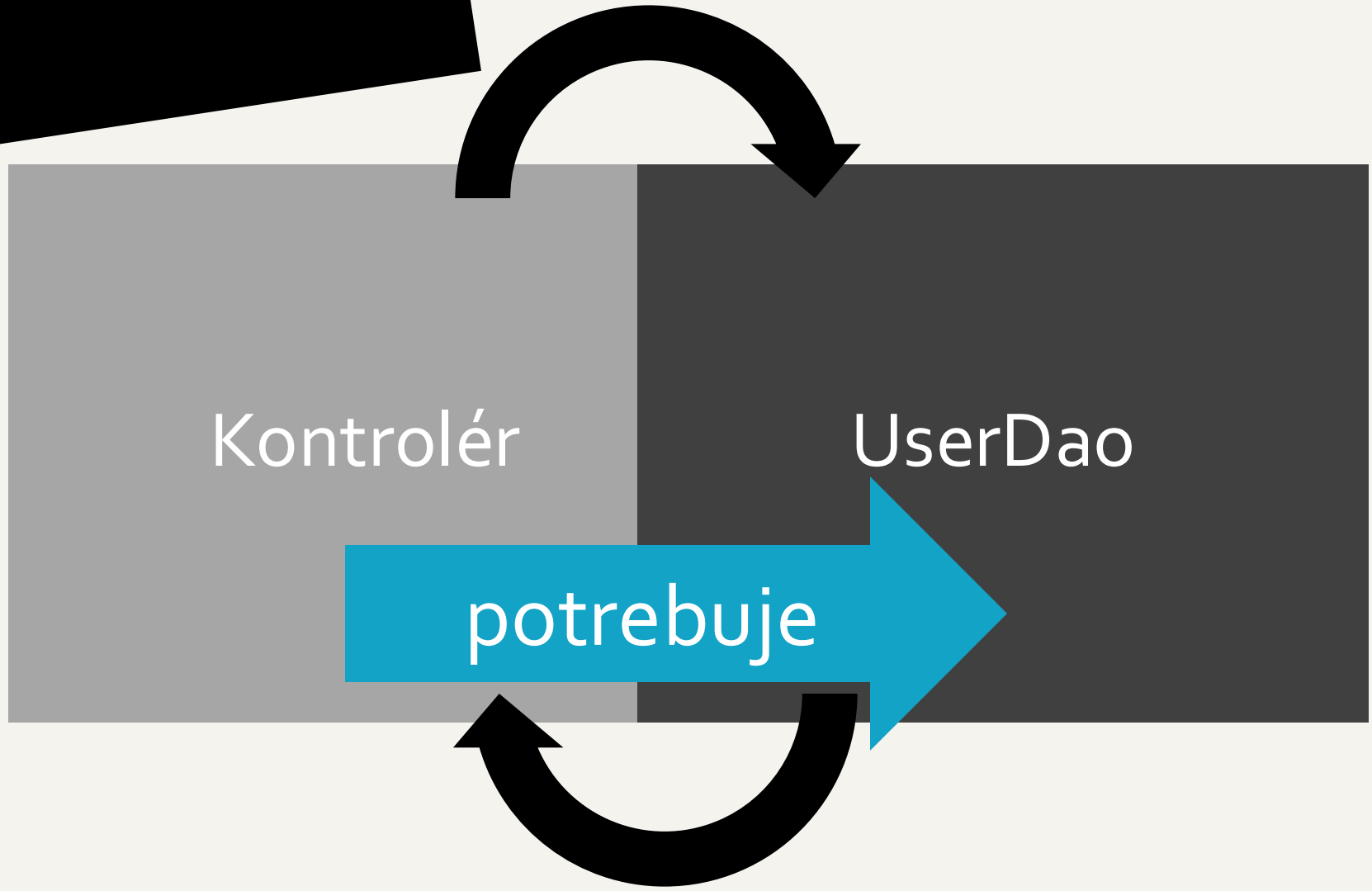
Kontrakty, Továrne, Jedináčikovia, Vymenované typy, Databázy a JDBCTemplate, Docker

UINF/PAZ1c

4. prednáška

prepojenie GUI a DAO



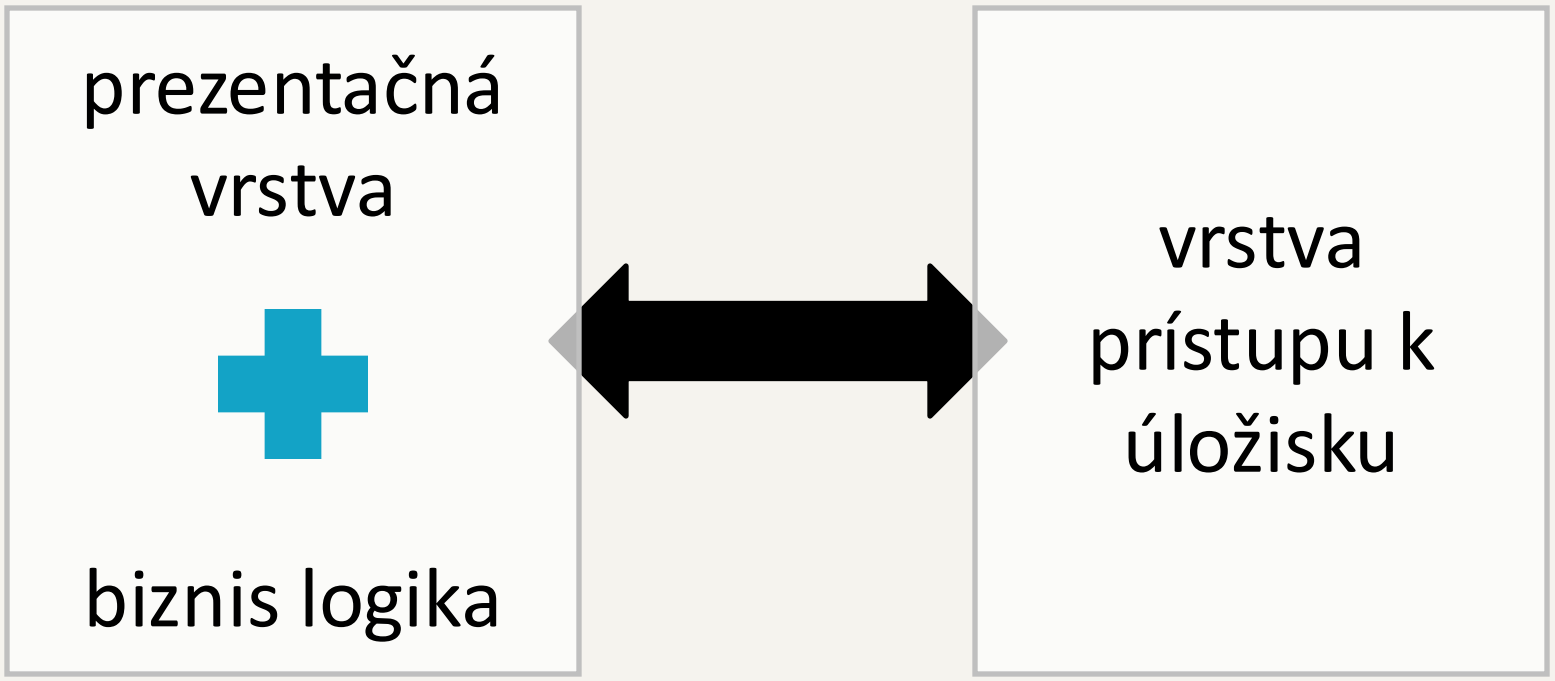
dependency

objekt vyžaduje pre fungovanie iný objekt

kolaborácia

závislosť

2vrstvové aplikácie



kód: inštančná premenná

```
class EntranceMainController {
  private UserDao userDao;
   ...
}
```

