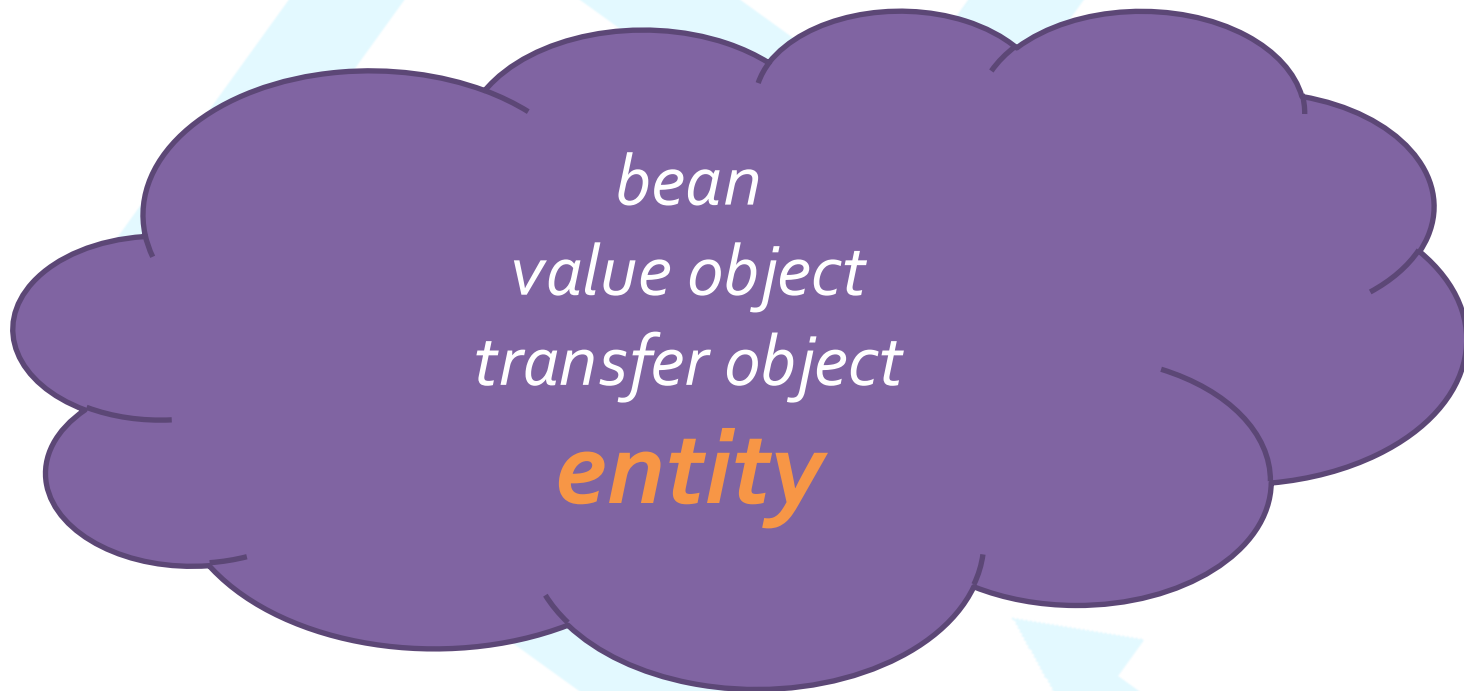
Stav:

- kód čipu
- meno
- aktívnosť
- čas posledného prihlásenia?
- ...

Entity zvyčajne nemajú žiadne rozumné schopnosti

Nie každá trieda musí mať schopnosti

- existujú triedy **bez** rozumných schopností
 - getters/setters sa nerátajú
- nie je to chyba v návrhu



Trieda User

- Aké máme inštančné premenné?
 - chipId: String
 - name: String
 - active: boolean
- Vieme z toho určiť identitu?
 - Vieme odlíšiť dve entity od seba?

Ako vyriešiť identitu?

