МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

ІМЕНІ ІГОРЯ СІКОРСЬКОГО"

Компоненти програмної інженерії

Архітектура програмного забезпечення

**КОМП’ЮТЕРНИЙ ПРАКТИКУМ**

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського

як навчальний посібник для студентів,

які навчаються за спеціальністю 121 «Інженерія програмного забезпечення»,

освітньо-професійної програми «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»

Київ

КПІ ім. Ігоря Сікорського

2020

Компоненти програмної інженерії. Архітектура програмного забезпечення. Комп’ютерний практикум [Електронний ресурс] : навч. посіб. для студентів спеціальності 121 «Інженерія програмного забезпечення», освітньо-професійної програми «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій» / КПІ ім. Ігоря Сікорського ; уклад.: Д. С. Смаковський, В. А. Тихоход – Електронні текстові дані (1 файл: 4,19 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 138 с.

https://osvita.kpi.ua/files/boxes/metod\_mater\_rukopis.html

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № \_\_ від \_\_.\_\_.2020 р.)

за поданням Вченої ради Теплоенергетичного факультету (протокол № \_\_ від \_\_.05.2020 р.)

Е л е к т р о н н е м е р е ж н е н а в ч а л ь н е в и д а н н я

Компоненти програмної інженерії

Архітектура програмного забезпечення

Комп’ютерний практикум

Укладачі: Смаковський Денис Сергійович, канд. техн. наук,

Тихоход Володимир Олександрович, канд. техн. наук,

Відповідальний редактор: Коваль О.В., канд. техн. наук, доцент, в.о. зав. кафедри

автоматизації проектування енергетичних процесів і систем

ТЕФ КПІ ім. Ігоря Сікорського

Рецензент:

Посібник розроблений на підставі робочої програми кредитного модуля «Компоненти програмної інженерії» та призначений для якісної організації виконання практичних робіт студентами обсягом 36 годин, підвищення розуміння основ архітектури програмного забезпечення та особливостям розробки застосунків.

Призначений для студентів денної форми, які навчаються за спеціальністю 121 – «Інженерія програмного забезпечення», освітньо-професійної програми підготовки бакалаврів «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій».

Спрямований на формування у студентів умінь та навичок з основ розробки програмного забезпечення на основі тестування, використання патернів проектування та фреймворків для впровадження залежностей та розробки веб-застосунків

© КПІ ім. Ігоря Сікорського, 2020

**ВСТУП**

Кредитний модуль «Компоненти програмної інженерії. Архітектура програмного забезпечення*»* входить до циклу базової підготовки навчального плану бакалаврів спеціальності 121 «Інженерія програмного забезпечення» освітньої програми «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій». Кредитний модуль викладається студентам третього курсу у шостому семестрі.

Знання та досвід, отримані у результаті вивчення даної дисципліни, дозволяють студентам здійснювати розробку програмного забезпечення на основі тестування, використання патернів проектування та фреймворків для впровадження залежностей та розробки застосунків, зокрема веб-застосунків.

Розглянуто: використання основ рефлексії для розробки програмного забезпечення, використання фреймворків для тестування програмного забезпечення зокрема таких, які підтримують підміну залежностей, використання патернів для розробки програмного забезпечення, використання фреймворків для ін’єкцій залежності.

Комп’ютерний практикум містить перелік завдань, в результаті виконання яких студенти здійснюють розробку програмного забезпечення.

Теоретичною основою щодо виконання комп’ютерного практикуму є курс лекцій з модулю „Компоненти програмної інженерії. Архітектура програмного забезпечення”. Комп’ютерний практикум є обов’язковим до виконання згідно робочої програми дисципліни.

Для виконання комп’ютерного практикуму необхідно мати комп’ютер з обсягом оперативної пам’яті не менше 4 Гб з операційною системою.

**ЗМІСТ**

|  | Стр. |
| --- | --- |
| Вступ ……………………………………………………………………… | 4 |
| КОМП’ЮТЕРНИЙ ПРАКТИКУМ 1 Основи роботи з Reflection та проксування викликів до об’єкту. …………………………………… | 5 |
| КОМП’ЮТЕРНИЙ ПРАКТИКУМ 2 Розробка компонентів програм придатних для модульного тестування та основи розробки за допомогою тестування.………………………………………………. | 18 |
| КОМП’ЮТЕРНИЙ ПРАКТИКУМ 3 Основи патерну проектування Model-View-Controller та тестування через підміну залежностей.……. | 27 |
| КОМП’ЮТЕРНИЙ ПРАКТИКУМ 4 Використання Gof-патернів у додатках з MVC архітектурою.…………………………………………. | 38 |
| КОМП’ЮТЕРНИЙ ПРАКТИКУМ 5 Проектування архітектури баз даних із застосуванням засобів моделювання та реверсного інжинірингу.……………………………………………………………… | 52 |
| КОМП’ЮТЕРНИЙ ПРАКТИКУМ 6 Використання ін’єкції залежності за допомогою фреймворку Spring у додатках з MVC архітектурою.…... | 64 |
| КОМП’ЮТЕРНИЙ ПРАКТИКУМ 7 Використання фреймворку Spring MVC для розробки Веб-застосунків…………………………………… | 81 |
| Список літератури | 138 |

**ЛАБОРАТОРНА РОБОТА 1**

**Основи роботи з Reflection та проксування викликів до об’єкту.**

**Мета роботи:**

- набуття навичок роботи з Reflection API, а саме: отримання інформації про класси під час виконання роботи, створення об’єктів, .

***Теоретичні відомості***

Мови програмування, компілятори яких генерують машинний код (С++, Object Pascal, Assembler), приховують практично всю інформацію про вихідний код програми. Імена змінних, а найчастіше і функцій, ховаються за адресами в пам'яті. Трохи окремо стоять динамічно завантажувані бібліотеки - в них можна отримати адресу функції під час її виконання. Ця особливість дозволяє підключати до однієї програми - зовнішні модулі прямо під час виконання програми. У мовах, де відбувається інтерпретація програми, або компіляція в проміжний код, інформація про саму програму (назви класів, їх методів, полів тощо) доступна всередині неї прямо під час виконання.

Крім підключення модулів користувача в різні фреймворки, вирішується ще цілий ряд завдань: відображення структури класів в IDE, питання серіалізації, RMI, та практично всі фреймворки.

Основа Reflection в в Java - це два класи ***Class*** і ***ClassLoader***, розташованих в пакеті ***java.lang***, і спеціальний пакет ***java.lang.reflect***, що містить допоміжні класи: ***Array, Member, Constructor, Field, Method, Modifier, InvocationHandler, Proxy, ReflectAccess, ReflectPermission, InvocationTargetException, UndeclaredThrowableExceptioCfn*** і ін.

Найпростіше освоювати техніку рефлексії, починаючи з класу ***Class***. Більш складні знаходяться в пакеті ***java.lang.reflect***, перш за все класи, що описують конструктори, поля і методи: ***Constructor, Field і Method***.

Отримати екземпляр ***Class*** можна у 2 способи:

**A.** Просто додаємо до імені класу суфікс «.class», наприклад:

*Class clazz = byte.class*

(«сlazz» - свідомо спотворене від «class»: компілятор не дозволяє використовувати в якості ідентифікатора зарезервоване слово «class».)

**B.** Якщо ми маємо в своєму розпорядженні екземпляром деякого класу, може бути навіть невідомого в даній точці програми, можна викликати метод ***getClass()***, присутній в кожному Java-об'єкт (успадкований від класу ***Object***):

*void myFunction (Object o) {*

*Class clazz = o.getClass ();*

*// щось робимо з об'єктом clazz;*

*}*

Примітивні типи Java - ***boolean, char, byte, short, int, long, float, double*** - звичайно не вважаються повноцінними класами. Вони не успадковані від ***Object***, для них не працює успадкування і т.д. Проте, з ними теж асоційовані екземпляри ***Class***, які можна отримати способом A.

