Menné konvencie

- zatiaľ máme
  - IBookService
  - BookService
  - FileBookService
- v niektorých jazykoch je konvencia ,,názvy interfejsov majú prefix "I" (IBookService)
- jasne vidím, že je to interfejs
- zabraňuje mi to vytváraniu inštancií
- v dokumentácii je to prehľadné, ale:

Refactor time!

- zbavme sa tejto konvencie
  - abstraktné triedy tiež nemajú inštancie, budeme ich značiť špeciálne?
  - koho zaujíma, že je to interfejs, dôležité je, že to má metódy?
  - prehistorický zvyk z COM
- Eclipse: menu Refactor | Rename
- IBookService -> BookService
- BookService -> SimpleBookService

Unit testy

- každá aplikácia by mala byť otestovaná
- testy zaručujú, že tam nie sú chyby
- testuje sa viacero aspektov systému
  - korektnosť kódu, funkčnosť, výkonnosť...
  - ostatné požiadavky (bezpečnosť, použiteľnosť)...
- testovanie "zdola nahor" – overenie korektnosti a funkčnosti najmenších možných súčastí programu (jednotiek, units)
- v OOP: unit = metóda

Unit testy

- základná filozofia: treba povedať, čo exaktne znamená, že test zbehol
  - exaktne povedať = naprogramovať
- v Jave: viacero bežne používaných knižníc na testovanie
- JUnit (JUnit.org) – klasik žánru
- Eclipse v sebe obsahuje priamu podporu

Unit testy

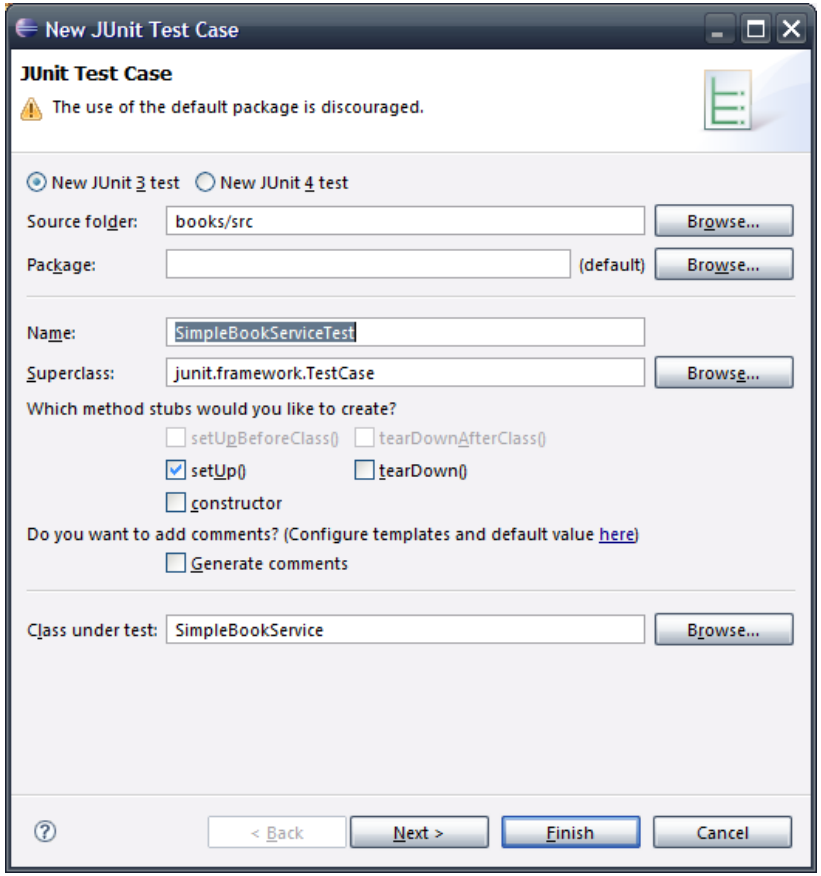
- v praktickom prípade by mali byť otestované všetky metódy všetkých tried
- **code coverage** – miera pokrytia kódu testami
- v našom príklade:
  - Book: asi nemá zmysel testovať, gettre a settre sú funkčné
  - SimpleBookService, FileBookService: test metód **list()** a **add()**

Unit testy – metóda add()

- zvyčajný postup
  - vytvoríme novú knihu
  - pridáme ju cez ***add()***
  - vytiahneme knihu metódou **getByISBN()**
  - **ak sa dve inštancie rovnajú, test zbehol**
- keďže metódu **getByISBN()** nemáme:
  - vytvoríme novú knihu
  - pridáme ju cez **add()**
  - vrátime zoznam kníh cez **list()**
  - **ak sa dve inštancie rovnajú, test zbehol**

