**ЛАБОРАТОРНАЯ РАБОТА №6**

по дисциплине «Модульное тестирование»

на тему

«**ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ»**

Автор: студент гр. 10701118 Воробей И.А.

Проверил: доцент, к.т.н. Попова Ю.Б.

**Цель работы**: закрепление практических навыков по проведению интеграционного тестирования.

**Задание:**

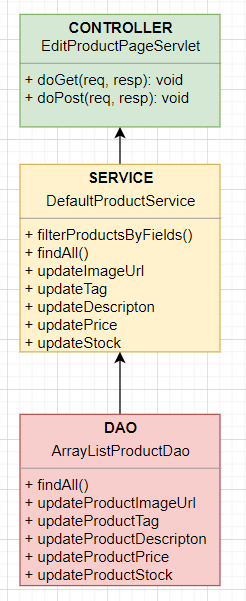
1. Разработать интеграционные тесты для методов, используемых в предыдущих лабораторных работах. Предусмотреть проверки вызовов определенных функций внутри тестируемого метода**.**

2. Запустить разработанные тесты. Занести результаты пункта в отчет в виде копий экранов с не пройденными тестами и листингом кода.

3. Произвести разработку кода, не прибегая к изменению тестов. Запустить тесты. Занести результаты пункта в отчет в виде копий экранов с результатами тестов и листингом разработанного кода.

4. Произвести рефакторинг кода и тестов. В отчет занести изменения по коду и комментарии к ним.

5. Сделать выводы по работе.

Тестирую взаимодествие слоев.

Сервис и дао являются синглтонами.

Методы контролера:

* doGet(HttpServletRequest, HttpServletResponce)

сетает в реквест список продуктов

* doPost(HttpServletRequest, HttpServletResponce) обновляет поля продуктов(цена, количество...)удовлетворяющих параметрам запроса

request & respnce необходимо замокать, все остальные объеты реальные.

Dao хранит список продуктов в листе. Так же дао содержит сетер для листа.

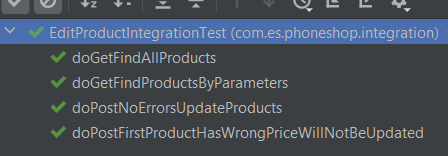
В классе теста я объявлю свой лист и проинициализрую его, потом засетаю в дао. Это позволит отслеживать изменения на моем тестовом объетке листа.

При тестировынии doGet() проверяю каой список продуктов вернулся из сервиса. Сначала теслирую без параметров поиска, потом с параметрами.

Когда тестирую doPost() сравниваю лист который я засетал в дао(после вызова метода он должен изменится) с придполпгаемым листом. Так же тестирую с неверным параметром.

@RunWith(MockitoJUnitRunner.class)  
public class EditProductIntegrationTest {  
 @Mock  
 private HttpServletRequest request;  
 @Mock  
 private HttpServletResponse response;  
 @Mock  
 private RequestDispatcher requestDispatcher;  
 private final EditProductPageServlet servlet = new EditProductPageServlet();  
 private ProductDao productDao = ArrayListProductDao.*getInstance*();  
 private Currency usd = Currency.*getInstance*("USD");  
 private List<Product> products;  
  
 private List<Product> getSampleProducts() {  
 List<Product> productList = List.*of*(  
 new Product("Samsung", "Samsung Galaxy S", new BigDecimal(100), usd, 100, "a"),  
 new Product("Samsung", "Samsung Galaxy S II", new BigDecimal(200), usd, 0, "a"),  
 new Product("Samsung", "Samsung Galaxy S III", new BigDecimal(300), usd, 5, "a"),  
 new Product("Apple", "Apple iPhone", new BigDecimal(200), usd, 10, "a"));  
 for (int i = 0; i < productList.size(); i++) {  
 productList.get(i).setId((long) i + 1);  
 }  
 return productList;  
 }  
  
