© Палєво ІО-ХХ
vk.com/io_palevo

МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Кафедра обчислювальної техніки

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт

з дисципліни

"Інженерія програмного забезпечення - 2"

Київ – 2015

Вступ

Дані лабораторні роботи виконувались студентами потоку ІО-2Х, у яких Болдак А.О. вів лекції та лабораторні роботи. Пояснювальну записку курсової роботи з цього предмету приймав Абу-Усхах О. Н., не факт, що Вас чекає теж саме, тому що, наскільки мені відомо, Абу-Усбах пішов з кафедри (але це не точно). Наступна викладка лабораторних робіт являється суто моїм баченням і в майбутньому можливі зміни. Розповім як у мене це було. У решти студентів, які самі зробили ці лабораторні (на жаль, таких було настільки мало, що Болдак палив палєво і питав по коду, по лабі і т.д.) приблизно теж саме.

Отже, є 4 лабораторні роботи, які собою складають курсову роботу. Потрібно зробити лабораторні та зв’язати їх компоненти між собою. Зв’язані лаби і працююча програма з базою даних складає собою курсову. Проте, її необхідно ще дуже багато оформлювати (ми здавали оформлення по частинах на бали, якщо спізнювались, з терміном здачі, мали вже менше балів, десь на половину).

Список лабораторних робіт:

1. Лабораторна робота. Конфігуратор програми
2. Лабораторна робота. Реалізація шаблону DAO
3. Лабораторна робота. Реалізація Контролера (з шаблону MVC)
4. Лабораторна робота. Графічний інтерфейс. (І курсова робота загалом).

Наскільки я зрозумів, задумали нам цю курсову роботу, щоб показати як працювати з базою даних і показати як писати програми, які потім можна буде з легкістю перероблювати. Якщо ваша програма – моноліт, то вносити в неї зміни досить незручно – краще застрілитись. Навпаки ж, коли програма модульна, із розділеними за функціональністю частинами, тоді змінити в ній щось запросто та не потрібно при цьому переписувати практично все з нуля. Це вкрай важливо і саме це ми маємо тут зрозуміти та навчитись.

Сама курсова загалом та лабораторні роботи зокрема, є частинами популярного шаблону MVC (Model-View-Controller). Спочатку зайдіть на Вікіпедію та почитайте про шаблон.

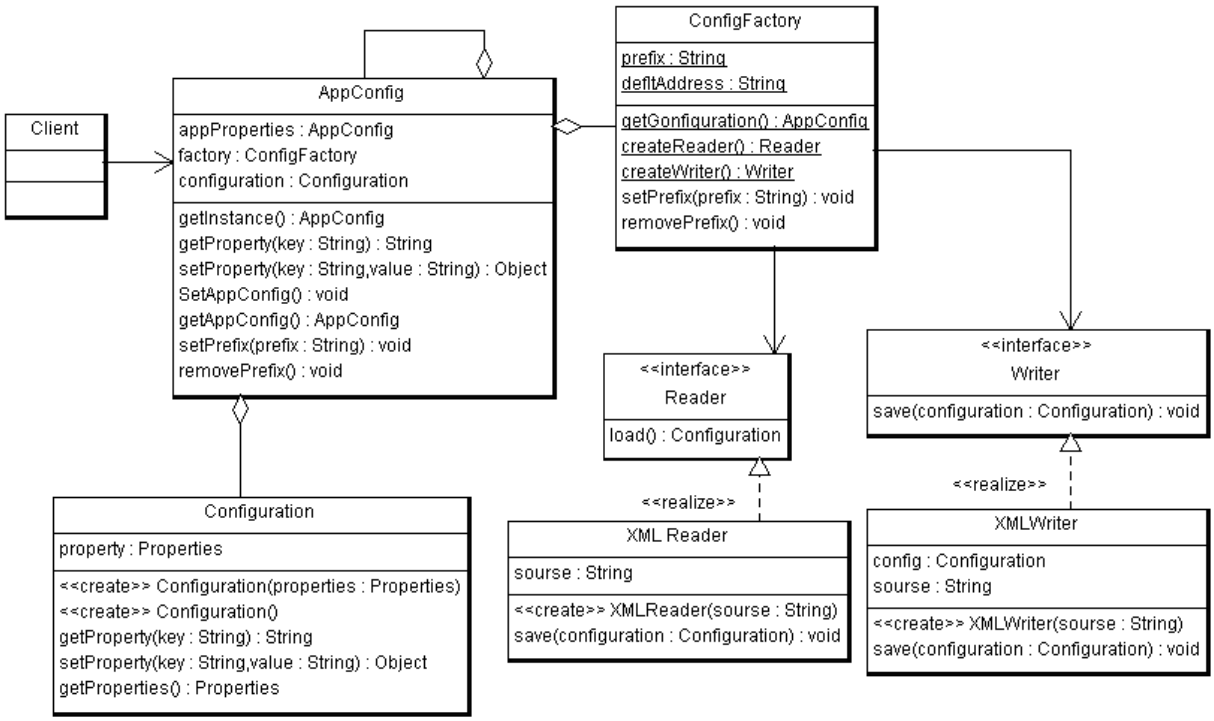
Лабораторна робота №1. Конфігуратор програми.