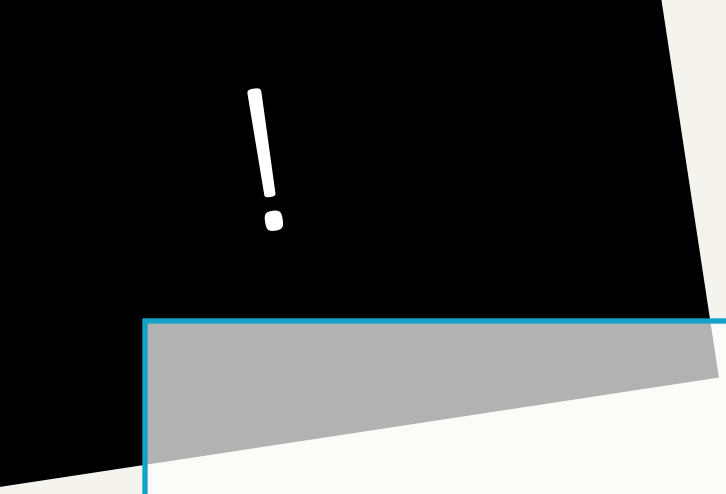
```
class UserDao {
   ...
}
```

kód

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

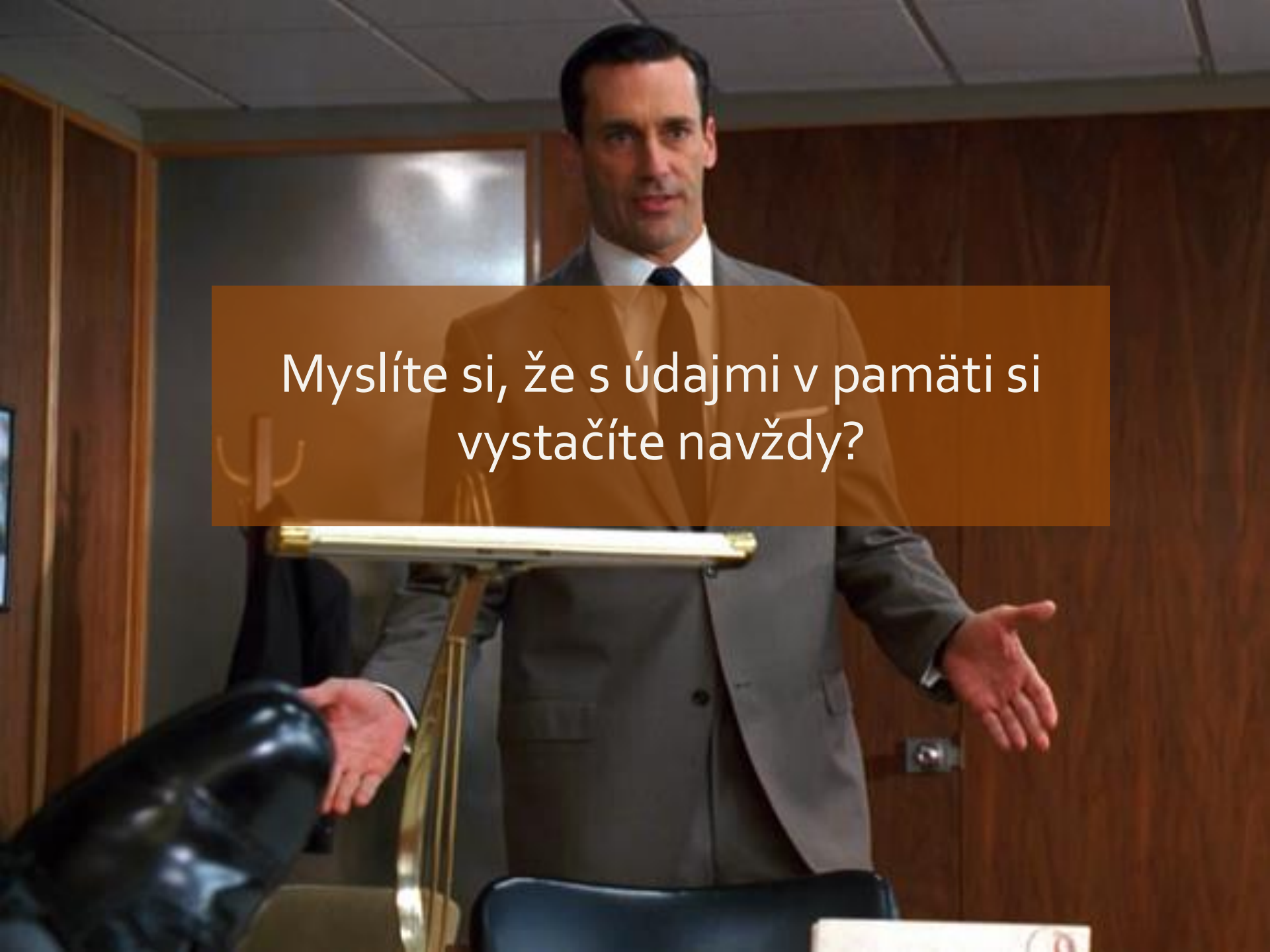
trieda si vytvorí závislosť sama

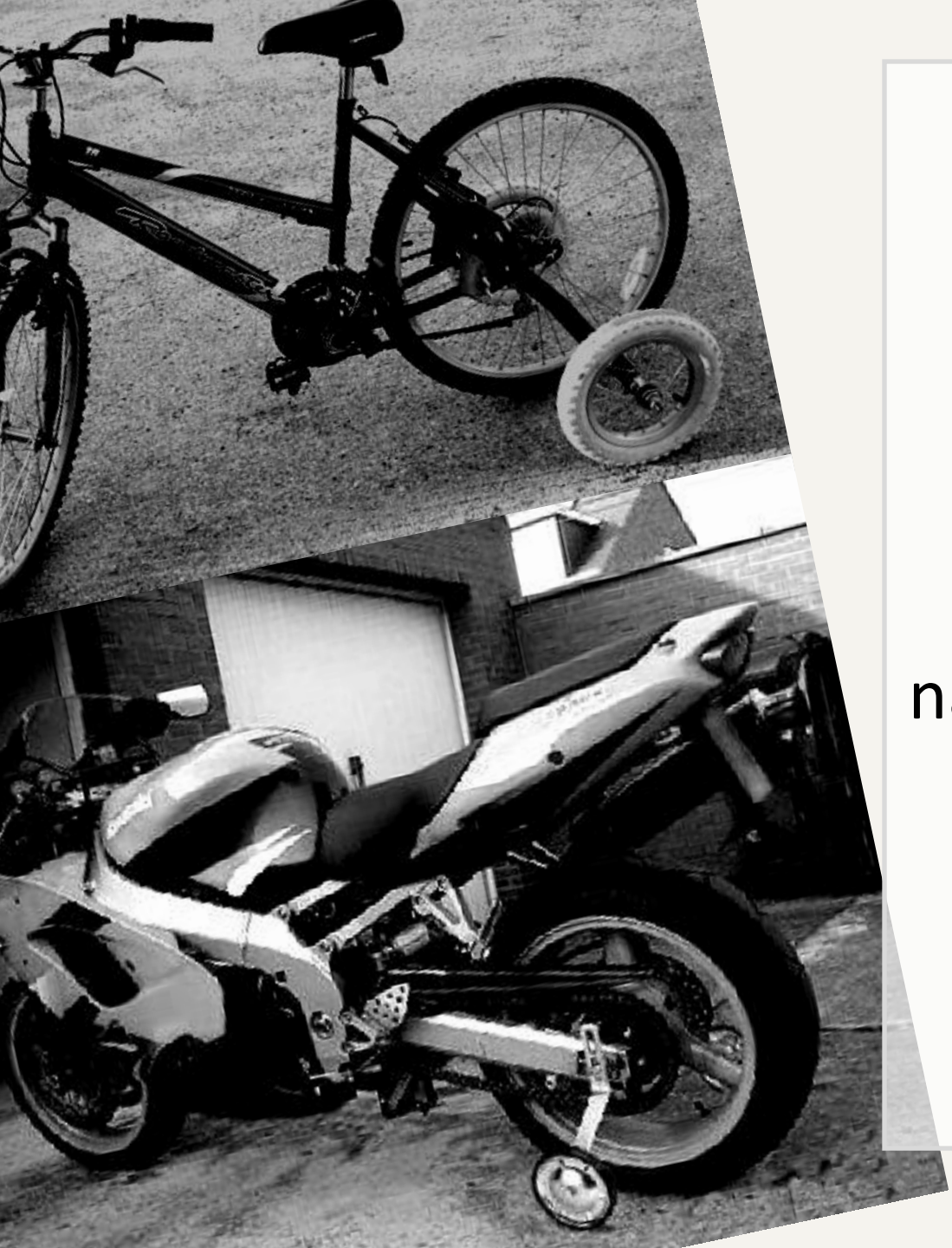
„jeden modul programu vyrába iný“



trieda sa zadrôtuje s implementáciou

čo ak chceme implementáciu meniť?


Myslíte si, že s údajmi v pamäti si vystačíte navždy?



migrujme
používateľov
na disk / do databázy

vymieňame perzistentnú vrstvu!

refaktor


zahodíme UserDao nad pamäťou?