1. využiť **prirodzený identifikátor**

- človek: meno nejednoznačné
- id čipu mení sa v čase

2. nájsť **sadu atribútov**, ktorá jednoznačne identifikuje entitu

3. vyrobiť **umelý identifikátor**

- ID INTEGER NOT NULL AUTOINCREMENT
- UUID – 128bit číslo (int – 32bit, long - 64bit)
 - v1 - čas* + MAC adresa
 - v4 - náhodné
 - v6/v7 - čas** + MAC/náhodnosť - najlepšie pre DB z hľadiska rýchlosti

Trieda User

- Teda máme inšt. premenné
 - Plus gettre, settre
- Mali by sme vždy prekryť **equals** a **hashCode**
 - Defaultne tieto metódy korektne fungujú iba ak porovnávame tú istú inštanciu
 - Ale entita môže mať rôzne inštancie objektu predstavujúce tú istú osobu
 - Môžeme nechať vygenerovať IDE
 - Alebo použijeme anotáciu **@Data** z knižnice Lombok (staršie projekty, no stále aktuálne)
 - Alebo použijeme **record class** (novšie projekty)

User

```
@With  
public record User (  
    Long id,  
  
    String name,  
    String chipId,  
    List<ChipReader> access,  
    isAdmin,  
    isActive  
) {}
```

Štyri základné operácie s entitami

C-reate

- vytvorenie novej inštancie entity

R-ead

- načítanie podľa kritérií

U-pdate

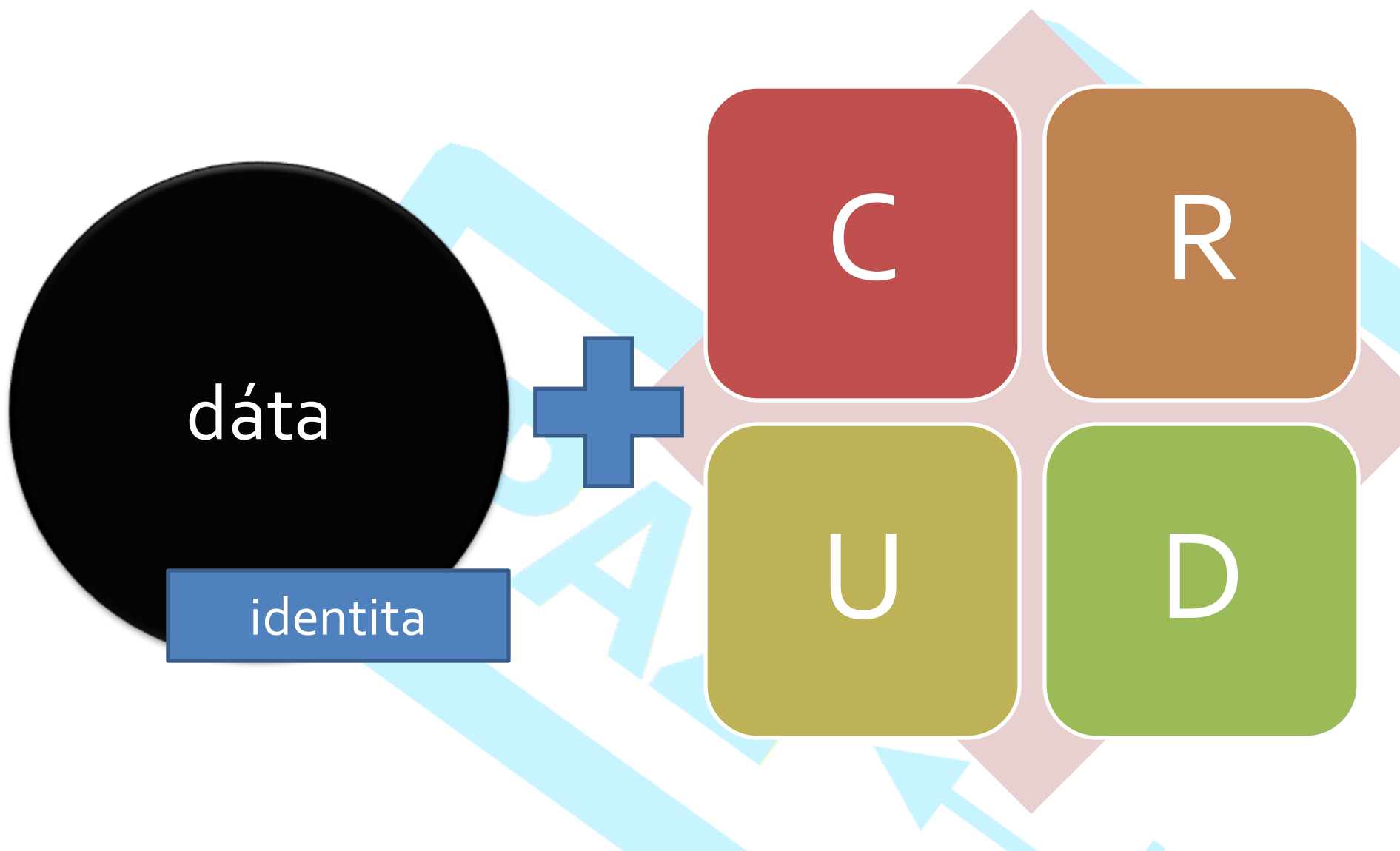
- aktualizácia entity

D-elete

- odstránenie

CRUD!