Задача лабораторної роботи полягає в тому, щоб написати програму для конфігурації проекту (простіше – програму для налаштування іншої програми). Ця програма буде як модуль для вашого курсового проекту.

Конфігуратор служить для того, щоб виокремити в модуль механізми налаштування програми. Зразу складно сказати, що можна налаштовувати, але в процесі розробки це стає ясно. У своїй курсовій я поклав на конфігуратор обов'язки налаштування і зберігання інформації для підключення до бази даних, останню тему інтерфейсу, що вибрав користувач та мову інтерфейсу. Взагалі, через конфігурацію можна запам'ятовувати будь-яку потрібну інформацію і змінювати параметри чи службову інформацію вашої програми.

Конкретне налаштування (конфігурація) тут − пара ключ-значення.

Діаграма класів Конфігуратора:



Ролі:

- **Configuration** – клас конфігурації. Об'єкти оперують з стандартним класом java.util.Properties. Клас відповідає за конкретні властивості, які містяться в масиві property.
- **Reader** – інтерфейс для зчитування налаштувань.
- **Writer** – інтерфейс для запису налаштувань.

- **XMLReader** – реалізація зчитування налаштувань з xml файлу (там помилка в назві методу в діаграмі. див код).
- **XMLWriter** – реалізація запису налаштувань в xml файл. Передаємо об'єкт Configuration, що подає конфігурацію, яку необхідно записати. У файл записуємо все, що знаходиться у полі property класу Configuration.
- **ConfigFactory** – фабрика, що породжує ріддери і райтери. Також відповідає за префікси. За допомогою префіксів я реалізовував різні файли, директорії.
- **AppConfig** – клас, що працює з клієнтом. Клієнту достатньо створити екземпляр, встановити префікс, а далі сеттером/геттером встановлювати/діставати конфігурації.

Отже, приблизно таку структуру треба сформувати та реалізувати. Не намагайтесь відтворити все точно як у мене, раджу скласти собі подібну функціональність, структуру конфігуратора і написати її.