uvažujme
úplne inak

trieda je čierna skrinka

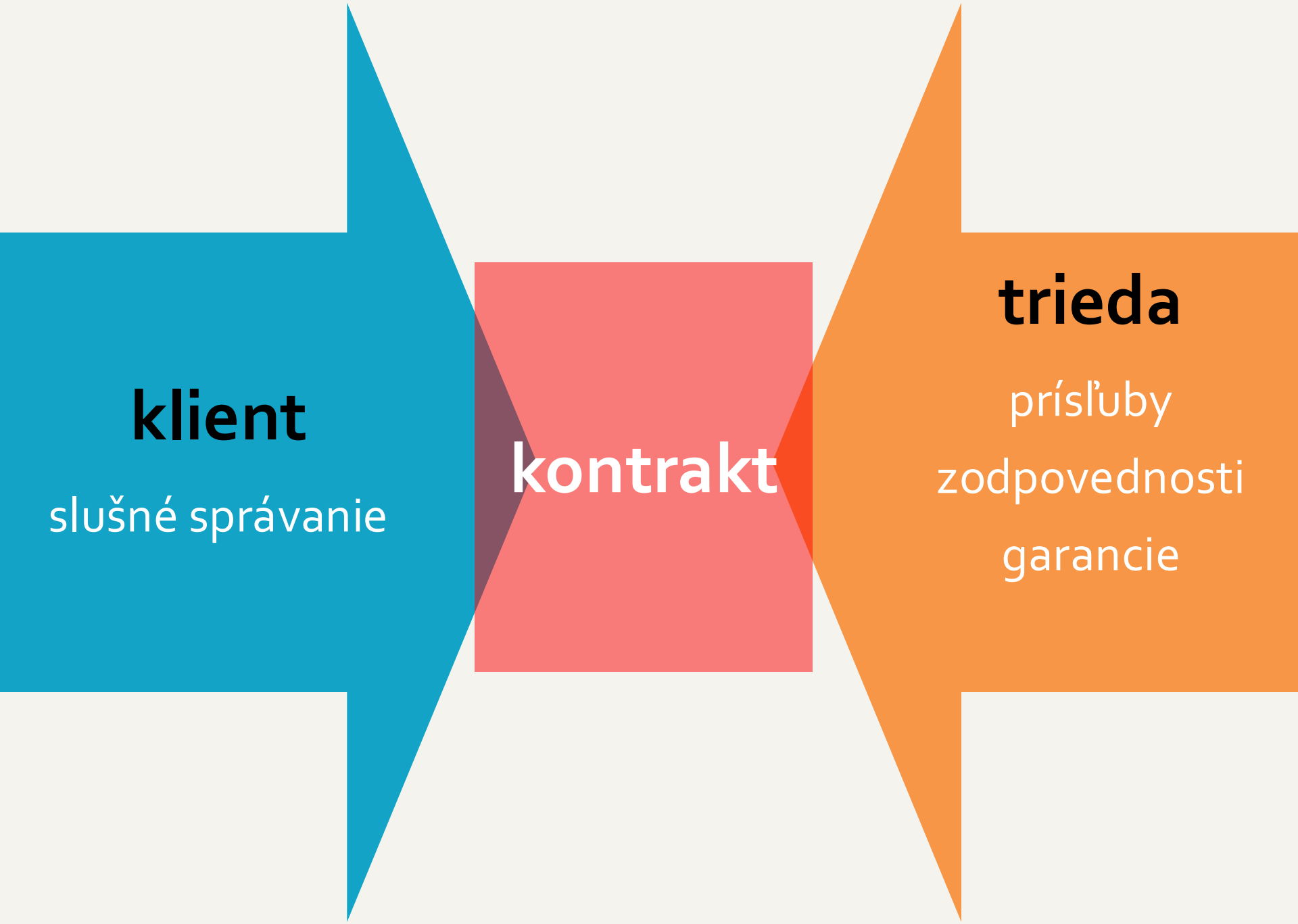
čo, nie ako

pri návrhu triedy začíname
od schopností

implementácia až v druhom
rade

spomeňme na TDD – najprv testy
nad prázdnymi metódami

vonkajší pohľad na triedu


klient
slušné správanie
kontrakt
trieda
prísľuby
zodpovednosti
garancie

interfejs

sada operácií, ktoré objekt poskytuje navonok

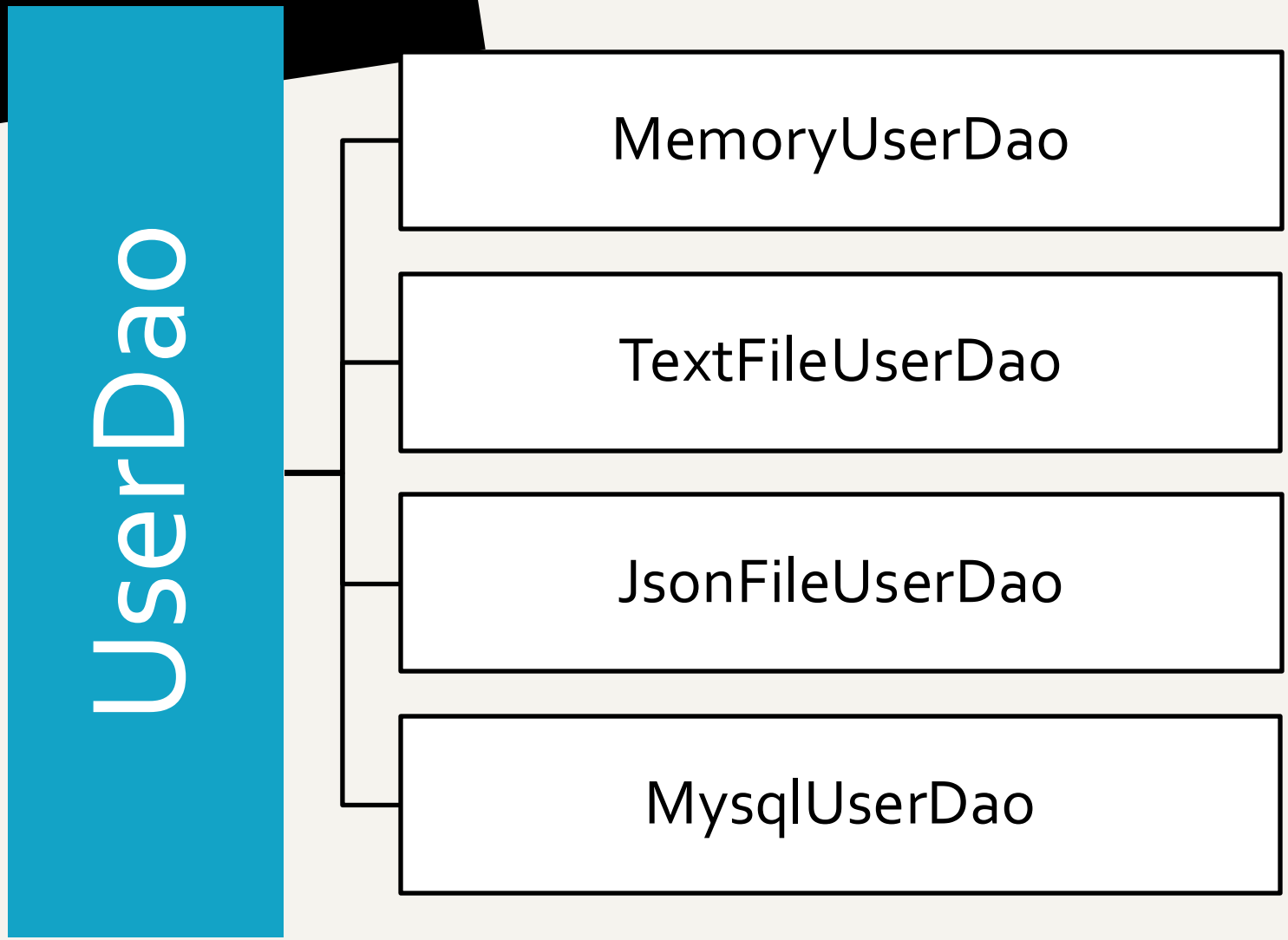
= rozhranie
= protokol
= kontrakt

```
public interface UserDao {
  User findById(Integer id);
  List<User> findByName(String name);