Класс ***Class*** підтримує наступні операції:

- метод ***getName()*** повертає повне ім'я класу, наприклад:

***String.class.getName()*** повертає «java.lang.String»;

***float.class.getName()*** повертає «float»;

***float []. class.getName()*** повертає «[F»;

***float [] []. class.getName()*** повертає «[[F»;

***String []. Class.getName()*** повертає «[Ljava.lang.String;»;

- можна перевірити, чи не є клас інтерфейсом, масивом або примітивним типом: за допомогою методів ***isInterface(), isArray()*** і ***isPrimitive()***.

Якщо клас - масив, можна отримати тип його елементів (тобто об'єкт ***Class***): метод **getComponentType***()*. Для не-масивів цей метод поверне null.

Метод ***getModifiers****()* для об'єкта типу ***Class*** повертає значення типу ***int***, біти якого представляють модифікатори класу. Після цього можна використовувати статичні методи класу ***java.lang.reflect.Modifier***, щоб визначити, які модифікатори були застосовані до класу:

*Class c = obj.getClass ();*

*int mods = c.getModifiers ();*

*if (Modifier.isPublic (mods)) {System.out.println ( "public"); }*

*if (Modifier.isAbstract (mods)) {System.out.println ( "abstract"); }*

*if (Modifier.isFinal (mods)) {System.out.println ( "final"); }*

Метод ***getSuperclass****()* повертає об'єкт типу ***Class***, який представляє суперклас поточного класу. Потрібно не забувати враховувати, що в Java відсутня множинне успадкування і клас ***java.lang.Object*** є базовим класом для всіх класів, внаслідок чого якщо у класу немає батька, то для нього метод ***getSuperclass()*** поверне ***null***. Для того, щоб отримати всі батьківські суперкласи, потрібно рекурсивно викликати метод ***getSuperclass()***.

За допомогою рефлексії можна також визначити, які інтерфейси реалізовані в заданому класі. Метод ***getInterfaces()*** поверне масив об'єктів типу ***Class***. Кожен об'єкт в масиві представляє один інтерфейс, реалізований в заданому класі.

*Class c = myObj.getClass ();*

*Class superclass = c.getSuperclass ();*

*Class clazz = LinkedList.class;*

*Class [] interfaces = clazz.getInterfaces ();*

*for (Class cInterface: interfaces) {*

*System.out.println (cInterface.getName ());*

*}*

Метод ***getFields*** () повертає масив об'єктів типу java.lang.reflect.Field, які відповідають усім відкритим полях об'єкта. Ці відкриті поля необов'язково повинні міститися безпосередньо всередині класу, з яким ви працюєте, вони також можуть містяться в суперкласі або інтерфейсі. За допомогою класу ***Field*** можна отримати ім'я поля, тип і модифікатори. Якщо відоме ім'я поля, то можна отримати про нього інформацію за допомогою методу ***getField****()*.

*Class c = obj.getClass();*

*Field [] publicFields = c.getFields();*

*for (Field field: publicFields) {*

*Class fieldType = field.getType();*

*System.out.println ( "Ім'я:" + field.getName() + "Тип:" + fieldType.getName ());*

*}*

*Field nameField = c.getField ( "name");*

Методи***getField****() і* ***getFields****()* повертають тільки відкриті члени даних класу і його предків. Якщо потрібно отримати всі поля деякого класу потрібно використовувати методи***getDeclaredField****() і* ***getDeclaredFields****().* Ці методи працюють точно також як їх аналоги ***getField()*** і ***getFields()***, за винятком того, що вони повертають все поля, включаючи закриті і захищені, але тільки поточного класу.

За допомогою методів ***forName***() і ***newInstance***() об'єкта Class можна динамічно завантажувати і створювати екземпляри класу в разі, коли ім'я класу невідомо до моменту виконання програми. Метод ***newInstance()*** повертає об'етов узагальненого типу ***Object***, тому його приводять до того типу, який потрібен.

*Class c = Class.forName ("Test");*

*Object obj = c.newInstance ();*

*Test test = (Test) obj;*

Якщо необхідно викликати конструктор з параметрами, то викликають метод ***newInstance()*** конкретного знайденого конструктора:

*class TestDecl extends ReflectionTest {*

*int testField;*

*public TestDecl (int testField) {this.testField = testField; }*

*}*

*Class clazz = TestDecl.class;*

*Constructor constr = clazz.getConstructor(new Class[]{int.class});*

*Object obj = constr.newInstance( 100500 );*

Метод ***getMethods()*** об'єкта Class повертає масив об'єктів типу ***java.lang.reflect.Method***. Потім за допомогою об'єкта класу ***Method*** можна отримати ім'я методу, тип значення, що повертається, типи параметрів, модифікатори і типи можливих виключень.

За допомогою метода ***getMethod()*** можна отримати інформацію по окремому об’єкту класу ***java.lang.reflect.Method****,* якщо відомі ім'я методу і типи параметрів:

*Class [] paramTypes = new Class [] {int.class, String.class};*

*Method m = clazz.getMethod ("methodA", paramTypes);*

Для того, щоб отримати всі методи класу, незалежно від типу доступу, потрібно скористатися методами ***getDeclaredMethod()*** і ***getDeclaredMethods()***, які працюють так само, як і їх аналоги (***getMethod()*** і ***getMethods()***).

За допомогою об’єкту класу ***Method*** можна динамічно обирати і викликати метод. Приклад виклику методу:

*Method toString = clazz.getMethod("toString");*

*String result = (String) toString.invoke(someObject);*

*System.out.println(result);*

Розглянемо проксування об’єктів. Проксі (від англ. proxy — «представник, уповноважений») - спеціальний об’єкт, який динамічно створюється для “огортання” викликів методів на деякому цільовому об’єкті. Цей об’єкт створюється на базі проксі-класу, який динамічно реалізує інтерфейси вихідного об’єкту. Реалізація проксі відбувається за допомогою інтерфейса ***InvocationHandler***, метод ***invoke()*** приймає всі виклики до проксованого об’єкту.

Таким чином, для створення проксі необхідно:

* створити інтерфейс(и), виклики методів яких будуть перехоплюватись;
* створити клас, який реалізує інтерфейс ***InvocationHandler*** та містить посилання на об’єкт класу, що проксується, перевизначивши метод ***invoke()***, у якому можна виконувати додатковий код та викликати метод проксованого класу;
* створити проксі-об’єкту за допомогою класу ***java.lang.reflect.Proxy*** та його метод ***newProxyInstance(),*** який приймає ***ClassLoader***, список інтерфейсів класу та посилання на об’єкт, який реалізує інтерфейс ***InvocationHandler.***

Нижче наведено приклад створення проксі.

*interface SomeInterface{*

*int getField();*

*void setField(int val);*

*}*

*class SomeEntity implements SomeInterface{*

*int field;*

*@Override*

*public int getField() {*

*return field;*

*}*

*public void setField(int field) {*

*this.field = field;*

*}*

*}*

*class SomeProxy implements InvocationHandler{*

*private Object ref;*

*private SomeProxy(Object ref) {*

*this.ref = ref;*

*}*

*@Override*

*public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {*

*if( method.getName().startsWith("get") ){*

*return method.invoke(ref, args);*

*}*

*throw new IllegalAccessException("Not allowed");*

*}*

*public static Object newProxyInstance(Object obj) {*

*return java.lang.reflect.Proxy.newProxyInstance(*

*obj.getClass().getClassLoader(),*

*obj.getClass().getInterfaces(),*

*new SomeProxy(obj));*

*}*

*}*

*public class ProxySample {*

*public static void main(String[] args) throws Exception {*

*SomeEntity entity = new SomeEntity();*

*SomeInterface proxy = (SomeInterface) SomeProxy*

*.newProxyInstance(entity);*

*System.out.println(proxy.getField());*

*proxy.setField(5);*

*}*

*}*

В даному прикладі *SomeInterface* - інтерфейс з методами, що будуть перехоплюватися; *SomeEntity -* клас, який буде проксуватися; *SomeProxy -* клас для створення проксі.