Entita + operácie s ňou





CRUD

rodina aplikácií

- málo biznis logiky
- väčšina sa točí okolo práce so súbormi alebo databázou

Ako to vyriešiť v Java?

Data Access Object [DAO]

```
public class UserDao {  
    List<User> findAll() {...}  
    User findById(Long id) {...}  
    List<User> findByName(String name) {...}  
    User findByIdChipId(String chipId) {...}  
    List<User> findByIsActive(boolean isActive)  
  
    User create(User u) {...}  
    User update(User u) {...}  
    void delete(Long id) {...}  
}
```

R

C

U

D

Implementácie

- hlavičky metód hovoria, **ČO** chceme spraviť
- telá metód hovoria **AKO** to spraviť?

odkiaľ ich
načítavať?

ako (a kam)
uložiť
používateľa?

ako má
fungovať
vyhľadávanie?



DAO = Data Access Objects

- sprístupňuje
dátové úložisko
 - obvykle databázu
 - nezriedka súbor(y)
 - v testoch často úložisko v pamäti

„Abstrahuje a zapúzdruje všetok prístup k dátovému zdroju. DAO spravuje pripojenia k dátovému zdroju, skrz ktoré získava a ukladá dáta.“

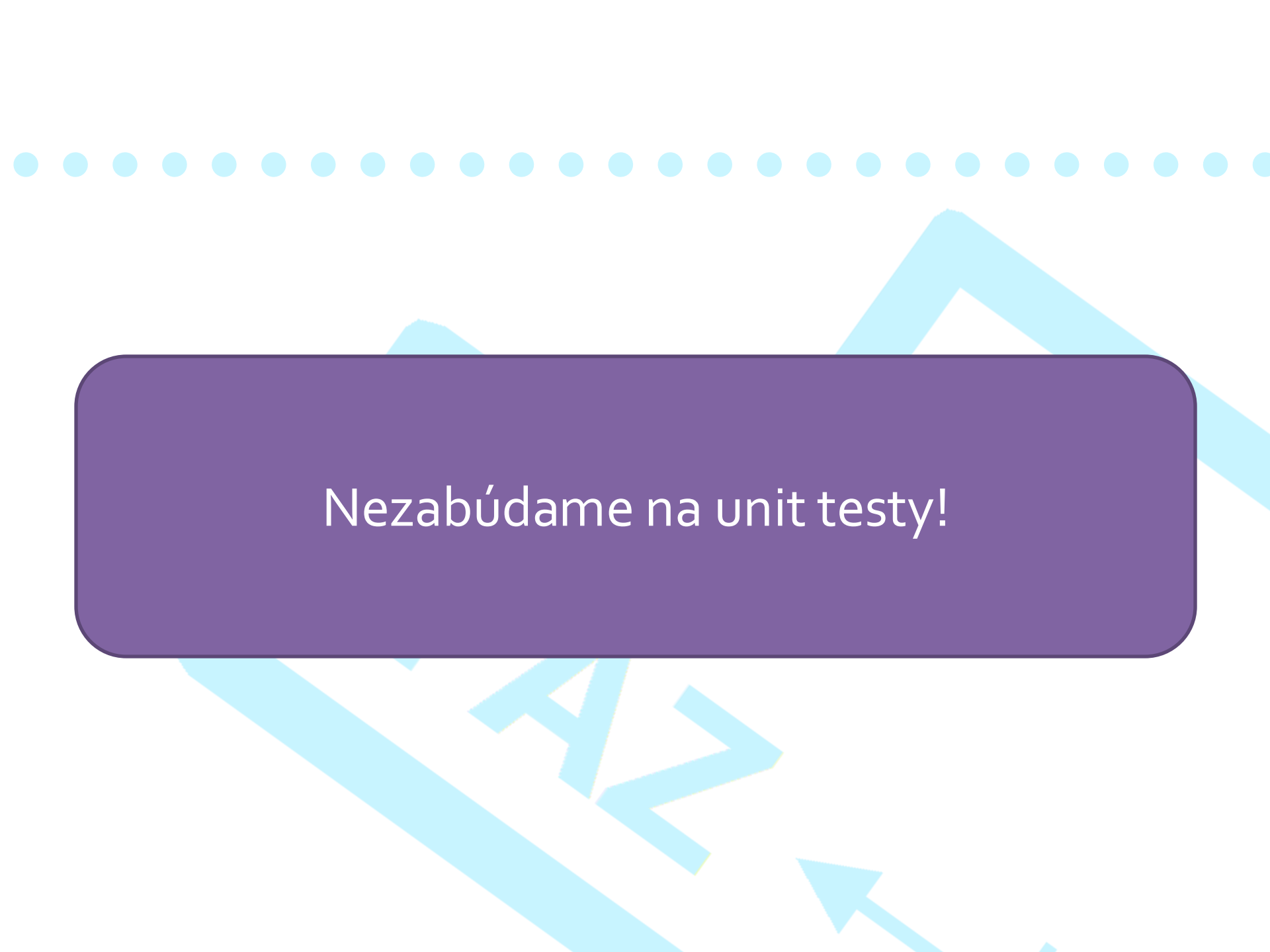
Náš UserList je základ pre pamäťové DAO

```
public class UserDao
{
    private List<User> users = new ArrayList<User>();

    public User findById(Long id) {
        /* prejdeme zoznam users, nájdeme záznam
           s daným ID
        */
    }
    void create(User user) {
        /* pridáme používateľa do zoznamu users
        */
    }
    /* ... ďalšie metódy ... */
}
```