  User findByChipId(String chipId);
  List<User> activeUsers();
  void save(User user);
  void delete(User user);
}
```

ktokoľvek vie naplniť kontrakt

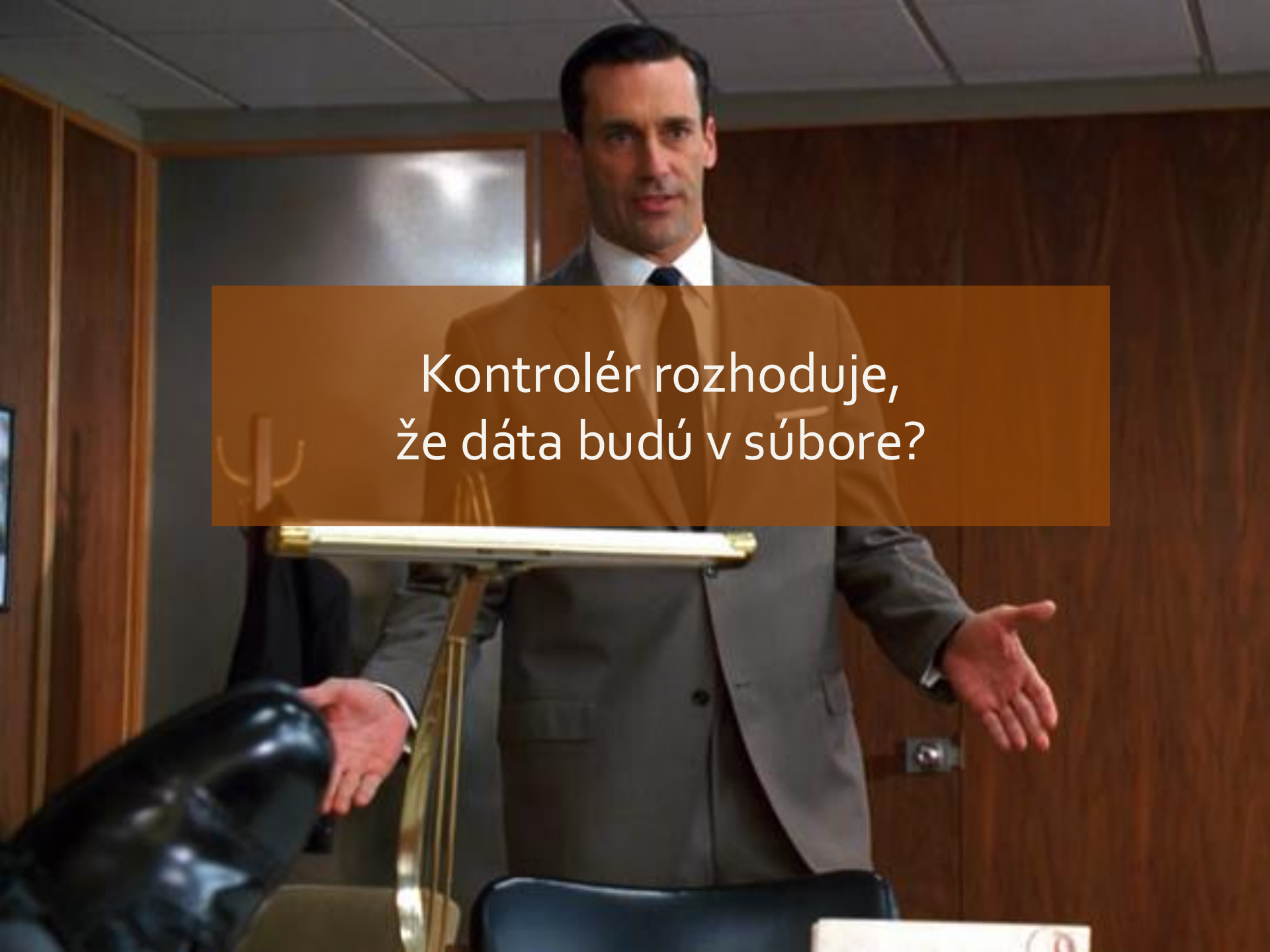
```
public class MemoryUserDao implements UserDao
{
    private List<User> users  = new ArrayList<>();

    public User findById(Integer id) {
      for(int i = 0; i < users.size(); i++)
          if (users.get(i).getId()==id)
              return users.get(i);
      return null;
    }
    /* implementácie ostatných metód */
}
```