Unit testy – v Eclipse

- Eclipse má priamu podporu
- pravý klik na triedu, ktorú ideme testovať
- New | JUnit TestCase
- názov testovacej triedy odvodený od názvu triedy **SimpleBookServiceTest**
- odporúčanie: triedy testov patria do samostatného adresára:
- **Source Folder = books/test**



Príklad jednoduchej triedy test

```
public class SimpleBookServiceTest extends TestCase {
  protected void setUp() throws Exception {
    super.setUp();
  }

  public void testAddAndList() throws Exception {
     //…
  }
}
```

- dedíme od TestCase
- metóda setUp() – inicializácie pred spustením testu
  - vytváranie inštancií, otváranie pripojení...
- metóda testAddAndList() – príklad testujúcej metódy

Testujúce metódy

- testujú korektnosť metódy v triede
- menná konvencia (bez ktorej to nebude fungovať):

  test + názov testovanej metódy

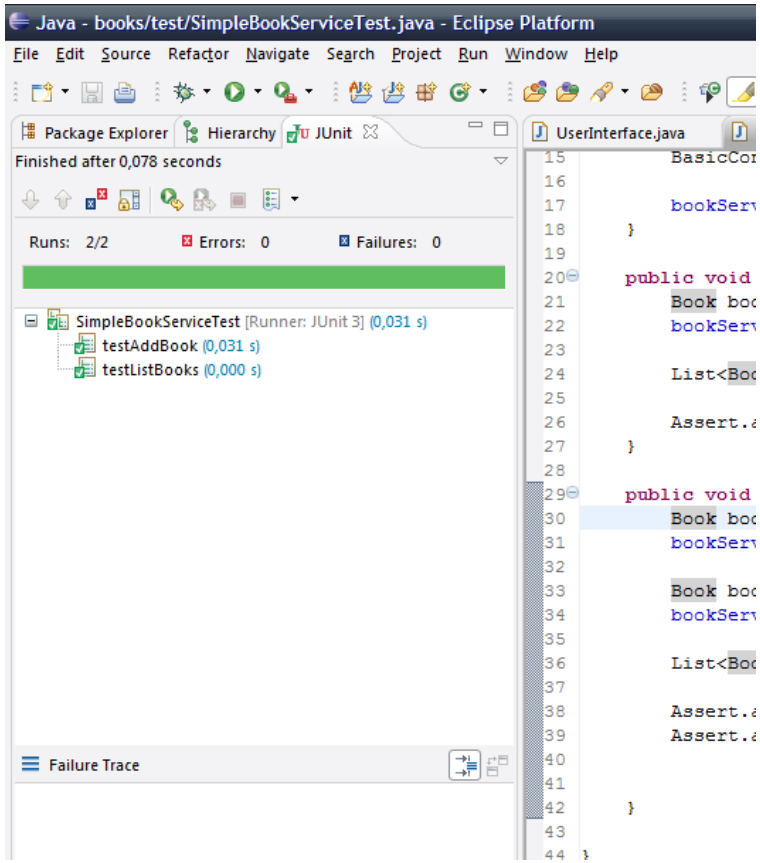
  listBooks() -> testListBooks()
- na konci testujúcej metódy musíme explicitne povedať, kedy je metóda korektná
- to robíme booleovskou podmienkou
- v každom unit teste je k dispozácii sada metód **assertXXX()**
- **assertTrue(books.contains(insertedBook))**
- assertion = vyhlásenie
- *"vyhlasujem, že metóda je korektná, ak zoznam books obsahuje inštanciu"*

Príklad unit tesetu

```
public class SimpleBookServiceTest extends TestCase {
  private SimpleBookService bookService;

  protected void setUp() throws Exception {
    super.setUp();
    bookService = new SimpleBookService();
  }

  public void testAddBookAndList() throws Exception {
    Book book = new Book("1", "J. K. Rowling", "Harry Potter");
    bookService.add(book);

    List<Book> books = bookService.listBooks();

    assertTrue(books.contains(book));
  }
}
```

Testujúce metódy

- v Eclipse spustíme:
  - pravý klik na unit test
  - Run | Run As | **JUnit Test**
- otvorí sa nový view JUnit
- zelený pás = testy zbehli
- červený = niekde je chyba