 @Before  
 public void setup() {  
 products = getSampleProducts();  
 productDao.setItems(products);  
 *when*(request.getContextPath()).thenReturn("contextPath");  
 *when*(request.getParameterValues("productId")).thenReturn(new String[]{"1", "2"});  
 *when*(request.getParameterValues("imageUrl")).thenReturn(new String[]{"1url", "2url"});  
 *when*(request.getParameterValues("tag")).thenReturn(new String[]{"htc", "palm"});  
 *when*(request.getParameterValues("description")).thenReturn(new String[]{"d1", "d2"});  
 *when*(request.getParameterValues("price")).thenReturn(new String[]{"1", "2"});  
 *when*(request.getParameterValues("stock")).thenReturn(new String[]{"1", "2"});  
 *when*(request.getRequestDispatcher("WEB-INF/pages/editProduct.jsp")).thenReturn(requestDispatcher);  
 }  
  
 @Test  
 public void doGetFindAllProducts() throws ServletException, IOException {  
 servlet.doGet(request, response);  
  
 *verify*(request).setAttribute("products", getSampleProducts());  
 }  
  
 @Test  
 public void doGetFindProductsByParameters() throws ServletException, IOException {  
 *when*(request.getParameter("searchTag")).thenReturn("Samsung");  
 *when*(request.getParameter("minPrice")).thenReturn("70");  
 *when*(request.getParameter("maxPrice")).thenReturn("200");  
 *when*(request.getParameter("searchStock")).thenReturn("80");  
 Product soughtProduct = new Product("Samsung", "Samsung Galaxy S", new BigDecimal(100),  
 usd, 100, "a");  
 soughtProduct.setId(1l);  
  
 servlet.doGet(request, response);  
  
 *verify*(request).setAttribute("products", List.*of*(soughtProduct));  
 }  
  
 @Test  
 public void doPostNoErrorsUpdateProducts() throws ServletException, IOException {  
 List<Product> actual = products;  
 List<Product> expected = List.*of*(  
 new Product("htc", "d1", BigDecimal.*valueOf*(1d), usd, 1, "1url"),  
 new Product("palm", "d2", BigDecimal.*valueOf*(2d), usd, 2, "2url"),  
 new Product("Samsung", "Samsung Galaxy S III", new BigDecimal(300), usd, 5, "a"),  
 new Product("Apple", "Apple iPhone", new BigDecimal(200), usd, 10, "a"));  
 for (int i = 0; i < expected.size(); i++) {  
 expected.get(i).setId((long) i + 1);  
 }  
  
 servlet.doPost(request, response);  
  
 *assertEquals*(expected, actual);  
 }  
  
 @Test  
 public void doPostFirstProductHasWrongPriceWillNotBeUpdated() throws ServletException, IOException {  
 *when*(request.getParameterValues("price")).thenReturn(new String[]{"notParsed", "2"});  
 List<Product> actual = products;  
 List<Product> expected = List.*of*(  
 new Product("htc", "d1", BigDecimal.*valueOf*(100), usd, 1, "1url"),  
 new Product("palm", "d2", BigDecimal.*valueOf*(2d), usd, 2, "2url"),  
 new Product("Samsung", "Samsung Galaxy S III", new BigDecimal(300), usd, 5, "a"),  
 new Product("Apple", "Apple iPhone", new BigDecimal(200), usd, 10, "a"));  
 for (int i = 0; i < expected.size(); i++) {  
 expected.get(i).setId((long) i + 1);  
 }  
  
 servlet.doPost(request, response);  
  
 *assertEquals*(expected, actual);  
 }  
}

Результаты



Выводы: проверил взаимодействие трех модулей своей программы. Модули были предварительно протестированы отдельно. Программа работает как предполагается.

Я привожо код тестируемых классов полностью, некоторые методы(сервиса и дао) не задествованы в этом тесте.