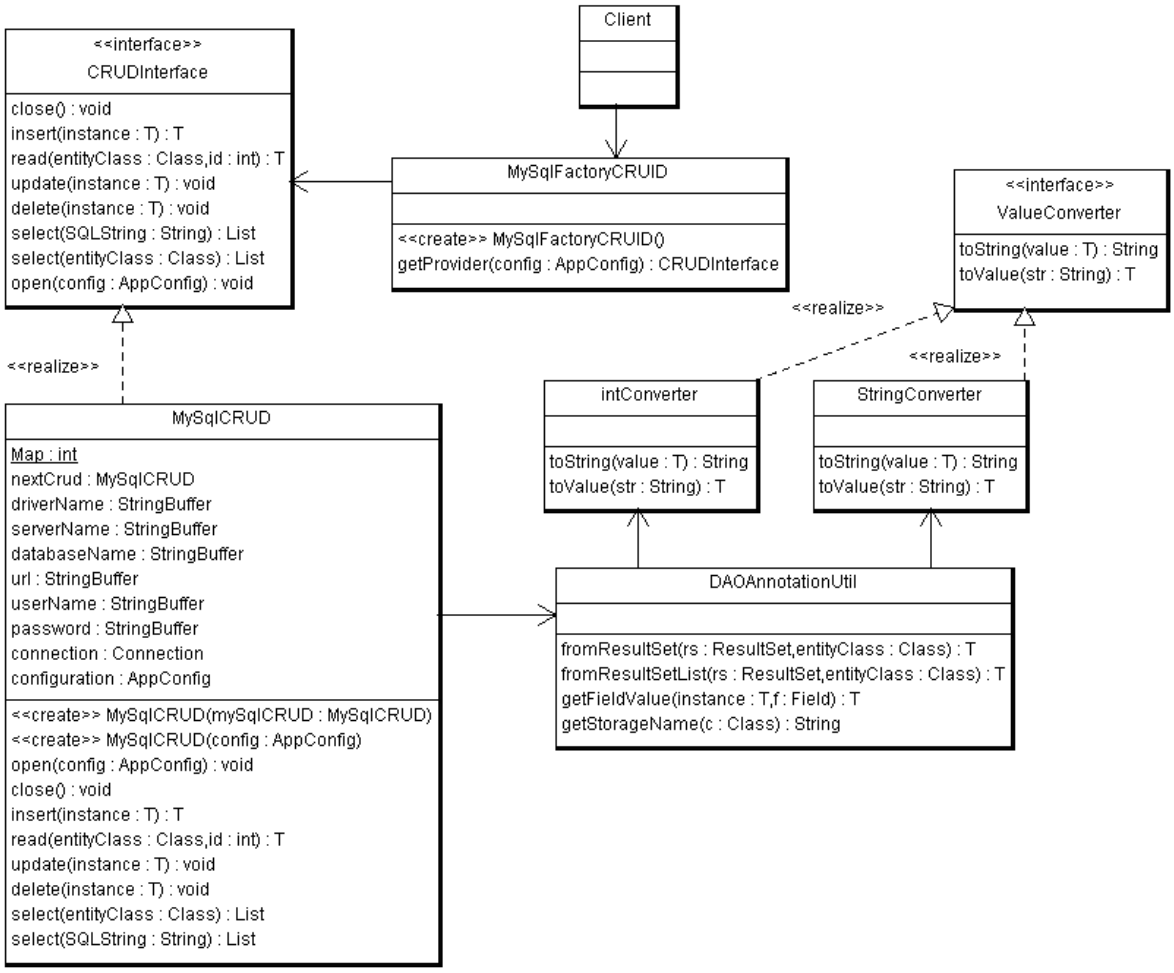
Лабораторна робота № 2. Реалізація шаблону DAO.

Із Вікіпедії:

« В программном обеспечении data access object (DAO) — это объект, который предоставляет абстрактный интерфейс к какому-либо типу базы данных или механизму хранения. Определённые возможности предоставляются независимо от того, какой механизм хранения используется и без необходимости специальным образом соответствовать этому механизму хранения.»

У чому полягає ідея: є об’єкт, за допомогою якого ми працюємо з базою, незалежно від того, яка у нас база (MySQL, Oracle і т.д.). Через цей об’єкт надсилаються запити та отримується результат.

Діаграма класів:



Інтерфейс CRUDInterface – це тип об’єкта DAO. MySqlCRUD – це конкретна реалізація для бази в MySql.

У чому полягає здача або що має робити клас MySqlCRUD? Цей клас виступає зв’язком між тим що приходить з програми і йде в базу та навпаки.

У базі даних ми маємо таблиці, які складаються з колонок. Для реалізації інтерфейсу (зв’язку з базою) нам треба знати в програмі, які ми маємо таблиці. Тобто, в java-коді треба мати якесь подання таблиць, щоб орієнтуватись як з коду надсилати запити та яким чином з коду їх отримувати. Для цього служать класи-моделі. Це класи, які повторюють назви таблиць, а поля в них відображають колонки таблиць. Для кожного поля повинен бути метод геттер і сеттер. Краще їх генерувати автоматично, самі ручками не пишіть.

Якщо звернете увагу на методи в інтерфейсі CRUDInterface, то вони чимось вам нагадають ключові слова при написанні запитів у базу даних. Ось і задача реалізації даного інтерфейсу. Ваш клас, що його реалізує, повинен формувати запити, за допомогою класів-моделі, і повертати ними ж результат.