Kritika unit testov

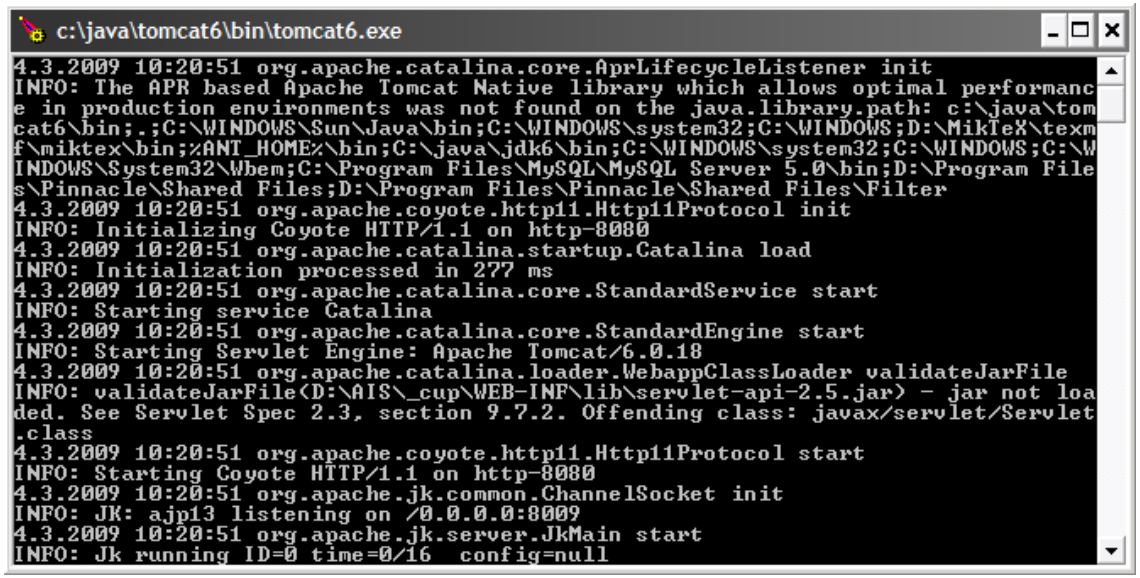
- unit testy sú v praxi relatívnou novinkou
- spojené s agilnými/extrémnymi metodikami
- základná kritika:
  - ale veď musím písať dvakrát toľko kódu!
  - testovací kód zdržiava, však za ten čas už môžem napísať oveľa viac základného kódu

Kritika unit testov

- unit testy sa oplácajú dlhodobo
  - automatizujú testovanie korektnosti
  - môžem refaktorovať!
- bez unit testov však nemôžem refaktorovať
- **refactor**: zmena vnútornej štruktúry bez zmeny vonkajšieho správania
- refactorom viem zlepšovať kód či upratovať
- unit testy mi zaručujú, že vonkajšie správanie sa mojimi úpravami nezmení
- bez nich je to "zmeň a modli sa, že to pôjde"

Logging – hlásenia o stave programu

- množstvo systémov podáva hlásenia o svojom stave
- štandardný príklad: System.out.println()
- lenže toto je v reálnych aplikáciách biedne riešenie
- namiesto neho logujeme
- log = denník



System.out.println vs. logovanie

sysout	logovanie
výstup len na konzolu	výstup do rôznych kanálov: súbor, databáza, SMS...
všetky hlásenia sú si rovné = nemožno ich filtrovať	hlásenie môže mať prioritu, podľa ktorej možno filtrovať. Zákazník vidí tie najdôležitejšie, vývojár všetko
hlásenia možno buď masovo vypnúť alebo masovo zapnúť	hlásenia možno vypínať nezávisle pre jednotlivé moduly
hlásenia majú taký formát, v akom ich uviedol vývojár	formát hlásenia možno nastavovať: pridávať čas, aktívne vlákno, triedu, riadok...

Logovanie v Jave

- dávno pradávno (pred 2002) logovanie v Jave nebolo
- niekedy v 1999 vznikol **log4j**
- podporuje všetko z predošlého slajdu a je *de facto* štandardom
- v Sune však neváhali a od JDK 1.4 je v Jave priamo k dispozícii **java.util.logging**.
- ale tá je považovaná za omyl prírody
- **„java.util.logging vzniklo z log4j cez kopirák. Lenže každá kopiráková verzia má horšiu kvalitu."**
  - z fóra TheServerSide.com
- budeme používať log4j