Testy DAO

- testy sa riadia AAA (arrange-act-assert)
 - nachystaj testovacie dáta
 - otestuj funkcionálnosť
 - zisti, či sa dáta zhodujú s výsledkom
- pripravíme si **testovací** vstup
- pripravíme si **očakávané dáta**
- otestujeme, či metóda **vracia, čo čakáme**
 - assertXXX

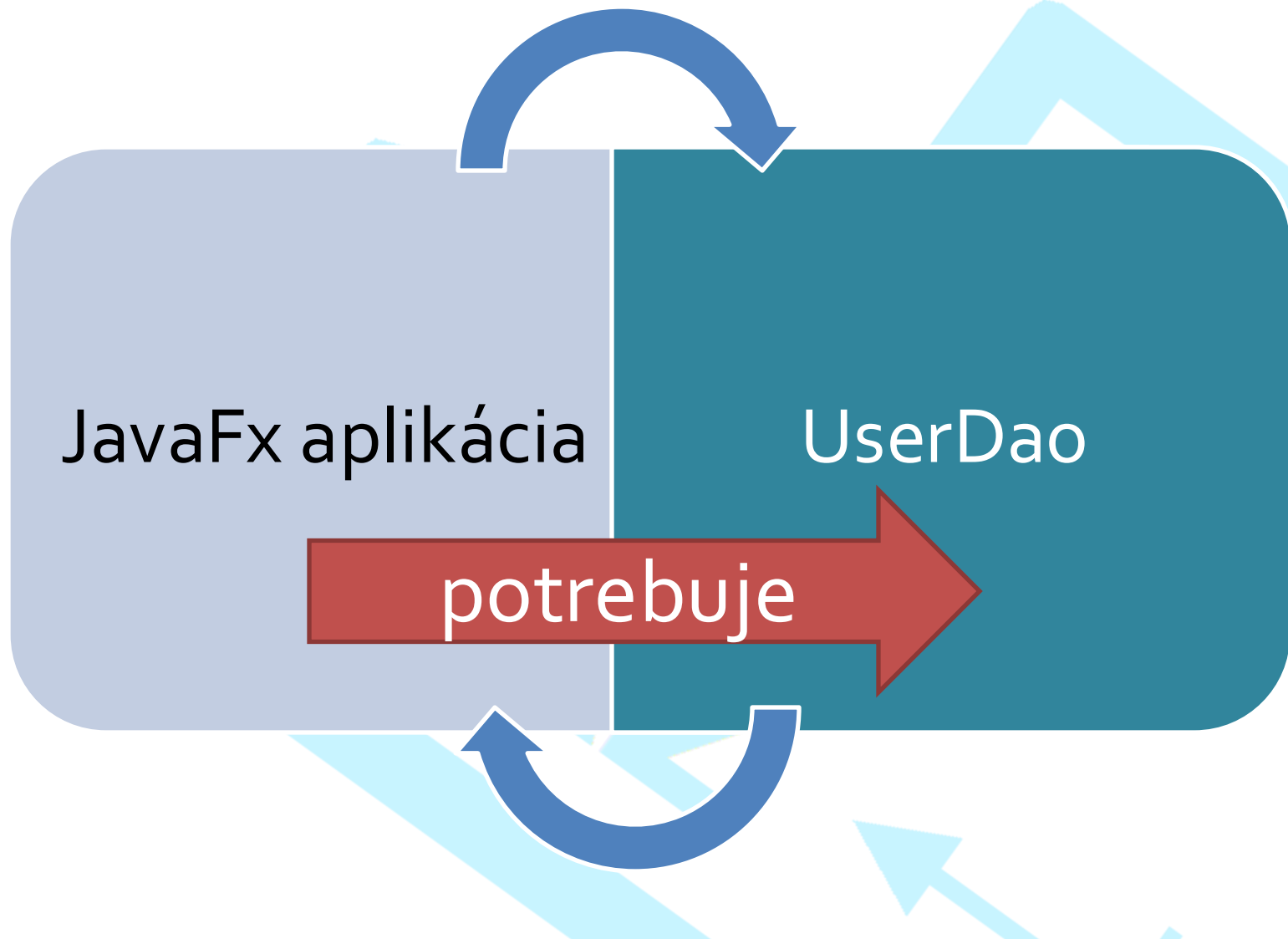
The background features a horizontal dotted line at the top and several large, light blue abstract shapes, including a zigzag line and an arrow, scattered across the slide.