**Завдання:**

1. Створити базовий і похідний класи згідно до варіанту індивідуального завдання. Визначити конструктори, методи доступу, віртуальні методи (хоча б *toString*()). Показати використання конструкторів похідного і базового класу. Можна визначити деякі константи або enum.

2. Розробити анотацію, позначити, нею метод(и) в класі, За допомогою рефлексії обійти методи класу і викликати відмічені анотацією методи за допомогою *invoke*().

3. За допомогою рефлексії вивести ім'я класу. Також здійснити використання рефлексії відповідно до варіанта індивідуального завдання з Таблиці 1.1.

4. Зробити для класу проксі (для незмінності), який пропускає звернення до getter-ів, там, де setter-и - породжує виключну ситуацію (*Exception*).

**Варіанти індивідуальних завдань**

**Класи для індивідуальних завдань**

**Варіант 1** Базовий клас - **дата** (день, місяць рік). Похідний клас - **записна книжка** (ПІБ, телефон, дата народження, метод обчислення кількості днів до дня народження);

**Варіант 2** Базовий клас - **час** (години, хвилини, секунди), похідний клас - **розклад** (дисципліну, аудиторію, час початку заняття і метод обчислення часу до початку заняття);

**Варіант 3** Базовий клас - **вікно** включає координати (лівий верхній та нижній правий кути вікна, колір вікна), похідний клас - **вікно з текстом** (текст, колір тексту у вікні);

**Варіант 4** Базовий клас - **поліном** (масив коефіцієнтів, метод обчислення значення поліному), похідний клас - **раціональне вираз** (поліном в чисельнику, поліном в знаменнику).

**Варіант 5** Базовий клас - **квадратна** **матриця** (значення коефіцієнтів, метод обчислення визначника). Похідний клас - **систему лінійних алгебраїчних рівнянь** (матриця, стовпець вільних членів, метод розв’язання).

**Варіант 6** Базовий клас - **дріб** (чисельник та знаменник), похідний клас - **дробове комплексне число** (включає дробову дійсну частину, дробову уявну частину, арифметичні операції (+, -, \*) над комплексними дробом);

**Варіант 7** Базовий клас - **комплексне число** (дійсна та уявна частина) і похідний клас - **комплексне число** **в експоненційній формі** (метод обчислення експоненціальної форми числа).

**Варіант 8** Базовий клас - **студент** (ім'я, курс і ідентифікаційний номер, метод друку). Похідний клас - **студент-дипломник** (тема диплома).

**Варіант 9** Базовий клас - **тварина** (вид тварини, число кінцівок, число нащадків). Похідний клас - **домашня тварина** (ім’я).

**Варіант 10** Базовий клас - **машина** (марка, число циліндрів, потужність). Похідний клас - **вантажівка** (вантажопідйомність).

**Варіант 11** Створіть клас **точка** (задається координатами x, y). Створити похідний клас - **еліпс** (2 піввісі). Напишіть віртуальний метод переміщення.

**Варіант 12** Базовий клас - **комп'ютер** (назва, частота, процесор, кількість ядер, обсяг ОЗУ, постійної пам'яті), похідний клас - **ноутбук** (діагональ екрану, вага, об'єм батареї).

**Варіант 13** Базовий клас - **мобільний телефон** (назва, вага, діагональ екрану, кількість вбудованої пам'яті, наявність камери). Похідний клас - **смартфон** (частота процесора, кількість ядер, об'єм ОП).

**Варіант 14** Створіть клас **точка**, яка має координати. Створити похідний клас - **прямокутник** (висота, ширина). Напишіть віртуальний метод переміщення.

**Варіант 15** Базовий клас - **комп'ютер** (назва, частота процесор, кількість ядер, обсяг ОЗУ, постійної пам'яті), похідний клас - **планшет** (назва ОС, діагональ екрану, обсяг батареї).

**Варіант 16** Базовий клас - **дата** (рік, місяць, день). Похідний клас - **дата з часом** (години, хвилини).

**Варіант 17** Створіть клас **точка**, яка має координати. Створити похідний клас - **правильний багатокутник** (кількість вершин, сторона). Напишіть віртуальний метод розрахунку площі.

**Варіант 18** Створити базовий клас **стек** (на базі масиву цілих чисел, з максимальним розміром і методами додавання і вилучення елемента) з методами додати і витягти елемент. Створіть клас **чергу** - на базі того ж масиву.

**Варіант 19** Базовий клас - **трикутник** (3 сторони, метод розрахунку площі). Похідний клас - **чотирикутник** (+1 сторона і діагональ).

**Варіант 20** Базовий клас - **книга** (назва, автор(и), рік видання, тираж, кількість сторінок). Похідний клас - **книга в бібліотеці** (інвентарний номер, хто взяв).

**Варіант 21** Базовий клас - **дріб** (чисельник, знаменник, арифметичні операції, перетворення в дійсний тип). Похідний клас - **ціле число** з дробової частиною.

**Варіант 22** Базовий клас - **предмет посуду** (назва, колір). Похідний клас - **тарілка** (діаметр та ємність).

**Варіант 23** Базовий клас - **будівля** (площа, адреса). Похідний клас - **житловий будинок** (кількість поверхів та квартир).

**Варіант 24** Базовий клас - **баржа** (водотоннажність, осадка, довжина, ширина). Похідний клас - **корабель** (потужність двигуна, кількість членів екіпажу).

**Варіант 25** Базовий клас - **меблі** (вид, колір, модель). Похідний клас - **м’які меблі** (назва тканини).

**Варіант 26** Базовий клас - **електроприлад** (модель, потужність). Похідний клас - кухонний електроприлад (вид прилада, виробник).

**Варіант 27** Базовий клас - **інструмент** (назва, вага). Похідний клас - **електроінструмент** (потужність, виробник).

**Варіант 28** Базовий клас - **одяг** (вид одягу, розмір, матеріал, колір). Похідний клас - **брендовий одяг** (назва бренду).

**Варіант 29** Базовий клас - **взуття** (розмір, колір). Похідний клас - **спортивне взуття** (вид спорту, назва бренду).

**Варіант 30** Базовий клас - **солодощі** (назва, калорійність). Похідний клас - **цукерки** (вид цукерок, виробник).

**Завдання з рефлексії**

1. Список конструкторів з параметрами.
2. Список методів з аннотаціями та типами параметрів.
3. Список методів зі специфікатором доступу та типами параметрів.
4. Список полів з іменами, типами таи модифікаторами доступу.
5. Список інтерфейсів, реалізованих класом
6. Модифікатори класу
7. Назву пакета та коротку (просту) назву класу
8. Назву суперкласу
9. Список анотацій классу
10. Список полів з типами та анотаціями

Таблиця 1.1 Індивідуальні завдання

| № варіанту | Завдання 1 | Завдання 2 | № варіанту | Завдання 1 | Завдання 2 | № варіанту | Завдання 1 | Завдання 2 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 6 | 11 | 1 | 8 | 21 | 1 | 10 |
| 2 | 2 | 7 | 12 | 2 | 9 | 22 | 2 | 6 |
| 3 | 3 | 8 | 13 | 3 | 10 | 23 | 3 | 7 |
| 4 | 4 | 9 | 14 | 4 | 6 | 24 | 4 | 8 |
| 5 | 5 | 10 | 15 | 5 | 7 | 25 | 5 | 9 |
| 6 | 1 | 7 | 16 | 1 | 9 | 26 | 1 | 9 |
| 7 | 2 | 8 | 17 | 2 | 10 | 27 | 2 | 10 |
| 8 | 3 | 9 | 18 | 3 | 6 | 28 | 3 | 6 |
| 9 | 4 | 10 | 19 | 4 | 7 | 29 | 4 | 7 |
| 10 | 5 | 1 | 20 | 5 | 8 | 30 | 5 | 8 |

Контрольні запитання.