CONTROLLER

public class EditProductPageServlet extends HttpServlet {  
 private static final String *ADVANCED\_SEARCH\_JSP* = "WEB-INF/pages/editProduct.jsp";  
 private ProductService productService = DefaultProductService.*getInstance*();  
  
 @Override  
 public void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 String productCode = req.getParameter(Const.RequestParam.*SEARCH\_TAG*);  
 String potentialMinPrice = req.getParameter(Const.RequestParam.*MIN\_PRICE*);  
 String potentialMaxPrice = req.getParameter(Const.RequestParam.*MAX\_PRICE*);  
 String potentialMinStock = req.getParameter(Const.RequestParam.*SEARCH\_STOCK*);  
 Map<String, String> errors = new HashMap<>();  
 Double minPrice = handleNonNegativeDouble(potentialMinPrice, Const.ErrorKey.*MIN\_PRICE*, errors);  
 Double maxPrice = handleNonNegativeDouble(potentialMaxPrice, Const.ErrorKey.*MAX\_PRICE*, errors);  
 Integer minStock = handleNonNegativeInt(potentialMinStock, Const.ErrorKey.*STOCK*, errors);  
 if (errors.isEmpty()) {  
 req.setAttribute(Const.AttributeKey.*PRODUCTS*, productService.filterProductsByFields(  
 productService.findAll(), minPrice, maxPrice, minStock, productCode));  
 } else {  
 req.setAttribute(Const.AttributeKey.*SEARCH\_ERRORS*, errors);  
 }  
 req.getRequestDispatcher(*ADVANCED\_SEARCH\_JSP*).forward(req, resp);  
 }  
  
 @Override  
 public void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 String[] productIds = req.getParameterValues(Const.RequestParam.*PRODUCT\_ID*);  
 String[] imageUrls = req.getParameterValues(Const.RequestParam.*IMAGE\_URL*);  
 String[] tags = req.getParameterValues(Const.RequestParam.*TAG*);  
 String[] descriptions = req.getParameterValues(Const.RequestParam.*DESCRIPTION*);  
 String[] prices = req.getParameterValues(Const.RequestParam.*PRICE*);  
 String[] stocks = req.getParameterValues(Const.RequestParam.*STOCK*);  
 Map<Long, Map<String, String>> productsErrors = new HashMap<>();  
 for (int i = 0; i < productIds.length; i++) {  
 long id = Long.*parseLong*(productIds[i]);  
 Map<String, String> errors = new HashMap<>();  
 updateString(id, imageUrls[i], errors, Const.ErrorKey.*IMAGE\_URL*, productService::updateImageUrl);  
 updateString(id, tags[i], errors, Const.ErrorKey.*TAG*, productService::updateTag);  
 updateString(id, descriptions[i], errors, Const.ErrorKey.*DESCRIPTION*, productService::updateDescription);  
 updateDouble(id, prices[i], errors, Const.ErrorKey.*PRICE*, productService::updatePrice);  
 updateInt(id, stocks[i], errors, Const.ErrorKey.*STOCK*, productService::updateStock);  
 if (!errors.isEmpty()) {  
 productsErrors.put(id, errors);  
 }  
 }  
 if (!productsErrors.isEmpty()) {  
 req.setAttribute(Const.AttributeKey.*ERRORS*, productsErrors);  
 doGet(req, resp);  
 }else{  
 resp.sendRedirect(req.getContextPath() + "/editProduct");  
 }  
 }  
  
 public Double handleNonNegativeDouble(String potentialDouble, String errorKey, Map<String, String> errors) {  
 if (potentialDouble == null || potentialDouble.isEmpty()) {  
 return null;  
 }  
 Optional<Double> number = CustomParser.*parseNonNegativeDouble*(potentialDouble);  
 if (number.isEmpty()) {  
 errors.put(errorKey, Const.ErrorInfo.*NON\_NEGATIVE\_NUMBER*);  
 }  
 return number.orElse(null);  
 }  
  