Nezabúdame na unit testy!

Testy

```
public UserDaoTest {  
    private UserDao userDao;  
  
    @Test  
    public void testFindById() {  
        /* ... */  
    }  
  
    @Test  
    public void testFindByName() {  
        /* ... */  
    }  
  
    /*.....*/  
}
```

Ako prepojiť UI s existujúcim kódom?



Vzťah „závisí na“ (dependency)

- trieda **potrebuje** pre svoje **fungovanie** inú triedu
- **typicky**: implementácia v objekte **nebude** fungovať, ak nemá k dispozícii iný objekt
- príklad:
 - DAO je **kolaborátorom / závislosťou** kontroléra



Vzťah „závisí na“ (dependency)

- riešenie: inštančná premenná

```
public class EntranceController {  
    private UserDao userDao;  
    ...  
}
```

Ako prepojiť inštancie
za behu?

Čo s gettrami a
settrami?

```
public class UserDao  
{  
    ...  
}
```

Trieda si vytvorí závislosť sama

```
public class EntranceController {  
    private UserDao userDao = new UserDao();  
    ...  
}
```

- **problém:** trieda sa zadrôtuje implementáciou!
- vhodné, ak naozaj vieme, že sa implementácia **meniť nebude**
 - ak nikdy nebudeme migrovať na súborový UserDao...

Trieda si vytvorí závislosť sama

```
public class EntranceController {  
    private UserDao userDao = new UserDao();  
    ...  
}
```

- čo s gettrami a settrami?
- getter zvyčajne nie je potrebný
 - za predpokladu správnej architektúry kódu
- niekedy sa zverejňuje setter
 - cez neho možno „prebiť“ implementáciu

Základné riešenie ťažkého kalibru

- ľahko sa implementuje a chápe
- málo flexibilný

Čo ak chceme
načítavať dáta z
disku?

Čo s viacerými
inštanciami DAO
objektu?

Čo ak je DAO
používané z
viacerých tried?

Riešenia problémov

- zatiaľ to funguje
- **problémy** nastanú pri viacerých druhoch perzistencie, ktoré sa striedajú
 - za behu zo súboru, v testoch z pamäte
- neskôr si ukážeme možnosti riešenia
 - **továrne**

„Problém s programátormi je v tom, že neviete čo to vlastne stvárajú, až do chvíle, kým nie je príliš neskoro.“

-- Seymour Cray

Take-away

- identifikujeme, či je aplikácia **CRUD**
- identifikujeme **entity**
- navrhujeme **DAO** objekty tak, aby reprezentovali operácie prístupu k dátam
 - **CRUD**
- cez inštančné premenné realizujeme pevné závislosti
- navrhujeme tak **dvojvrstvovú** aplikáciu



Otázky?