1. Що таке рефлексія?
2. Яку інформацію можна отримати за допомогою рефлексії?
3. Як викликати метод за допомогою рефлексії?
4. Чи можна отримати значення приватних полів?
5. Як завантажити попередньо невідомий клас під час виконання програми та створити його екземпляр?

Лабораторна робота 2

Розробка компонентів програм придатних для модульного тестування та основи розробки за допомогою тестування.

Короткі теоретичні відомості

**Мета роботи:**

- набуття навичок

Модульне тестування або unit testing - процес перевірки на коректність функціонування окремих частин вихідного коду програми шляхом запуску тестів в штучному середовищі. Під частиною коду в Java слід розуміти виконуваний компонент. За допомогою модульного тестування зазвичай тестують низькорівневі елементи коду - такі, як методи. JUnit дозволяє поза досліджуваного класу створювати тести, при виконанні яких відбудеться коректне завершення програми. Крім основного позитивного сценарію може виконуватися перевірка працездатності системи в альтернативних сценаріях, наприклад, при генерації методом виключення як реакція на помилкові вихідні дані. Оцінюючи кожну частину коду ізольовано і підтверджуючи коректність її роботи, точно встановити проблему значно простіше, ніж якби елемент був частиною системи. Технологія дозволяє і пропонує зробити більш тісний зв'язок між розробкою коду і його тестуванням, а також надає можливість перевірити коректність роботи класу, не вдаючись до трасування при налагодженні коду.

При включенні модульного тестування в проект:

- тести розробляються для нетривіальних методів системи;

- помилки виявляються в процесі проектування методу або класу;

- в першу чергу розробляються тести на основний позитивний сценарій;

- розробнику доводиться більше приділяти уваги альтернативним сценарієм поведінки, так як вони є джерелом помилок, що виявляються на пізніх стадіях розробки;

Бібліотеки модульного тестування звичайно не входять в дистрибутив мови програмування. Зокрема в Java JUnit є зовнішньою залежністю. Для керування зовнішніми залежностями та збирання найчастіше використовується інструменти Gradle або Maven. Розглянемо Apache Maven як найбільш використовуваний.

Maven складається із наступних компонентів:

- парсер командного рядка;

- менеджер Class Path;

- парсер конфігураційних файлів POM (Project Object Model);

- завантажувач зовнішніх бібліотек.

Схема роботи Maven наведена на Рис.1.

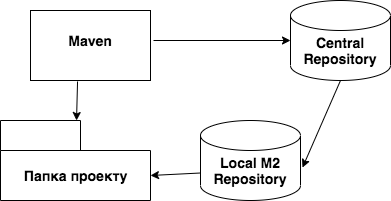


Рис 1. Робота Maven з центральним та локальним репозиторіями

Модульні тести розміщуються в спеціально призначеній папці (звичайно /src/test/java). Вони являють собою звичайні класи, методи яких позначені анотацією @Test.

Для підключення JUnit 5 найкраще створити проект Maven або Gradle.

Додати наступні залежності:

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

Версію JUnit можна вказати в розділі properties

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<junit5.version>5.3.2</junit5.version>

</properties>

Також необхідно підключити plugin

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M3</version>

</plugin>

Приклад класу та тестів.

import java.util.Optional;

public class CalculatorService {

private static CalculatorService instance = new CalculatorService();

public static CalculatorService getInstance(){

return instance;

}

public int add(int first, int second){

return first + second;

}

public int sub(int first, int second) {

return first - second;

}

public Optional<Integer> div(int first, int second){

Optional<Integer> result = Optional.empty();

if( second != 0){

result = Optional.of(first / second);

}

return result;

}

}

import org.itstep.calculator.model.CalculatorService;

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class CalculatorServiceTest {

public CalculatorService calculatorService = CalculatorService.getInstance();

@Test

@DisplayName("new test for add")

public void add() {

assertEquals(calculatorService.add(3, 5), 8);

assertNotEquals(calculatorService.add(3, 2), 3);

}

@Test

@DisplayName("new test for sub")

public void sub() {

assertEquals(calculatorService.sub(5, 3), 2);

assertNotEquals(calculatorService.sub(3, 2), 3);

}

}

Завдання до лабораторної роботи

Для варіантів лабораторної роботи №1 забезпечити покриття тестами не менше ніж 80 відсотків, як позитивного так і негативного сценаріїв.

Контрольні запитання.

1. Що таке модульне тестування?
2. Які переваги та недоліки модульного тестування?

Як порахувати відсоток покриття тестами?

1. Які обмеження модульного тестування?
2. Які очікувані результати модульного тестування?
3. Які недоліки модульного тестування?
4. Які обмеження накладаються на класи для забезпечення їх модульного тестування?

Лабораторна робота 3

Основи патерну проектування Model-View-Controller та тестування через підміну залежностей.

Короткі теоретичні відомості

Найбільш загальним в описі Model-View-Controller є те, що програма складається з трьох частин:

* Модель, яка не залежить ні від Контроллеру ні від Відображення, що дає можливість тестувати її як окремий компонент.
* Відображення, яке відповідає за представлення користувачу моделі.
* Контроллер, який завжди знає про модель та може її змінювати.

Розбиття програми на наступні частини дозволяє тестувати кожну частину окремо, для Контроллеру та Відображення підміняючи залежності від моделі. Модульний тест повинен перевірити клас в ізоляції. Побічні ефекти від інших класів або системи повинні бути усунені, якщо це можливо. Для усунення цих побічних ефектів, ви повинні замінити залежності від інших класів. Це може бути зроблено за допомогою заміни для реальних залежностей.

Mock-тести можуть підтвердити, що клас правильно реагує під час випробувань. Наприклад, ви можете перевірити, якщо були викликані деякі методи на макеті об'єкта. Це допомагає гарантувати, що перевіряється конкретний, відокремлений клас під час виконання тестів і що ваші тести не залежать від будь-яких побічних ефектів.

Фіктивні об'єкти зазвичай налаштовуються. При цьому вони, як правило, вимагають менше коду для налаштування і, отже, їх застосування повинно бути кращим. Для підміни залежності існує набір фреймворків для тестування. Ми розглянемо найбільш вживаний з них - Mockito.

Виконання тесту з Mockito включає 4 етапи:

- створити на налаштувати;

- замінити зовнішні залежності Mock-об'єктами в коді модульного теста;

- виконати код тесту;

- підтвердити, що код виконується правильно, а також тестований об'єкт правильно взаємодіє зі своїми залежностями.

Для додавання Mockito у проект необхідно включити наступну залежність в pom.xml

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>RELEASE</version>

<scope>test</scope>

</dependency>

Основним у бібліотеці є класс org.mockito.Mockito, який звичайно статично імпортується.