Тепер у Вас є два шляхи – робити халяву чи чаклунство за допомогою рефлексії. От Ви маєте клас-моделі, тобто якусь таблицю. Вам хочеться написати для неї запит Select. У Вас в коді буде, наприклад, строка String s = new String ("select from category"). Цей запит вибере всі записи з таблиці Category і JDBC поверне результат об'єктом ResultSet. Далі саме для цієї команди треба написати метод, який буде з цього об'єкту вибирати записи і повертати екземпляр (якщо в таблиці один запис) чи масив екземплярів (якщо в таблиці багато записів) класу-моделі Category. Це халявний варіант. От скільки операцій і це лише для однієї таблиці, тільки один select. А якщо у Вас близько десяти таких таблиць? І для кожної таблиці select, insert, update, read. Можна застрілитись, пишучи стільки одноманітних операцій, у яких змінюватись буде лише тип – клас-моделі. Причому це ж має бути реалізація інтерфейсу. Якщо йти таким шляхом, буде просто, але буде чимало однотипного коду.

Не халявний варіант – це робити через рефлексію. Ідея полягає в тому, що в DAO інтерфейсі 5-10 універсальних методів і Ви їх реалізуєте. Універсальність їх полягає в тому, що при виклику, Ви вказуєте тип(клас) як параметр і уже код цього методу бере цей клас, перебирає його поля, будує динамічно строку із запитом та надсилає її в базу. Отримує стандартну відповідь і після цього, за допомогою цього ж класу (не об’єкту, а саме класу, типу), створює його об’єкт. Знаючи поля, по їх імені (там діє договір про іменування геттерів і сеттерів, того їх треба генерувати, а не писати самостійно, щоб не помилитись) викликає сеттери та заповнює поля об’єкта. Ось у коді явно Вам не відомо в який момент часу буде метод працювати з якою таблицею, це буде визначатись динамічно, під час роботи. У книзі Хорстмана за 2014 рік я бачив розділ про рефлексію (на жаль, коли я розбирав це, його не було), почитайте що там викладено. Тепер Ваш метод може виконувати операцію для всіх класів-моделі. Нижче наведу приклад одного з моїх методів, що виконує операцію Insert:

```
1.@Override
2.      /**
3.       * метод що записує інформацію в базу даних
4.       * @author Vova
5.       * @param instance клас моделі, що представляє таблицю для запису
6.       */
7.      public <T> T insert(T instance) throws Exception {
8.            try {
9.                  @SuppressWarnings("rawtypes")
10.                 Class class1 = instance.getClass();
11.                 StringBuilder str = new StringBuilder("INSERT INTO ");
12.                 StringBuilder str1 = new StringBuilder("(");
13.     str.append(DAOAnnotationUtil.getStorageName(instance.getClass()));
14.                 Field[] fields = instance.getClass().getDeclaredFields();
15.                 str.append("(");
16.                 for (Field field : fields) {
17.                 if(!field.getAnnotation(Stored.class).name().equals("id")) {
18.                             str1.append("?,");
19.                       }
20.                 }
21.           str1 = new StringBuilder(str1.substring(0, str1.length() - 1));
22.                 str1.append(")");
23.                 str = new StringBuilder(str.substring(0, str.length() - 1));
24.                 str.append(" VALUES");
25.                 str.append(str1);
26.           PreparedStatement p = connection.prepareStatement(str.toString(),
27.                             Statement.RETURN_GENERATED_KEYS);
28.
29.                 int i = 1;
30.                 for (Field field : fields) {
31.
32.           if (!field.getAnnotation(Stored.class).name().equals("id")) {
33.                       if (field.getAnnotation(Stored.class).converter()
34.                                   .equals(intConverter.class)) {
35.                 p.setInt(i, (Integer) DAOAnnotationUtil.getFieldValue(
36.                                         instance, field));
37.                             }
38.                       if (field.getAnnotation(Stored.class).converter()
39.                                   .equals(StringConverter.class)) {
40.                             p.setString(i, (String) DAOAnnotationUtil
41.                                   .getFieldValue(instance, field));
42.                       }
43.                       if (field.getAnnotation(Stored.class).converter()
44.                                         .equals(BooleanConverter.class)) {
45.                             p.setBoolean(i, (Boolean) DAOAnnotationUtil
46.                                   .getFieldValue(instance, field));
47.                       }
48.                       i++;
49.                 }
50.           }
51.                 p.executeUpdate();
52.           } catch (Exception e) {
53.                 e.printStackTrace();
54.           }
55.           return null;
56.     }
```