Logovanie v Jave

- základnou triedou je trieda org.apache.log4j.Logger
  - pozor pri importe v Eclipse, aby sme neimportli java.util.logging!
- päť základných metód na logovanie podľa priority
  - debug() – ladiace hlášky, ktoré zaujímajú len vývojára
  - info() – informačné hlášky ("Server spustený")
  - warn() – varovania ("Podpora je vypnutá pre neprítomnosť modulu Y")
  - error() – chyby ("Modul Y bol vypnutý. Systém pobeží s chybami.")
  - fatal() – ohrozujú beh aplikácie ("Nenašiel sa procesor. Vypínam.")
- namiesto System.out.println("Rátam")
- píšeme **logger.debug("Rátam");**

Získanie inštancie loggera

```
public class SimpleBookService implements BookService {
  private final static Logger log = Logger.getLogger(SimpleBookService.class);

  public List<Book> listBooks() {
    log.info("Vracia sa zoznam kníh...");
    return books;
  }
}
```

- logger nezískavame cez **new Logger()**, ale statickou metódou
- logger je **static**, pretože všetkým inštanciám SimpleBookService stačí jeden spoločný logger
- parameter getLoggera() ustanoví logger identifikovaný menom triedy (pomenovanie loggerov umožňuje ich vypínanie, filtre...)

Spusťme unit test

- po spustení unit testu však dostaneme divnú hlášku

```
log4j:WARN No appenders could be found for logger (SimpleBookService).
log4j:WARN Please initialize the log4j system properly.
```

- log4j je nutné nainicializovať
- existuje viacero spôsobov
- najjednoduchší: do setUp() v unit teste dodáme

BasicConfigurator.configure()

- nastaví štandardy: výstup na konzolu, štandardný formát

```
0 [main] INFO SimpleBookService  - Pridala sa knihu do zoznamu.
0 [main] INFO SimpleBookService  - Vracia sa zoznam kníh...
```

Nastavenie v iných prípadoch

- BasicConfigurator.configure() dodáme do metódy main()

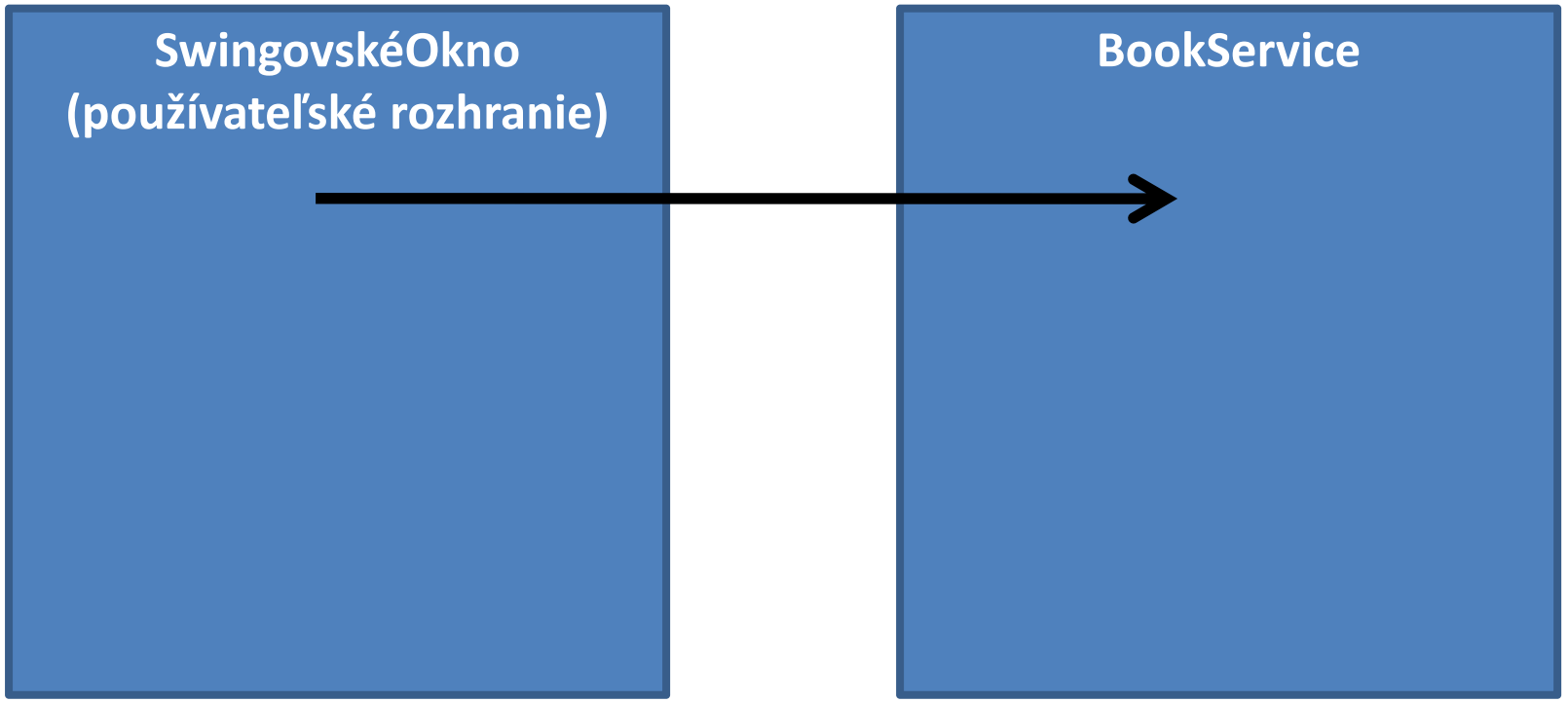
```
0 [main] INFO SimpleBookService  - Pridala sa knihu do zoznamu.
0 [main] INFO SimpleBookService  - Vracia sa zoznam kníh...
```