Для створення заглушки (або моку), використовуйте метод ***mock(Class clazz)***. Потім використовуйте *when(mock.method())*.*thenReturn(value)*, щоб вказати значення, що повертається для методу. Якщо Ви вкажете більше одного значення, що повертається, вони будуть повернуті методом послідовно, поки не повернеться останнє, після цього при наступних викликах буде повертатися тільки останнє значення.

import static org.mockito.Mockito.\*;

import static org.junit.Assert.\*;

....

@Test

public void iteratorWillReturnHelloMockito() {

Iterator i = mock(Iterator.class); // створюємо

when(i.next()).thenReturn("Hello").thenReturn("Mockito"); //налаштовуємо

String actual = i.next()+" "+i.next(); //виконуємо

assertEquals("Hello Mockito", actual); //порівнюємо

}

Для методів з параметрами для різних параметрів можна повертати різні значення. Якщо значення параметру не налаштоване, тоді повертається null або 0.

@Test

public void testWithTwoArgs(){

Function<String,Integer> mockFunction = mock(Function.class);

when(mockFunction.apply("first")).thenReturn(1);

when(mockFunction.apply("second")).thenReturn(2);

int firstActual = mockFunction.apply("first");

int secondActual = mockFunction.apply("second");

Integer thirdActual = mockFunction.apply("third");

assertEquals(1, firstActual);

assertEquals(2, secondActual);

assertEquals(null, thirdActual);

}

Підміну залежностей можна побачити у наступному прикладі.

public class ModelSample { // класс який підміняється

public int someMethod(int val){

return val + 10;

}

}

@AllArgsConstructor

public class ControllerSample { //класс залежність якого підміняється

ModelSample model;

public int callModel(){

return model.someMethod(10);

}

}

public class TestForDependencySubstitution {

@Test

public void testDependencySubstitution(){

ModelSample modelMock = mock(ModelSample.class);

when(modelMock.someMethod(10)).thenReturn(15);

ControllerSample controller = new ControllerSample(modelMock);//підміна залежності

int actual = controller.callModel();

int expected = 15;

assertEquals(expected, actual);

verify(modelMock).someMethod(10); //перевірка взаємодії із залежністю

}

}

У прикладі, наведеному вище також наведено приклад перевірки взаємодії із залежністю за допомогою методу **Mockito.verify()**.

Завдання до лабораторної роботи

Для варіантів лабораторної роботи №1 створити програму, що здійснює введення та виведення даних на консоль за допомогою патерну MVC. Забезпечити модульне тестування контроллеру, підмінивши введення даних з консолі та Модель на заглушки за допомогою Mockito.

Контрольні запитання.

1. Яка мета підміни залежності?
2. Який метод застосовується для генерації заглушки?
3. Яким чином можна налаштувати заглушку?
4. Які особливості налаштування методів із параметрів на заглушці?
5. Які підмінити залежність від реального класу на заглушку?
6. Які вимоги для класів, залежності яких можна підміняти?
7. Які перевірити виклики методів на заглушці?

Лабораторна робота 4

Використання Gof-патернів у додатках з MVC архітектурою.

Патерн проектування описує задачу, яка знову і знову виникає в роботі розробників, а також принцип її рішення, причому таким чином, що це рішення можна потім можна використовувати багаторазово, нічого не вигадуючи заново.

GoF патерни поділяються на 3 групи:

- породжуючі;

- структурні;

- поведінкові.

Будь-який паттерн складається із наступних 4 елементів.

**Назва.** Пославшись на нього, ми можемо відразу описати проблему проектування; її рішення і їх наслідки. Присвоєння паттернам імен дозволяє проектувати на більш високому рівні абстракції.

**Завдання**. Опис того, коли слід застосовувати патерн. Необхідно сформулювати завдання і її контекст. Може описуватися конкретна проблема проектування, наприклад спосіб представлення алгоритмів у вигляді об'єктів.

**Рішення.** Опис елементів дизайну, зв’язків між ними, функцій кожного елемента. Конкретний дизайн або реалізація не маються на увазі, оскільки патерн - це шаблон, який можна застосовувати в різних ситуаціях. Надається абстрактне опис завдання проектування і того, як вона може бути вирішена за допомогою якогось узагальненого поєднання елементів (в нашому випадку класів і об'єктів).

**Результати** - це наслідки застосування патерну і різного роду компроміси. Хоча при описі проектних рішень про наслідки часто не згадують, знати про них необхідно, щоб можна було вибрати між різними варіантами і оцінити переваги і недоліки даного патерну.

Породжуючі патерни проектування роблять процес створення об’єктів абстрактним. Вони допоможуть зробити систему незалежною від способу створення, композиції та представлення об'єктів. Патерн, який породжує об'єкти, делегує інстанціювання іншому об’єкту.

Для породжують патернів актуальні дві особливості. По-перше - ці патерни інкапсулюють знання про конкретні класах, які застосовуються в системі. По-друге - приховують деталі того, як ці класи створюються і взаємодіють. Єдина інформація про об'єкти, відома системі, - це їх інтерфейси, визначені за допомогою абстрактних класів (інтерфейсів). Отже, породжують патерни забезпечують більшу гнучкість при вирішенні питання про те, що створюється, хто це створює, як і коли. Можна зібрати систему з "готових" об'єктів з самої різною структурою і функціональністю статично (на етапі компіляції) або динамічно (під час виконання). Найбільш вживаними породжуючими паттернами є **Abstract Factory, Builder, Factory Method, Singleton**.

Патерни поведінки пов'язані з алгоритмами і розподілом обов'язків між об'єктами. Мова в них йде не тільки про об'єкти і класи, але і про типові способи взаємодії. Патерни поведінки характеризують складний потік управління, який важко простежити під час виконання програми. Увага акцентована не на потоці управління як такому, а на зв'язках між об'єктами.

У патернах поведінки рівня класу використовується спадкування - щоб розподілити поведінку між різними класами. Прикладом є шаблонний метод, який являє собою абстрактне визначення алгоритму.

У патернах поведінки рівня об'єктів використовується не успадкування, а композиція. Деякі з них описують, як за допомогою кооперації безлічі рівноправних об'єктів справляється із завданням, яка жодному з них не під силу. Важливо тут те, як об'єкти отримують інформацію про існування один одного. Об'єкти-колеги можуть зберігати посилання один на одного, але це збільшить ступінь зв'язаності системи. При максимальному ступені зв'язаності кожному об'єкту довелося б мати інформацію про всі інші. Цю проблему вирішує патерн посередник. Посередник, що знаходиться між об'єктами-колегами, забезпечує опосередкованість посилань, необхідну для розірвання зайвих зв'язків. Найбільш уживаними поведінковими паттернами є - **Command, Iterator, Observer, Strategy, Template Method.**

У структурних патернах розглядається питання про те, як з класів і об'єктів утворюються більш великі структури. Структурні патерни рівня класу використовують успадкування для складання композицій з інтерфейсів і реалізацій. Простий приклад - адаптер робить інтерфейс одного класу (адаптованого) сумісним з інтерфейсом іншого, забезпечуючи тим самим уніфіковану абстракцію різнорідних інтерфейсів. Замість композиції інтерфейсів або реалізацій структурні патерни рівня об'єкта компонують об'єкти для отримання нової функціональності. Додаткова гнучкість в цьому випадку пов'язана з можливістю змінити композицію об'єктів під час виконання, що неприпустимо для статичної композиції класів. Прикладом структурного патерну рівня об'єктів є компонувальник. Він описує побудову ієрархії класів для двох видів об'єктів: примітивних і складених. Останні дозволяють створювати довільно складні структури з примітивних і інших складених об'єктів. Найбільш уживаними структурними паттернами є - **Adapter, Proxy, Facade, Decorator.**

Завдання до лабораторної роботи

В попередній лабораторній за шаблоном MVC додати використання не менше 5 GoF-патернів. Програма має містити шар контроллеру, в якому можна використати паттерн Command, також можна використати паттерн Proxy для контролю доступу або логування подій. В шарі моделі можна використати клас Сервісу (Decorator), Strategy, Chain of Responsibility тощо. Для будь - якого класу, єдиний екземпляр якого існує весь час виконання програми можна використати паттерн Singleton.