ktokoľvek vie naplniť kontrakt



použitie

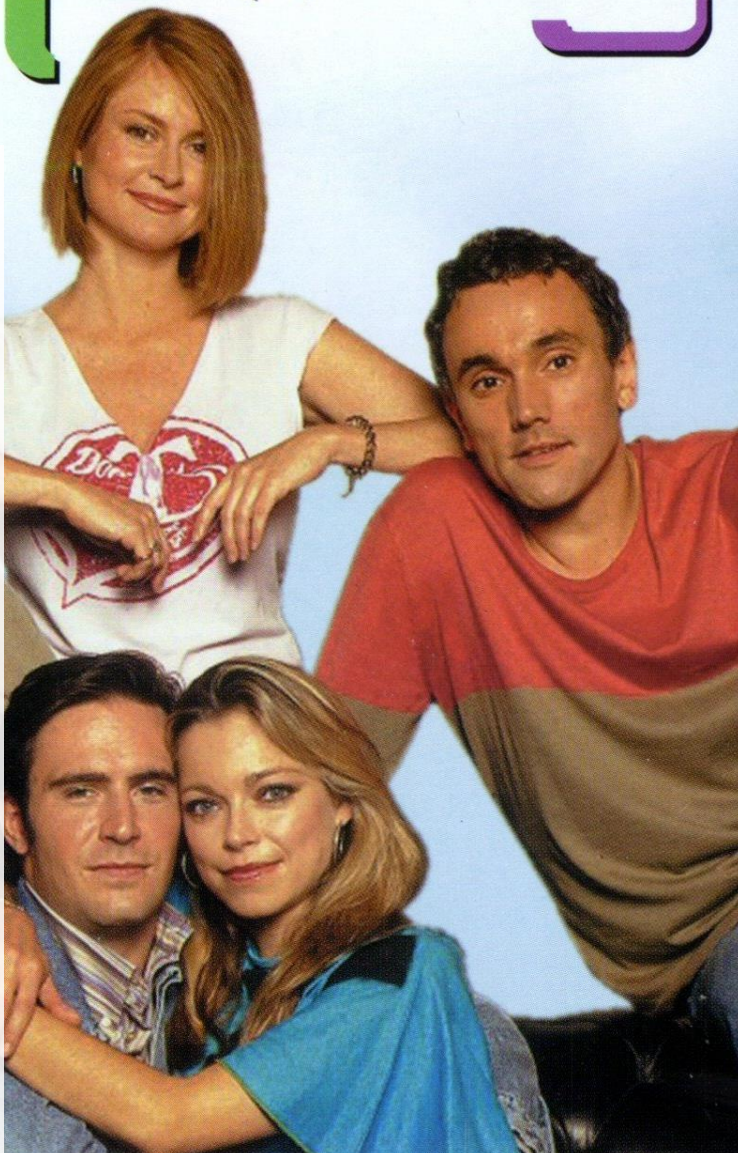
```
public class EntranceMainController {
  private UserDao userDao
        = new TextFileUserDao();
  ...
}
```


Kontrolér rozhoduje,
že dáta budú v súbore?

Čo ak máme viac tried v GUI?

- Viac scén a kontrolérov, „vyskakovacie okná“
- Každá trieda z GUI bude mať inštančnú premennú DAO
  - Bude si každá trieda GUI vyrábať vlastnú perzistentnú vrstvu?
  - Budú si musieť objekty hovoriť medzi sebou referenciu na spoločné DAO?
    - A ktorý z nich ju má vyrábať?

Tight Coupling



- úzka zviazanosť tried
- priveľká závislosť
- krehké pri zmenách

Nevolajte nám, my sa vám ozveme.

Trieda poprosí niekoho, nech
jej vyrobí a dodá kontraktorov.

továreň

objekt, ktorý vyrába iné objekty

továreň

- Rozhoduje, ktorá implementácia interfejsu sa vyberie
  - pamäť, súbor, databáza,...
- **Našteluje danú implementáciu**
  - login, heslo, umiestnenie, ...

...Factory

```
public class UserDaoFactory {
  public UserDao getUserDao() {
    /* tu sa vráti inštancia vybranej
       implementácie UserDao*/
  }
}
```

getUserDao()

nový objekt?

rovnaký objekt?

z poolu?

akého typu?

použitie továrne

kontrolér potrebuje DAO

továreň vytvorí DAO

kontrolér potrebuje továreň

trieda potrebuje továreň

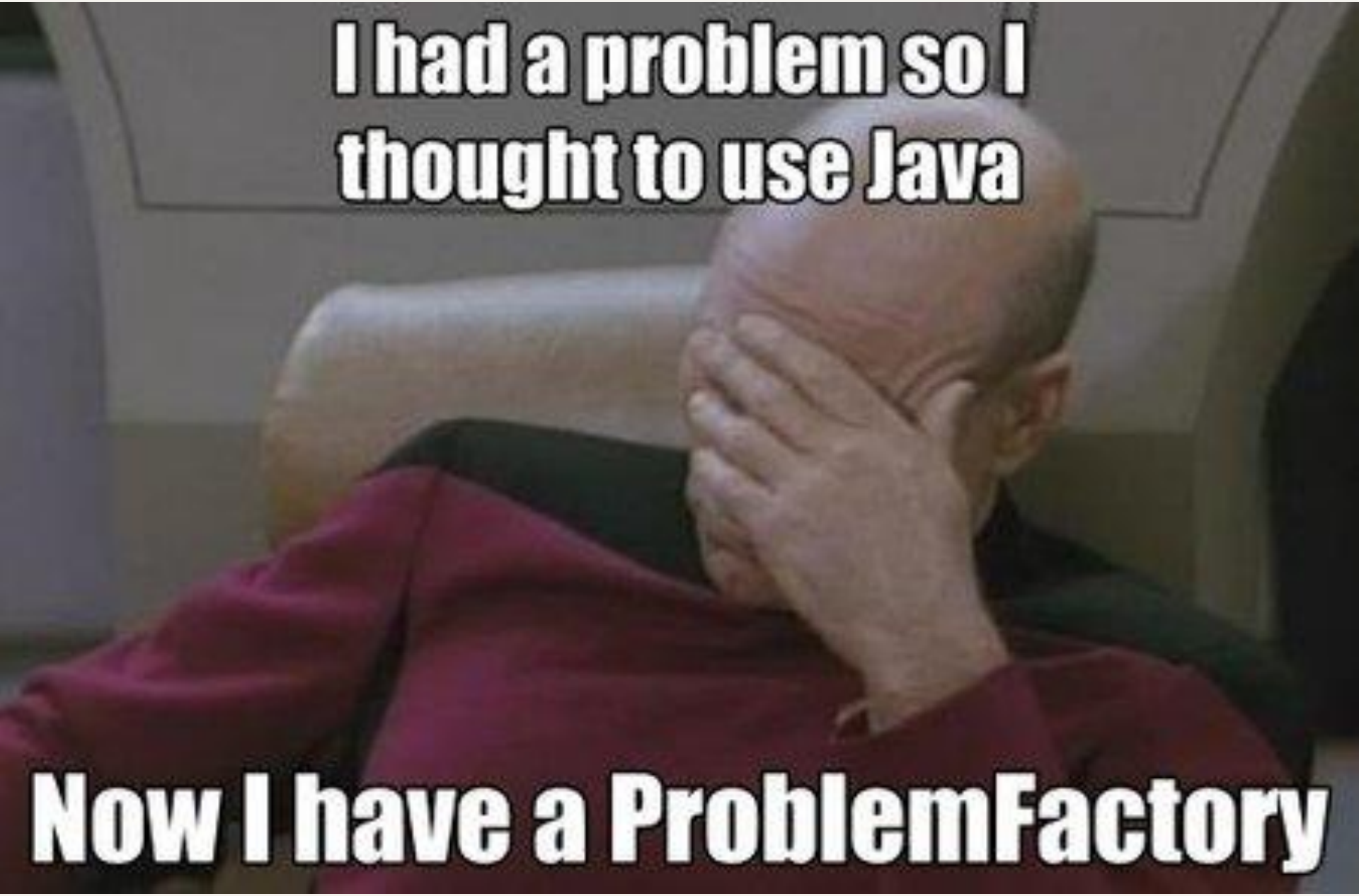
```
public class EntranceMainController {
    private UserDaoFactory factory
                = new UserDaoFactory();
}
```