Ще така річ, з якою мало хто стикався – це анотації. Також у новому Хорстмані десь має бути. В інтернеті теж є, погугліть, розберіться. Коротко, анотації – це як допоміжна інформація про поля, методи класів. Ви уже стикались з анотацією @Override, яка означає, що ми перевизначаємо, реалізуємо метод. Або з анотаціями в JUnit, коли Ви тестували чи вилітає виключна ситуація. Ви позначали анотацією метод та вказували там клас виключної ситуації, який очікується. Якщо ситуація виникла, тоді супер – метод пройдений.

Тут ми писали свої анотації і ними позначали поля класів-моделі. Це робилось для того, щоб видавати результат з бази користувачеві уже об'єктами класів-моделі. Ось, наприклад, у строці 13 ми за допомогою анотації дізнаємось назву таблиці, яку відображає клас-моделі. Наприклад, для класу Category, позначеного анотацією ось так (строка 9):

```
1.          package app.model;
2.
3.           import dao.annotation.Stored;
4.           import dao.annotation.utils.converter.intConverter;
5.      /**
6.      * Клас моделі. Представляє собою категорію користувача
7.      * @author Vova
8.      */
9.      @Stored(name="category")
10.     public class Category {
11.            /**
12.            * ідентифікаційний номер категорії
13.            */
14.            @Stored(name="1", converter = intConverter.class)
15.            private int id;
16.            /**
17.             * категорія
18.            */
19.            @Stored(name="2")
20.     private String category;
 //решта
```

Решта анотацій у мене вказують порядок слідування полів (у базі це колонки) і конвертори. Конвертори призначені для того, щоб знаючи яке це поле, через анотацію використати той конвертер із строки, який необхідний, щоб не було помилки приведення типів. Наприклад, тут(строки 14) для поля id написано, що це перша колонка (name=1) і це поле типу int, тому треба int конвертер. Конвертер у нас в залежності від класу конвертує дані, наприклад, із Integer в String та навпаки. Про це далі детальніше.

Наскільки мені відомо, Болдак напрягав Вас читати дані з бази та записувати в базу з коду, тому на цьому детально не буду зупинятись, лише скажу, що наш об'єкт DAO повинен містити в собі Connection і тільки через цей Connection ми працюємо з базою. Взагалі, наш об'єкт DAO – це Одинак (шаблон Singlton). З'єднання з базою даних дороге задоволення, тому його треба раз відкрити, лише один раз!, і далі його використовувати, нових підключень робити не можна (дорого ж!). При завершенні роботи програми з'єднання треба закрити.

Про конвертери. Почитайте код та коментарі, далі має стати ясно. Повністю буде зрозуміло, якщо сядете і напишете подібне, інакше буде каша в голові.

На діаграмі в принципі ясно, хто за що відповідає та що робить. Знову ж ідея роботи DAO проста: зв'язок з базою, при цьому видаючи результати

об’єктами. І не важливо яка база. У нас є інтерфейс і через його методи ми спілкуємось з базою. Приклада халявного варіанту DAO в мене нема. Читайте код, запускайте і розбирайтесь як воно працює, технічні особливості того, що, наприклад, результат у JDBC подається через ResultSet і як з цим класом працювати – гуглите, там пояснено точно краще аніж я коли-небудь зможу.

І ще, ось тут можна використати конфігуратор. Далі приведу приклад як я зберігав налаштування для підключення до бази даних. Мій код є в палєві, там по пакетах все структуровано.

```
<entry key="driverName">com.mysql.jdbc.Driver</entry>
<entry key="password">123456</entry>
<entry key="url">jdbc:mysql://</entry>
<entry key="userName">root</entry>
<entry key="databaseName">database</entry>
<entry key="serverName">localhost</entry>
</properties>
```

Лабораторна робота №3. Контролер.

Якщо Ви думали, що DAO це складно, хочу розчарувати, DAO з рефлексією це просто (з нуля в тому, щоб зрозуміти і написати його весь-весь витратив 3 дні). А ось з контролером проблем буде багато (особливо з реалізацією команд). Для порівняння, на команди та інтерфейс пішло 2,5 тижні. Перш за все, тому що це модуль, який зв'язує всі частини (інтерфейс, DAO+база, конфігуратор). І ще там дійсно багато треба реалізовувати та вчитись реалізовувати.