Контрольні запитання

1. Які ви знаєте групи патернів?
2. Із яких 4 елементів складається патерн?
3. Яка мета породжуючих патернів?
4. Наведіть 3 приклади породжуючих патернів.
5. Яка мета структурних патернів?
6. Наведіть 3 приклади структурних патернів.
7. Яка мета поведінкових патернів?
8. Наведіть 3 приклади поведінкових патернів.

Лабораторна робота 5

Проектування архітектури баз даних із застосуванням засобів моделювання та реверсного інжинірингу.

Короткі теоретичні відомості

Звичайно засоби моделювання баз даних пропонують 2 моделі - логічну та фізичну. Фізична модель прив’язана до конкретної системи керування базами даних і може містити особливості індексів, зовнішніх ключів, обмежень цілісності тощо. Логічна модель не прив’язана до конкретної системи керування базами даних та може містити цілком інші назви стовпчиків таблиць і більше призначена для проектування або збереження схеми загалом або, наприклад, для демонстрації стейкхолдерам.

Для виконання роботи одним із рекомендованих варіантів є засіб моделювання SQL Power Architect. За допомогою нього можна виконувати як прямий, так і зворотній інжиніринг (генерацію моделі із наявної бази даних).

Загальне завдання до роботи

На основі короткого опису предметної області створити модель бази даних. Складність моделі - близько 8 таблиць, забезпечити виконання вимог 3 нормальної форми. Створити скрипт і саму базу даних, продумати, які індекси необхідні для пошуків. Завантажити невеликою кількістю тестових даних. Продумати обмеження на поля.

Варіанти індивідуальних завдань

1. **«Бібліотека»**: БД повинна підтримувати режим пошуку книги по заданому критерію (автор, назва), замовлення книги, обліку клієнтів і книг у книгосховищі, видачі звітів за запитами (місцезнаходження книги в архіві або її відсутність), видачі документів про боржників .

2. **«Магазин»**: В системі повинні підтримуватися режими замовлення товару (продовольчих товарів), покупки та обліку товарів, аналізу купівельному товару, аналізу купівельному продуктів в залежності часу дня і дня тижня.

3. **«Будинок»**: В системі повинні підтримуватися режими обліку мешканців та обліку доходів і витрат, пов'язаних з проживанням мешканців в будинку (оренда, електроенергія, кім. Послуги і т. Д.), Видачі аналітичної інформації, видачі списку мешканців.

4. **«Готель»**: В системі повинні підтримуватися режими обліку та розподілу за номерами приїжджих гостей в залежності від вимог і побажань мешканців.

5. **«Театр»**: В системі повинні підтримуватися режими обліку вистав, акторів, які грають у виставах, концертів, розподілу місць і вартості квитків, аналізу популярності вистав за різними критеріями.

6. «**Успішність студентів на факультеті**»: В системі повинні підтримуватися режими обліку учнів і результатів здачі іспитів, аналізу сесії по семестрах, по факультетах, спеціальностях, генерації звітів відмінників і двієчників.

7. «**Водійські курси**»: В системі повинні підтримуватися режими обліку учнів на курсах в залежності від категорії, відвідуваності, тренувань водіння, обліку помилок водіння, попередніх результатів тестів і результатів здачі іспитів, аналізу популярності курсів в залежності від вартості, тривалості і якості курсів .

8. «**Магазин (аудіо-відео продукції)**»: У системі повинні підтримуватися режими обліку товару в магазині за категоріями (аудіо, відео, CD або DVD диски, касети), по темам (фільми, музика за категоріями), замовлення товару та аналізу популярності продукції. В БД повинна бути дані для лояльності клієнтів.

9. «**Конкурс поетів**»: В системі повинні підтримуватися режими обліку конкурсантів, їх творчого багажу, їх опубліковані роботи, порядку проведення, сценарію виступу кожного і в цілому в залежності від теми і аналізу конкурсів по роках, обліку відвідування та участі в конкурсі.

10. «**Конференція**»: В системі повинні підтримуватися режими урахування виступаючих студентів, аспірантів і викладачів за тематиками, за спеціальностями, реєстрації учасників та гостей на конференції, обліку тривалості і новизни тим, аналізу конференції.

11. «**Фестиваль фільмів**»: В системі повинні підтримуватися режими обліку гостей, учасників і журі, фільмів і номінацій за категоріями, реєстрації учасників та проведення голосування з урахуванням глядацьких симпатій і рейтингу популярності фільмів. Оцінка за різними критеріями повинна залежати від категорії (складність, глибина і т. Д.).

12. «**Центр зайнятості**»: В системі повинні підтримуватися режими обліку безробітних, їх стажу, кваліфікації, бажання працювати за певною спеціальністю, місця розташування і заробітної плати, обліку вже стоять на обліку в центрі зайнятості, аналізу зайнятості від часу, спеціальності тощо. д.

13. «**Лікарня**»: В системі повинні підтримуватися режими обліку хворих по відділеннях, захворювань, складності захворювання і кількості захворювань в однієї людини, тривалості хвороби, кількістю ліжко-місце і аналізу захворюваності по районам і категоріям.

14. «**Автобусний парк**»: В системі повинні підтримуватися режими обліку транспортних засобів в автопарку, маршрутів, водіїв, обліку доходів і витрат (оплата за проїзд, ремонт і т. П.), Видачі звітів за запитами.

15. «**Розклад занять в університеті**»: В системі повинні підтримуватися режими пошуку заняття по заданому критерію (час, викладач), реєстрації занять, обліку занять по типу, генерації розкладів.

16. «**Школа**»: В системі повинні підтримуватися режими обліку класів та учнів в них, реєстрації нового учня, обліку відвідування занять і оцінок учнів, генерації звітів по успішності учнів.

17. «**Поліклініка**»: В системі повинні підтримуватися режими обліку докторів і пацієнтів, пошуку пацієнта по заданому критерію, аналізу захворюваності по районам, по місяцях і видачі довідок про хворобу.

18. **«Аптека»**: В системі повинні підтримуватися режими пошуку ліків по заданому критерію (назва, хвороба, ціна), замовлення, покупки і обліку ліків, аналізу попиту на ліки в залежності від вартості, пори року і т. п., видачі відповідних звітів.

19. **«Музей»**: В системі повинні підтримуватися режими обліку відвідувачів і експонатів музею, реєстрації нових експонатів, пошуку експонатів по заданому критерію (назва, епоха і т. Д.), Обліку доходів і витрат, пов'язаних з проведенням екскурсій, видачі звітів по запитам.

20. «**Ресторан**»: В системі повинні підтримуватися режими замовлення і пошуку страви по заданому критерію (назва, ціна), аналізу попиту на різні страви в залежності від ціни, часу дня, дня тижня, пори року і т. п., видачі відповідних звітів, генерації меню.

21. «**Ведення рахунків клієнтів у банку**». Зняття, поповнення грошей. Пошук клієнта

22. «**Щоденник метеоспостережень**». Реєструвати дані про погоду (температура, вологість, напрямок і сила вітру, тиск, хмарність / опади. З різних метеостанцій). На метеостанціях працюють співробітники. На основі даних будується прогноз на наступну добу.

23. «**Мийка автомобілів**». Різна вартість миття джипів, легкових, мікроавтобусів + прибирання салону. Знижки постійним клієнтам. Ведення обліку робіт персоналом. Графік чергувань.