trieda potrebuje továreň

```
public class EntranceMainController {
    private UserDaoFactory factory
                = new UserDaoFactory();
}
```

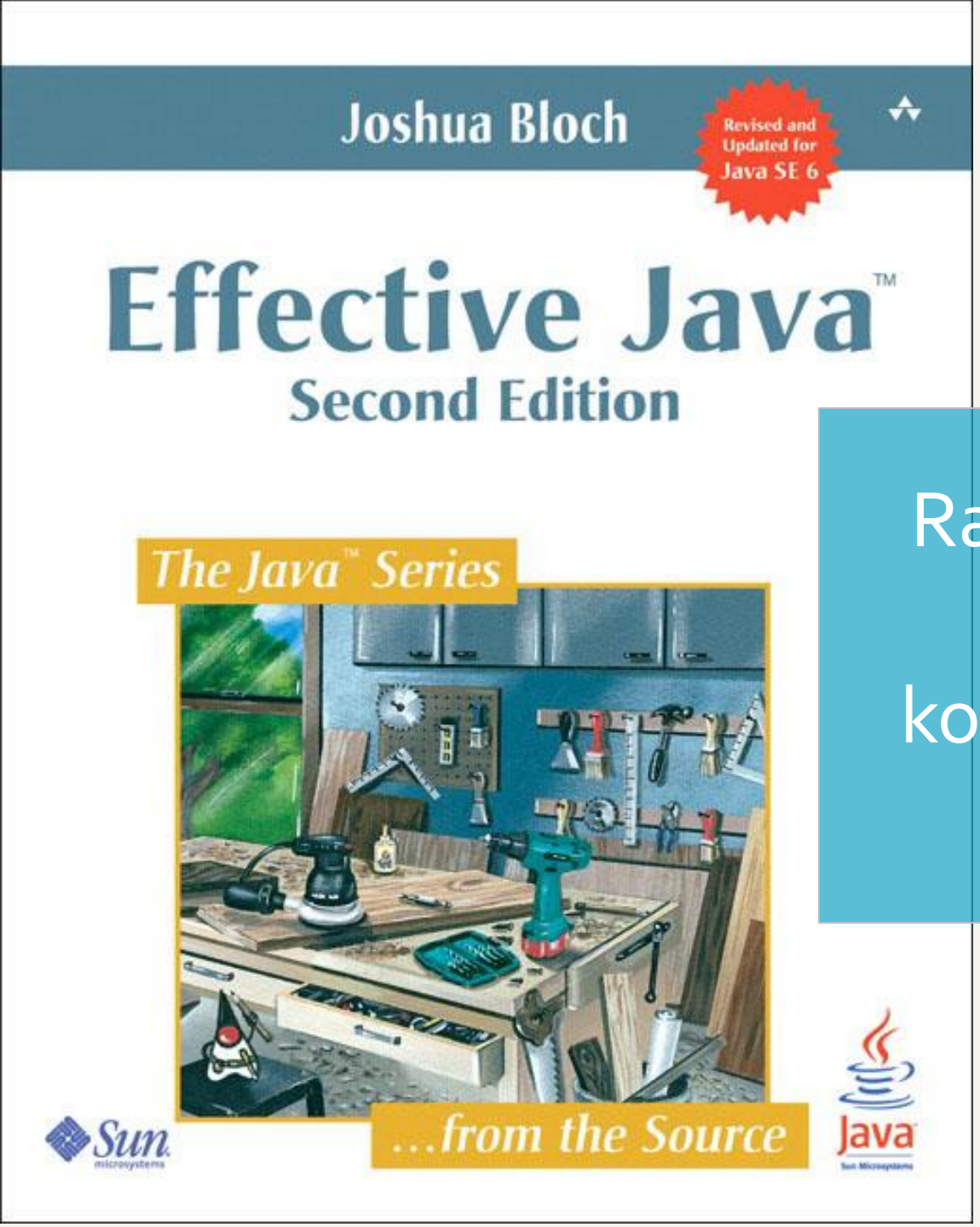
lolwut?

„Každý problém v informatike možno vyriešiť pridaním ďalšej úrovne nepriameho prístupu / abstrakcie.“
– Butler Lampson, 1972


I had a problem so I thought to use Java
Now I have a ProblemFactory


koľko tovární
potrebujeme?


FROM ANOTHER TIME COMES A MAN
OF GREAT POWER.
A MAN OF INCREDIBLE STRENGTH.
AN IMMORTAL ABOUT TO FACE HIS
GREATEST CHALLENGE...
SINGLETON
jediná inštancia v systéme
FEATURING ORIGINAL SONGS BY QUEEN
CANNON



Rada 3: singleton cez privátny konštruktor alebo enum

továreň v kóde

```
public enum UserDaoFactory {
  INSTANCE;

  public UserDao getUserDao() {
    /*...*/
  }
}
```

továreň v kóde

```
UserDao userDao =
        UserDaoFactory.INSTANCE
        .getUserDao();
```

hack,
ale
korektný

migrácia: 1 riadok!

```
public enum UserDaoFactory {
  INSTANCE;

  public UserDao getUserDao() {
    return new MemoryUserDao();
    return new MysqlUserDao();
  }
}
```

prístup k databázam

Relačné databázy

dominantný spôsob ukladania dát

hojný výskum

kilá implementácií

Naša entita User v relačnej databáze

```
CREATE TABLE user (
 id INT NOT NULL AUTO_INCREMENT,
 chip_id VARCHAR(20) NOT NULL,
 name VARCHAR(45) NOT NULL,
 active BOOLEAN,
 PRIMARY KEY (`id`)
);
```

Ako prepojiť Javu a DB?