Із Вікіпедії: «Model-view-controller (MVC, «модель-представление-контроллер», «модель-вид-контроллер») — схема использования нескольких шаблонов проектирования, с помощью которых модель данных приложения, пользовательский интерфейс и взаимодействие с пользователем разделены на три отдельных компонента таким образом, чтобы модификация одного из компонентов оказывала минимальное воздействие на остальные. Данная схема проектирования часто используется для построения архитектурного каркаса, когда переходят от теории к реализации в конкретной предметной области».

Вся програма на курсову роботу та її складові частини – лабораторні, являють собою архітектуру MVC. Модель – це база даних і DAO її складова частина. Конфігуратор особливо до MVC не відноситься, але також важлива річ. Графічний інтерфейс користувача – це вигляд. У нас він був реалізований за допомогою засобів бібліотеки Swing. У 1-2 томах Хорстмана про цю бібліотеку добре розписано. Контролер – це вузлова частина, яка обробляє команди з графічного інтерфейсу, зв'язується з базою і графічному інтерфейсу повертає результат. Жодних інших операцій не повинно бути в інтерфейсі.

Розглянемо детальніше команди, які йдуть від графічного інтерфейсу до контролера. Команда являє собою код, який повинен виконуватись, наприклад, якщо користувач натиснув на кнопку. Цей код у нашому випадку не міститься у лістенері графічного об'єкта(наприклад кнопки), а знаходиться в класі, що реалізує інтерфейс Command, який, у свою чергу, наслідується від інтерфейсу Runnable. Виходить, кожна команда – це окремий потік і всі операції, які виконує ця команда, виконуються у методі run.

При виконанні команди, потрібно якось повертати результат в інтерфейс (ось натиснули ми кнопку, створили об'єкт команди, віддали її контролеру, контролер запустив потік на виконання. Болдак нам не розказував (напряг самим думати) як реалізувати повернення інформації назад у інтерфейс. Ми зробили у графічному інтерфейсі метод update(), який буде оновлювати графічний інтерфейс. Посилання на об'єкт графічного інтерфейсу посилали з командою в контролер і якраз метод setDataView встановлював посилання на графічний інтерфейс у полі команди. У певний момент, коли в потоці (в коді

методу run) ми отримували, наприклад, інформацію з бази, в об’єкта графічного інтерфейсу викликали метод update і передавали йому результати виконання команди. Код методу update, у свою чергу, лише оновлював графічний інтерфейс з новими даними, які йому передали.

Проте в кінці у мене виникла крутіша ідея – використати шаблон Observer. Проте думати і реалізовувати повернення інформації у графічний інтерфейс через Observer часу не було. Можливо серед Вас знайдуться ті, хто зробить таке, це буде круто.

У метод toController() надходить посилання на об’єкт графічного інтерфесу, анотація про параметри та опціонал (це ми мали брати в Болдака).

Приклад коду:

```
@COMMAND(key = "RegistrationCommand")
@CONTEXT(list = {
@PARAMETER(key = "Profile", type = Profile.class, optional = true),
@PARAMETER(key = "Category", type = Category.class, optional = true),
@PARAMETER(key = "ContactInformation", type = ContactInformation.class, optional =
true),
@PARAMETER(key = "State", type = State.class, optional = true),
@PARAMETER(key = "password1", type = String.class, optional = true),
@PARAMETER(key = "password2", type = String.class, optional = true) })
class cmnd extends RegistrationCommand {
}

Controller.toController(Registration.this,
cmnd.class.getAnnotation(CONTEXT.class),
cmnd.class.getAnnotation(COMMAND.class));
```

Теоретично, опціонал та параметри служать ключами для ідентифікації команди. Команд може бути багато, але за такими параметрами ми можемо шукати унікальну команду, створювати її та працювати з нею. Як у мене це працює: з графічного інтерфейсу в контролер надходить посилання на сам графічний інтерфейс та інформація (важливо, саме інформація, а не об’єкт команди) у вигляді анотацій:

```
        //код лістенера якогось графічного об’єкта


        Authorisation.this.profile.setPassword(passwordField.getText());

@COMMAND(key = "AutorisationCmnd")

@CONTEXT(list = {

        @PARAMETER(key = "login", type = String.class, optional = true),

        @PARAMETER(key = "password", type = String.class, optional = true) })

 class cmd extends AutorisationCommand {

        }

              Controller.toController(Authorisation.this,
          cmd.class.getAnnotation(CONTEXT.class),
          cmd.class.getAnnotation(COMMAND.class));

 category = Authorisation.this.getCategory();
```

```
//якийсь код далі
```