24. «**Відділ кадрів**», список співробітників (поля: прізвище, ім'я, по батькові, дата народження, стать, освіти, номер документа про освіту, навчальний заклад, що видав документ, дата надходження на роботу, домашню адресу) по підрозділах. Історія роботи співробітників. Можливість роботи сумісником. Різні види пошуку.

25. «**Продаж квитків на міжміський автобус**». Заздалегідь відомо кількість місць. Необхідно продати квитки виходячи з наявності вільних місць (приблизний список полів: дата, номер рейсу, місце, ім'я гостя)

26. «**Список комп'ютерів**» на кафедрі з зазначенням характеристик та номера лабораторії, Матеріально-відповідального, заміни запчастин при ремонтах, гарантії.

Контрольні запитання

1. Навіщо потрібні засоби моделювання баз даних?
2. Які основні функції засобів моделювання баз даних?
3. Які відмінності логічної та фізичної моделей бази даних?
4. Як позначаються зв’язки між таблицями у вигляді зовнішнього ключа?
5. Як забезпечити третю нормальну форму?
6. Чому слід тримати модель бази даних актуальною до самої бази даних?
7. Що таке зворотній інжиніринг?

Лабораторна робота 6

Використання ін’єкції залежності за допомогою фреймворку Spring у додатках з MVC архітектурою.

Короткі теоретичні відомості

Ін’єкція залежностей - це шаблон проектування програмного забезпечення, який дозволяє змінювати жорсткі залежності в програмному коді на можливість їх зміни, під час виконання або під час компіляції програми. Основна ідея ін’єкції залежностей полягає в тому, щоб мати окремий об'єкт, асемблер, який заповнює поле в деякому класі екземпляром відповідної реалізації. Звичайно виділяють 2 види ін’єкції залежностей:

* через конструктор;
* через методи встановлення або за допомогою рефлексії безпосередньо у поля.

Безпосередньо з ін’єкцією залежностей використовується інверсія управління (Inversion of Control IoC) - шаблон проектування, який відповідає за вирішення залежностей компонентів, конфігурації та життєвого циклу. Це передбачає, що управління вказаними вище аспектами не повинен вирішуватись самим компонентом. В інверсію управління звичайно входять:

- голлівудський принцип: "Не дзвоніть нам, ми зателефонуємо вам";

- об'єкт програми, як правило, реєструється у фреймворку, який бере на себе відповідальність за виклик методів зареєстрованого об'єкта у відповідний час або як реакція на певну подію;

- управління інвертовано, оскільки замість коду програми, що викликає API фреймворку, все відбувається навпаки: фреймворк викликає методи зареєстрованих об’єктів.

Прикладом й найбільш використовуваним фреймворком для мови Java по керуванню залежностями є Spring. Існують також реалізації Spring для .NET, PHP, JavaScript. Перевагами Spring є те, шо він є полегшеним фреймворком для корпоративної розробки, володіє широкими можливостями по управлінню бізнес-об'єктами, надає базову модель програмування POJO (Plain Old Java Objects), спрощує використання існуючих технологій. Крім того, контейнер для управління залежностями незалежний, фреймворк теоретично працює і без контейнера.

Контейнер Spring для ін’єкції залежностей та інверсії управління представлений інтерфейсом org.springframework.context.ApplicationContext відповідає за створення, створення та налаштування вищезгаданих об’єктів. Контейнер, зчитуючі метадані конфігурації, отримує вказівки щодо того, які об’єкти створювати з налаштуванням їх залежностей. Метадані конфігурації можуть бути представлені у форматі XML, анотаціях або Java-коді. Це дозволяє описати об’єкти, що складають програму, та взаємозалежності між такими об’єктами. Наведемо приклад XML конфігурації.

<?xml v*ersion="1.0" encoding="UTF-8"?>*

*<****beans*** *xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">*

***<bean name="view" class="ua.kpi.view.View" />***

***<bean name="controller" class="ua.kpi.controller.Controller">***

***<property name="view" ref="view"/>***

***</bean>***

*</****beans****>*

Файл з конфігурацією звичайно розміщується у кореневій папці ресурсів. Нижче наведено приклад завантаження та отримання компоненту із контексту.

***ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml");***

***Controller controller = ctx.getBean("controller", Controller.class);***

Така конфігурація призначена для створення компонентів (beans) **controller** та **view** та ін’єкції залежності (заповнення поля **view** в екземплярі контроллера).

***public class Controller {***

***private View view;***

***...***

***}***

Наступний вид конфігурації - конфігурація на анотаціях. Для неї в XML файл додаються наступні рядки для підключення конфігурації на анотаціях та налаштування пакету з якого скануються компоненти з анотаціями. Рядки, що створюють компоненти у такому випадку не потрібні.

**<context:annotation-config></context:annotation-config>**

**<context:component-scan base-package="ua.kpi"/>**

Над класами компонентів ставиться анотація @Component, над полем для ін’єкції залежності - @Autowired. Наведемо приклад коду.

***@Component***

*public class Controller {*

***@Autowired***

*private View view;*

*...*

*}*

***@Component***

*public class View {*

*...*

*}*

Ще один тип конфігурації - за допомогою Java-коду наведено нижче. Для цього створюється спеціальний клас з анотацією @Configuration. Компоненти створюються за допомогою відповідних методів з анотацією @Bean.

**@Configuration**

@ComponentScan("ua.kpi")

public class Config {

**@Bean**

public View view(){

return new View();

}

**@Bean**

public Controller controller(){

return new Controller(view());

}

}

XML-файл з конфігурацією може бути такий самий як і у конфігурації на анотаціях. Можна обійтися і без XML- файлу, використовуючи іншу реалізацію контексту контейнера.

***ApplicationContext ctx = new AnnotationConfigApplicationContext(Config.class);***

***Controller controller = ctx .getBean("controller", Controller.class);***

Можна також використовувати комбінацію різних способів конфігурації, але останнім часом найчастіше використовується конфігурація за допомогою анотацій та Java-коду.

Завдання до лабораторної роботи

Використовуючи завдання лабораторної роботи №4 замінити породжуючі патерни на контекст Spring або іншого фреймворка впровадження залежностей. Частину конфігурації можна виконати на анотаціях, частину - у інший спосіб.

Контрольні запитання

1. Що таке ін'єкція залежностей?
2. Що таке інверсія управління?
3. Які є види ін’єкції залежностей?
4. Чим фреймворк відрізняється від бібліотеки?
5. Які функції контейнера компонентів?
6. Які є види конфігурації контейнера Spring?
7. Який тег використовується в XML - конфігурації для створення об’єкту?
8. Яка анотація використовується для створення об’єкту при конфігурації за допомогою анотацій?
9. Яка анотація використовується для ін’єкції залежностей над полем або конструктором компоненту.
10. Яка анотація використовується для створення об’єкту при конфігурації за допомогою Java-коду?

Лабораторна робота 7

Використання фреймворку Spring MVC для розробки Веб-застосунків.

Короткі теоретичні відомості

Всі веб-застосунків можна розділити на 2 види: односторінкові застосунки та багатосторінкові застосунки. В односторінковому застосунку весь необхідний код - HTML, JavaScript, та CSS - завантажується разом зі сторінкою, або динамічно завантажується за потребою. Сторінка не перенаправляє користувача до іншої сторінки у процесі роботи з нею. Таким чином, користувач отримує досвід використання дуже близький до desktop-застосунком. На клієнті (браузер) широко використовується JavaScript. Для взаємодії з сервером звичайно використовується (Ajax) Asynchronous JavaScript and XML. Замість XML останнім часом використовується передача даних у форматі JSON. До переваг односторінкових застосунків можна віднести вищу швидкість роботи застосунку, майже всі ресурси завантажуються один раз. Недоліком односторінкового застосунку є більш важка SEO-оптимізація.