JDBC = Java API
pre SQL

JDBC

- Funkčné, ale ťažkopádne
- Potenciálne nebezpečné
  - Nutnosť korektne obsluhovať a uvoľňovať prostriedky
  - V tutoriáloch sa kvôli prehľadnosti ignorujú závažné chyby

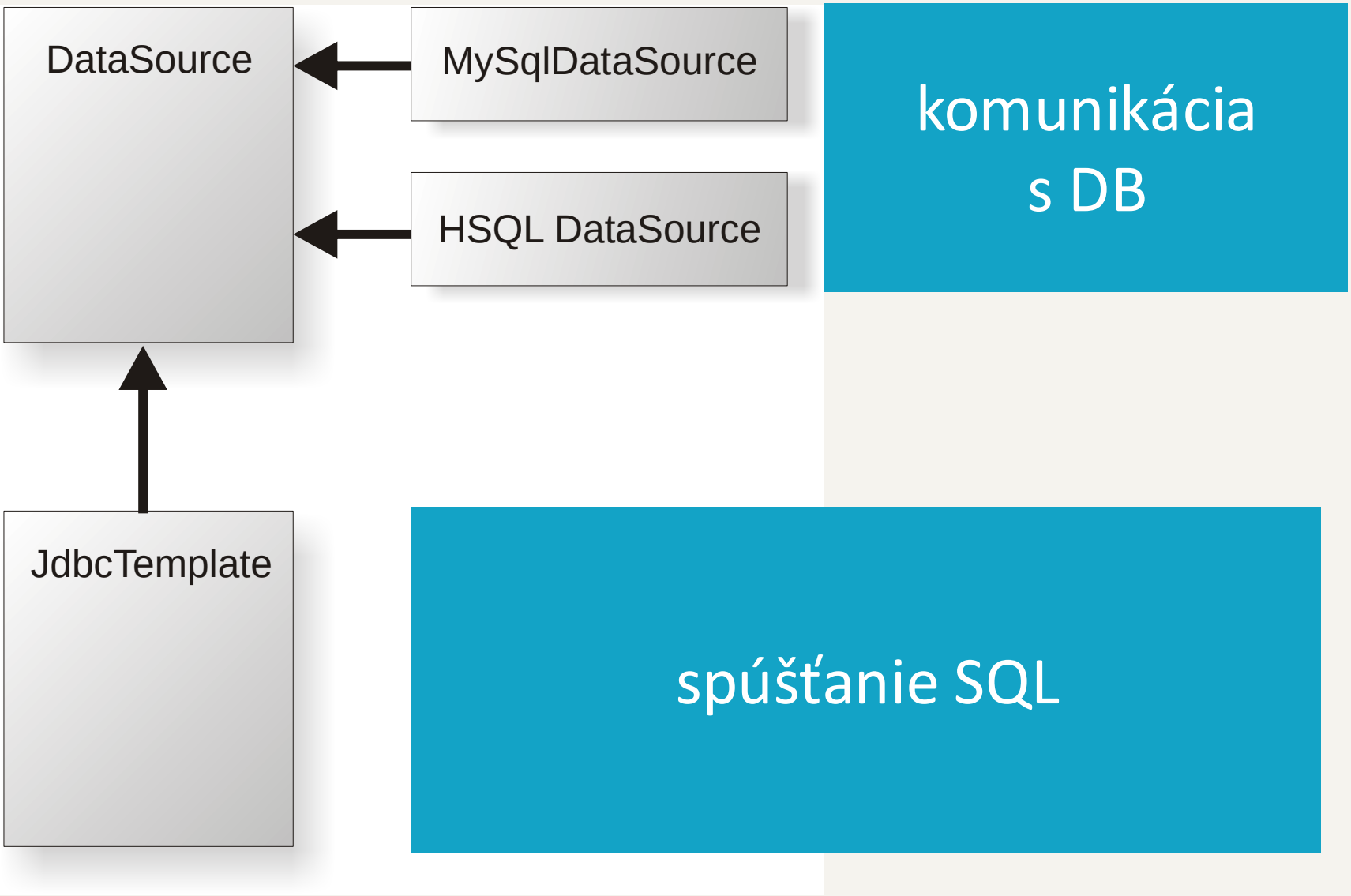
DB spojenia

- Časté vytváranie spojenia je drahé
  - Bežne chceme niekoľko (aj tisíce) dopytov za sekundu
  - Robiť všetko cez jedno trvalé spojenie je pomalé
  - Trpíme: „head of line blocking" – krátke dopyty zbytočne čakajú na predchádzajúce dlhé operácie
- Typicky databáza dovolí max. 20 - 50 paralelných spojení
  - Potrebujeme to nejako spravovať

Spring Framework

„Budujme lepšie enterprise aplikácie“

Spring JDBC


DataSource
MySqlDataSource
HSQL DataSource
komunikácia s DB
JdbcTemplate
spúšťanie SQL

JdbcTemplate v Mavene

mysql-connector-j

spring-jdbc

Inicializácia

```
MysqlDataSource dataSource = new MysqlDataSource();

dataSource.setDatabaseName(„entrance_system");
dataSource.setUser("paz1c");
dataSource.setPassword("paz1cIsGreat");

JdbcTemplate jdbcTemplate = new
                            JdbcTemplate(dataSource);
```

Dopyty

```
String sql = "SELECT id, chip_id, name, active FROM user";
RowMapper<User> rowMapper = new UserRowMapper();
List<User> users = jdbcTemplate.query(sql, rowMapper);
```

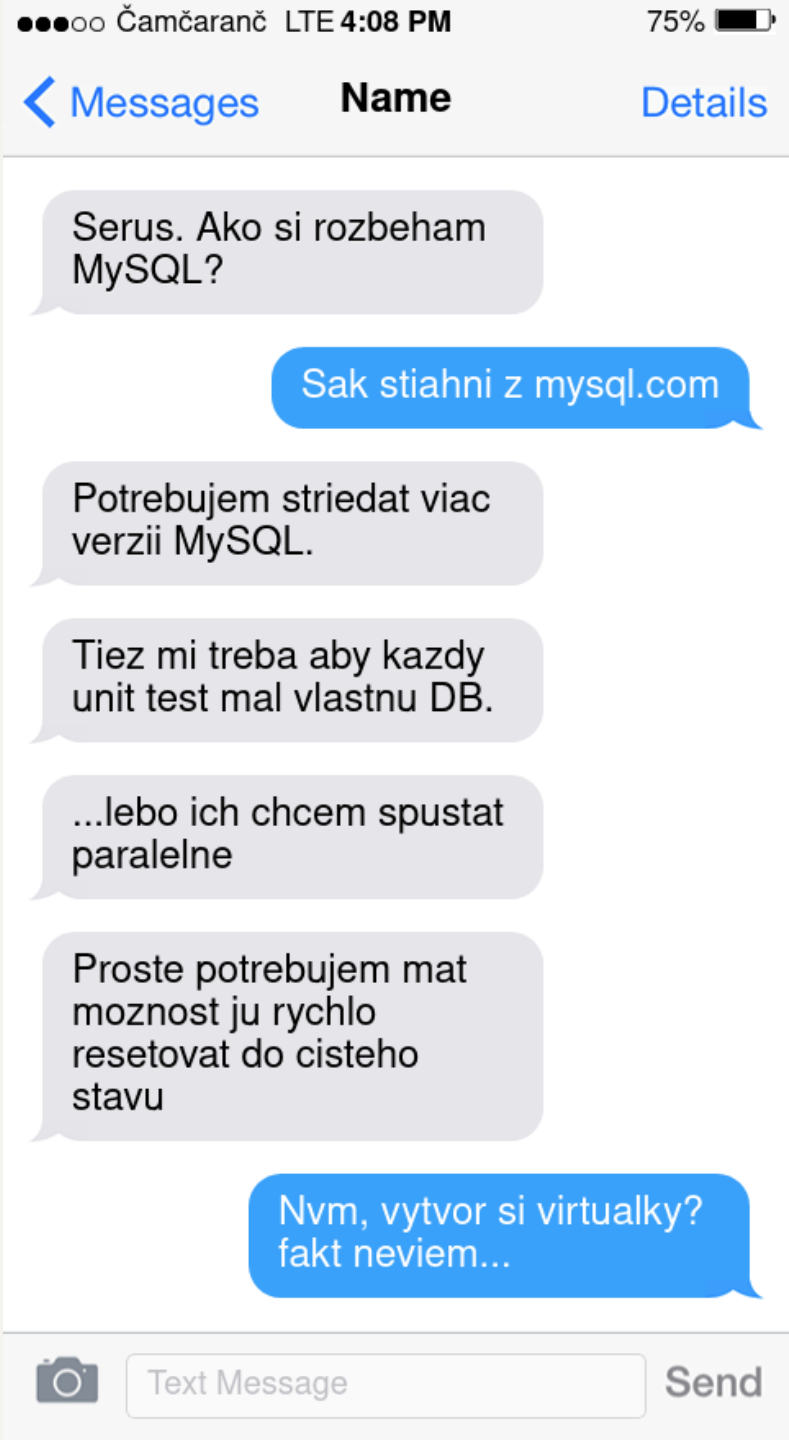
- Pre každý riadok výslednej tabuľky sa zavolá metóda mapRow() objektu rowMapper
  - mapovač vráti jednu inštanciu triedy User
- Metóda query() vyzbiera všetky inštancie a vráti ich ako zoznam