У контролері, у хеш-таблиці за цією інформацією знаходжу клас команди, який іде на валідацію (ще одна важлива річ, але тут вона практично не використовується. Коли у нас багато команд, валідація (наприклад, перевірка формату чи ще чогось) також необхідна. В моєму випадку валідація – це перевірка чи поле в анотації optional==true. Якщо умова виконується, команда валідна) і створюється екземпляр знайденого класу. Йому передаю посилання на графічний інтерфейс і відправляю на виконання.

Ось код цих операцій.

```
        /**
         * метод який валідує та виконує команду
         * @param dataView силка вигляду
         * @param context контекст команди
         * @param command команда
         */
        @SuppressWarnings("unchecked")
        public static synchronized void toController(Object dataView,
                        CONTEXT context, COMMAND command) {
                //валідація
                if (Validator.isValide(hashMap.get(command.key()), context)){
                        try {
                                //створення екземпляру команди
                                Command cmnd = (Command)
hashMap.get(command.key()).newInstance();
                                //встановлення посилання на граф інтерфейс
                                        cmnd.setDataView(dataView);
                                        //виконання
                                executor.execute(cmnd);
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                }
        }
```

Одночасно команд може бути багато. За вказівкою Болдака, ми робили чергу команд. В Java є такий механізм, що отримує потоки і виконує їх в черзі, один за другим:

private static Executor executor = Executors.newSingleThreadExecutor();

Далі строчка executor.execute(cmnd): ставить нашу команду в чергу та виконує її, викликаючи метод run, коли прийде її черга.

Таким чином є три частини, які чітко знають свої ролі: модель – зберігає дані та видає чи приймає їх, вигляд – їх відображає, а контролер обробляє та зв'язується з інтерфейсом і моделлю.

На прикладі, що може реалізовуватись у команді. Візьмемо авторизацію. Користувач вводить у полях свій логін і пароль. Коли поля заповнені, користувач натискає кнопку ОК і в лістенері цієї кнопки ми формуємо інформацію про команду (не саму команду) та відправляємо її Контролеру. Контролер за інформацією шукає клас команди, валідує його. Створює екземпляр класу та віддає його в чергу. Коли настає черга команди, виконується її метод run. У цьому методі ми через об'єкт DAO звертаємось у базу, вибираємо там логіни і паролі, шукаємо збіг. Збіг є/нема, в залежності від результату викликаємо метод update() у графічного об'єкта (панелі чи фрейму,

в якому ця авторизація, тощо) і метод update, наприклад, каже користувачу, що неправильний логін/пароль (збіг не знайдено).

В палєві є мій код з документацією, там можна розбиратись детальніше.

Лабораторна робота №4. Графічний інтерфейс.

Її я здавав разом із 3 роботою. Готовий інтерфейс, усі готові команди, усе працює, налаштовується, входить в базу, видає результат. Як працюють елементи графічного інтерфейсу почитаєте в книгах. Скажу лише, що не пишіть все самі. У Eclipse є плагін – Windows Builder. Там є кнопки, фрейми, текстові поля, текстові мітки і т.д. Складаєте там інтерфейс, віконця і Вам автоматично генерується код. З ним і працюєте.

Корисним буде наступне посилання. Там показаний опис і приклади використання всіх можливих Swing компонентів. http://www.math.uni-hamburg.de/doc/java/tutorial/uiswing/components/components.html

У палєві також є пояснювальна записка моєї курсової роботи, у ній пояснюються всі елементи: від конфігуратора до команд і контролера.

Висловлюю величезну подяку моєму редактору, без якого цей документ не можливо було б навіть читати, не те що зрозуміти. Хоч і мої викладки та грамотність жахливі, тобі вдалося їх хоч трішки зробити хорошими.

Кузьменко Володимир
група ІО – 21

Київ – 2015