Багатосторінкові застосунки є технологією, яка з’явилась на початку Веб-програмування. Генерація сторінки повністю відбувається на сервері. Перевагами є ситуації, коли за вимогами треба мати багато веб-сторінок та більш легка SEO-оптимізація у порівнянні з багатосторінковим застосунком. Недоліками багатосторінкових застосунків є більше навантаження на мережу та сервер, зв’язність розробки фронтенду та бекенду, значний час завантаження при переході на іншу сторінку.

Сучасні фреймворки підтримують розробку як односторінкових так і багатосторінкових застосунків. Розглянемо засоби фреймворку Spring MVC для розробки Веб-застосунків.

Для односторінкових застосунків використовується контроллер з анотацією **@Controller**. Його задача - читання параметрів HTTP запитів, заповнення атрибутами, отриманими з шару сервісів, об’єкту, який реалізує інтерфейс Model, та зв’язування запиту з представленням (View). Розглянемо реалізацію задач контроллеру на конкретному прикладі:

***@Controller***

*public class ApplicationController {*

***@Autowired***

*private StudentService studentService;*

***@GetMapping({"/", "/home"})***

*public String home() {*

*return "home";*

*}*

***@GetMapping("/students")***

*public String list(Model model) {*

*model.addAttribute("results", studentService.getAll());*

*return "students";*

*}*

***@GetMapping("/students/{id}")***

*public String student(Model model, @PathVariable("id") Integer id){*

*model.addAttribute("results", studentService.findById(id););*

*return "students";*

*}*

*}*

Метод *home()* з анотацією ***@GetMapping({"/", "/home"})*** зв’язує URI  ***"/", "/home"*** з шаблоном сторінки *templates/home.html*. Метод *list()*  з анотацією ***@GetMapping("/students")*** зв’язує URI  ***"/students"*** з шаблоном сторінки *templates/students.html*. Крім того, метод встановлює атрибут "results" моделі, списком об’єктів, отриманим з методу studentService.getAll(). Метод *student()* з анотацією ***@GetMapping("/students/{id}")*** зв’язує URI  ***"/student"*** з шаблоном сторінки *templates/student.html*. Крім того, метод зчитує з URI id об’єкта, встановлює атрибут "results" моделі, об’єктом, отриманим з методу studentService.*findById*(). При використанні Spring Boot можна використовувати залежність від Thymeleaf (шаблонізатор HTML сторінок):

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-thymeleaf</artifactId>*

*</dependency>*

Також можна використовувати JSP, або іншу технологію.

Для односторінкових додатків у Spring MVC використовується підтримка REST (Representational State Transfer — «передача стану відображення») — архітектурний стиль взаємодії компонентів розподіленого застосунку в мережі. Для передачі даних використовується протокол HTTP. Звичайно при реалізації REST виконуються певні домовленості. Для назв URI використовуються іменники, що відповідають предметній області. Для стандартних дій використовуються стандартні типи HTTP запитів:

* GET - для отримання інформації (GET призначений лише для отримання даних. GET - це операція, яка не повинна мати побічних ефектів, властивість також відома як ідемпотентність);
* POST - для додавання нової інформації;
* PUT - для оновлення інформації;
* DELETE - для видалення інформації.

Приклад реалізації REST для URI для наступного прикладу наведено у наступному прикладі.

***@RestController***

***@RequestMapping("api/students")***

*public class StudentController {*

*@Autowired*

*private StudentService studentService;*

***@GetMapping***

*public ResponseEntity<List<Student>> getAllStudents(){*

*return ResponseEntity.ok(studentService.getAll());*

*}*

***@GetMapping("/{id}")***

*public ResponseEntity getStudent(****@PathVariable("id")*** *int id){*

*return studentService.getStudentById(id).map( student->ResponseEntity.ok(student) )*

*.orElse(****ResponseEntity.notFound().build()****);*

*}*

***@PostMapping***

*public ResponseEntity createStudent(****@RequestBody*** *Student newStudent ){*

*if( newStudent.getId() != 0 ){*

*return ResponseEntity.badRequest().build();*

*}*

*return ResponseEntity.ok( studentService.create(newStudent));*

*}*

*}*

Анотація ***@RestController*** відповідає за підтримку REST контроллером. Анотація ***@RequestMapping("api/students")*** - відповідає за префікс URI для всіх методів контроллера. Анотація **@GetMapping** - відповідає за відображення кореневого URI (***api/students***) на метод, що повертає список студентів з автоматичним перетворенням об’єктів класу *Student* у формат JSON за допомогою Jackson. Анотація ***@GetMapping("/{id}")*** - відповідає за відображення URI формату ***api/students/число*** на метод *getStudent(),* який повертає об’єкт класу *Student* з автоматичним перетворенням у формат JSON. Анотація ***@PostMapping*** відповідає за відображення кореневого URI (***api/students***) на метод *createStudent(),* який приймає дані для створення об’єкту класу *Student* у форматі JSON із подальшим викликом відповідного методу шару сервісу.

Завдання до лабораторної роботи

Для предметної області із лабораторної роботи №1 або №5 створити REST ресурс із набором методів **GET, POST, PUT, DELETE** для принаймні одного об’єкту предметної області. Також необхідно розробити сторінку, згенеровану на сервері, для відображення списку об’єктів із предметної області.

Контрольні запитання

1. На які типи можна розділити Веб-застосунки?
2. Чим відрізняються односторінкові і багатосторінкові застосунків?
3. Які недоліки односторінкових застосунків?
4. Які переваги односторінкових застосунків?
5. Які недоліки багатосторінкових застосунків?
6. Які переваги багатосторінкових застосунків?
7. Що таке REST?
8. Що звичайно використовується для імен URI для REST?
9. Які є основні методи для REST ресурсів?
10. Для чого призначений метод HTTP запиту GET?
11. Що таке ідемпотентність?
12. Для чого призначений метод HTTP запиту POST?
13. Для чого призначений метод HTTP запиту PUT?
14. Для чого призначений метод HTTP запиту DELETE?
15. Які основні анотації використовуються у фреймворку Spring для розробки Веб-застосунків?

**СПИСОК ЛІТЕРАТУРИ**

1. Мартін Р. Чиста Архітектура. Мистецтво розробки програмного забезпечення. - Харків : Ранок, 2019. 368 с.

2. Брауде Э. Технология разработки программного обеспечения – СПб.: Питер, 2004. – 655с.

3. Пышкин Е. В. Основные концепции и механизмы объектно-ориентированного программирования – СПб.: БХВ-Петербург, 2005. – 640 с.

3. Соммервилл И. Инженерия программного обеспечения – М. : Вильямс, 2002. – 624 с.

4. Константайн Л. Разработка программного обеспечения – СПб.: Питер, 2004. – 592 с.

5. Буч Г., Джекобсон А. Язык UML. Руководство пользователя – М.: ДМК Пресс, 2001. – 257 с.

6. Хамбл, Д. Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ / Джез Хамбл. – М.:Издательский дом «Вильямс», 2011. – 436 с.

7. Грэхем И. Объектно-ориентированные методы. Принципы и практика – М.: Вильямс, 2004. – 880 с.

8. Рамбо Дж., Якобсон А., Буч. Г. UML: специальный справочник. – СПб.: Питер, 2002. – 656с.

9. Буч Г. Язык UML. Руководство пользователя – М.: ДМК Пресс; СПб.: Питер, 2004. – 432 с.

10. Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. – М.:Вильямс, 2010. – 448 с.