Dopyty

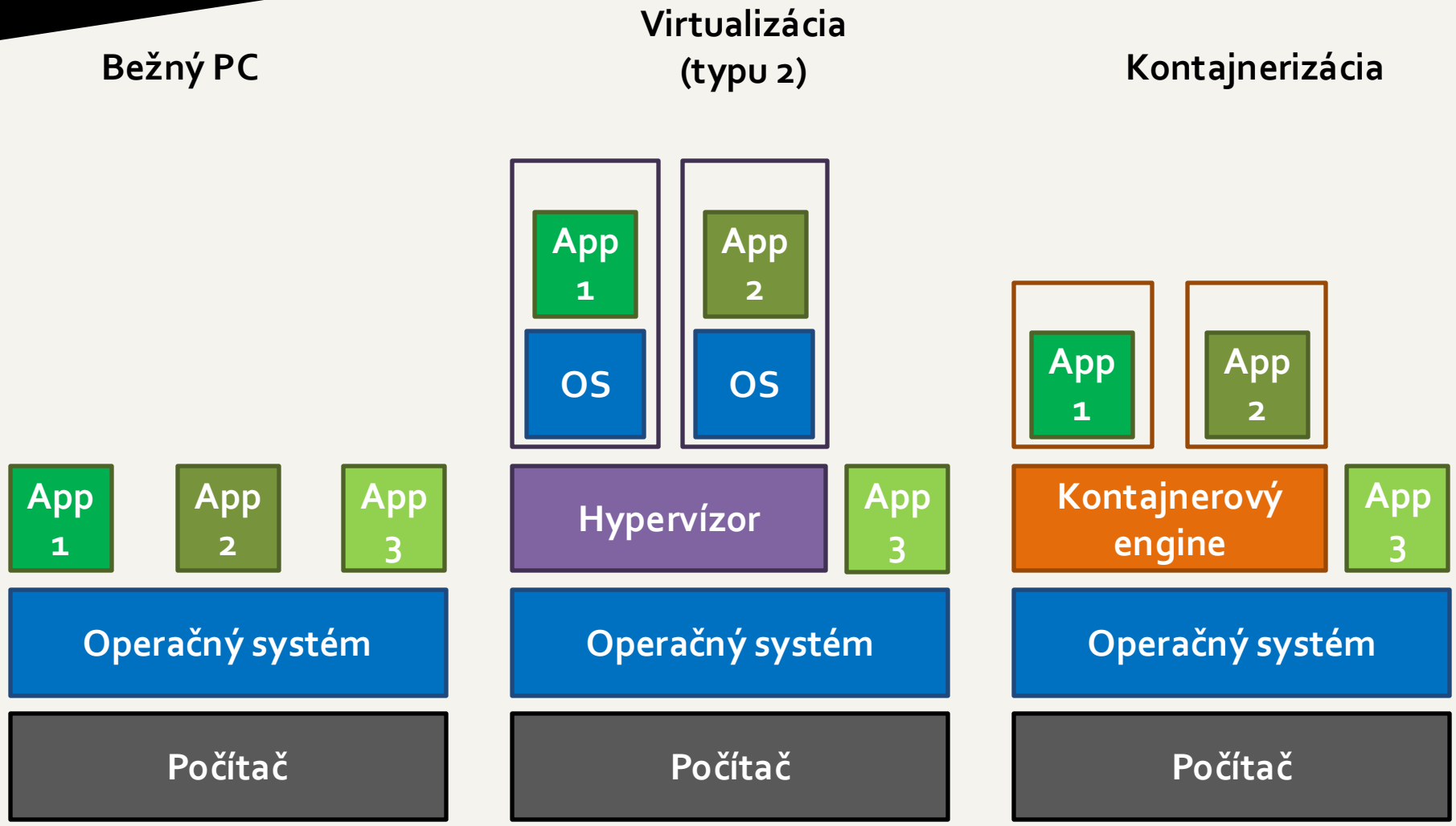
```
String sql = "SELECT id, chip_id, name, active FROM user";
RowMapper<User> rowMapper = new UserRowMapper();
List<User> users = jdbcTemplate.query(sql, rowMapper);
```

```
public class UserRowMapper implements RowMapper<User> {
  public User mapRow(ResultSet rs, int rowNum)
    throws SQLException {
      User user = new User();
      user.setId(rs.getInt("id"));
      /* ďalšie volania setterov */
      return user;
  }
}
```

Spúšťanie DB



Kontajnerizácia



- Docker je kontajnerizačný nástroj
  - kontajner - "odľahčená virtuálka"
- Moderná appka potrebuje viacero iných appiek
  - MySQL/PostgreSQL, Redis, Kafka, ElasticSearch, ...
- Docker umožní na jeden klik/príkaz tieto appky
  - sťahovať, spúšťať, vypínať, **resetovať**...
  - skvelé aj na testy (cez TestContainers knižnicu)

MySQL cez Docker Compose

docker-compose.yaml

```yaml
services:
  mysql:
    image: mysql:8.4.2
    environment:
      MYSQL_ROOT_PASSWORD: superheslo
      MYSQL_DATABASE: entrance
      MYSQL_USER: entrance
      MYSQL_PASSWORD: hesielko
    ports:
      - "3306:3306"
    volumes:
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

stiahni nejaký MySQL

konfigurácia

mapuj port host<-kontajner

(voliteľne) sprístupni súbor init.sql do magického priečinka */docker-entrypoint-initdb.d* v kontajneri. MySQL ho načíta pri prvom spustení.

Spustíme príkazom docker compose up

alebo docker compose up –f docker-compose.yaml

Otázky?