- 0 – čas od spustenia aplikácie v milisek.
- [main] – hlavné vlákno
- INFO – priorita
- SimpleBookService – trieda, v ktorej hlásenie vzniklo

Vrstvové aplikácie

- vrstvové aplikácie
  - vrstva dát (databáza)
  - vrstva biznis logiky
  - klientská vrstva (používateľské rozhranie)
- v našej aplikácii máme ledva 1 vrstvu
- urobme dvojvrstvovú aplikáciu:
  - dáta + biznis logika = služby
  - klientská vrstva = UI

Vrstvové aplikácie



Vrstvové aplikácie

- používateľské rozhranie bude zasielať dáta službe BookService
- zvyčajne: vyzbiera dáta od používateľa, vytvorí inštanciu knihy, odošle ju službe
- používateľské rozhranie = trieda **UserInterface**
- **UserInterface** závisí na **BookService**
- **UserInterface** potrebuje inštanciu **BookService**

UserInterface potrebuje BookService

- používateľské rozhranie bude zasielať dáta službe BookService
- zvyčajne: vyzbiera dáta od používateľa, vytvorí inštanciu knihy, odošle ju službe
- používateľské rozhranie = trieda **UserInterface**
- **UserInterface** závisí na **BookService**
- **UserInterface** potrebuje inštanciu **BookService**
- **BookService** bude inštančnou premennou v **UserInterface**

UserInterface potrebuje BookService

```
public class UserInterface {
  private BookService bookService;
}
```

- možnosť 1: UserInterface si inštanciu získa sama

```
public class UserInterface {
  private BookService bookService = new SimpleBookService();
}
```

- lenže máme problém: trieda má napevno danú implementáciu BookService
- ak chceme migrovať na FileBookService, musíme vykonať zmenu vo všetkých takýchto triedach

UserInterface potrebuje BookService

- možnosť 2: aby sme zaručili flexibilitu, trieda si nebude vytvárať inštanciu sama, ale dostane ju v konštruktore

```
public class UserInterface {
  private BookService bookService;
  public UserInterface(BookService bookService) {
    this.bookService = bookService;
  }
}
```

```
main() {
  BookService bookService = new SimpleBookService();
  UserInterface ui = new UserInterface(bookService);
}
```

UserInterface potrebuje BookService

```
main() {
  BookService bookService = new SimpleBookService();
  UserInterface ui = new UserInterface(bookService);
}
```

- predtým: trieda si svoje závislosti vyhľadala sama
- teraz: triede nastaví závislosti niekto iný
- hollywoodsky princíp: **nevolajte nám, zavoláme vás**
- triede sme nastavili závislosti v main() metóde
- otočenie filozofie sa volá "inversion of control" (prevrátená kontrola), IoC
- lepší pojem: **dependency injection** (injekcia závislostí)

Dependency Injection

- *dependency injection* umožňuje elegantne zamienať implementácie za behu
- ak chcem zmigrovať: jednoducho vložím do konštruktora triedy inú závislosť
  - tento prístup sa volá **constructor injection**
- všimnime si, ako je dôležitá zásada ***program to interfaces!***
- závislosti sme vyskladali a poprepájali v metóde main()

Dependency Injection

- kým je závislostí málo, je to v poriadku
- v prípade zložitých závislostí treba niečo inteligentnejšie: **IoC / DI kontajnery**
- príklad: **Spring Framework**
- závislosti sú definované v XML
- takto môžeme zmeniť správanie aplikácie zmenou XML súboru a aplikáciu nemusíme vôbec prekompilovávať!
- podrobnosti nabudúce