 public Integer handleNonNegativeInt(String potentialInteger, String errorKey, Map<String, String> errors) {  
 if (potentialInteger == null || potentialInteger.isEmpty()) {  
 return null;  
 }  
 Optional<Integer> number = CustomParser.*parseNonNegativeInt*(potentialInteger);  
 if (number.isEmpty()) {  
 errors.put(errorKey, Const.ErrorInfo.*NON\_NEGATIVE\_INT*);  
 }  
 return number.orElse(null);  
 }  
  
 private void updateString(long productId, String imageUrl, Map<String, String> error, String errorKey,  
 BiConsumer<Long, String> consumer) {  
 if (imageUrl.isEmpty()) {  
 error.put(errorKey, Const.ErrorInfo.*VALUE\_IS\_REQUIRED*);  
 } else {  
 consumer.accept(productId, imageUrl);  
 }  
 }  
  
 private void updateDouble(long productId, String potentialDouble, Map<String, String> error, String errorKey,  
 ObjDoubleConsumer<Long> consumer) {  
 CustomParser.*parseNonNegativeDouble*(potentialDouble)  
 .ifPresentOrElse(d -> consumer.accept(productId, d),  
 () -> error.put(errorKey, Const.ErrorInfo.*NON\_NEGATIVE\_NUMBER*));  
 }  
  
 private void updateInt(long productId, String potentialInt, Map<String, String> error, String errorKey,  
 ObjIntConsumer<Long> consumer) {  
 CustomParser.*parseNonNegativeInt*(potentialInt)  
 .ifPresentOrElse(i -> consumer.accept(productId, i),  
 () -> error.put(errorKey, Const.ErrorInfo.*NON\_NEGATIVE\_INT*));  
 }  
}

SERVICE

public class DefaultProductService implements ProductService {  
 private ArrayListProductDao productDao = ArrayListProductDao.*getInstance*();  
  
 private DefaultProductService() {  
 }  
  
 private static class DefaultAdvancedSearchServiceHolder {  
 private static final DefaultProductService *DEFAULT\_PRODUCT\_SERVICE\_INSTANCE* =  
 new DefaultProductService();  
 }  
  
 public static DefaultProductService getInstance() {  
 return DefaultAdvancedSearchServiceHolder.*DEFAULT\_PRODUCT\_SERVICE\_INSTANCE*;  
 }  
  
 @Override  
 public void save(Product product) {  
 productDao.save(product);  
 }  
  
 @Override  
 public List<Product> findProducts(String query, SortField sortField, SortOrder sortOrder) {  
 return productDao.findProducts(query, sortField, sortOrder);  
 }  
  
 @Override  
 public List<Product> filterProductsByFields(  
 List<Product> items, Double minPrice, Double maxPrice, Integer minStock, String... productTags) {  
 return items.stream()  
 .filter(i -> productTags == null || productTags.length == 0 || productTags[0] == null ||  
 productTags[0].isEmpty() || Arrays.*asList*(productTags).contains(i.getTag()))  
 .filter(i -> minStock == null || i.getStock() >= minStock)  
 .filter(i -> minPrice == null || i.getPrice().doubleValue() >= minPrice)  
 .filter(i -> maxPrice == null || i.getPrice().doubleValue() <= maxPrice)  
 .collect(Collectors.*toList*());  
 }  
  
 @Override  
 public List<String> findTags() {  
 return productDao.findTags();  
 }  
  
 @Override  
 public Product find(long productId) throws ItemNotFoundException {  
 return productDao.find(productId);  
 }  
  
 @Override  
 public List<Product> findAll() {  
 return productDao.findAll();  
 }  
  
 @Override  
 public void saveProduct(String imageUrl, String productCode, String description, BigDecimal price, int stock) {  
 Product product = new Product();  
 product.setImageUrl(imageUrl);  
 product.setTag(productCode);  
 product.setDescription(description);  
 product.setPrice(price);  
 product.setStock(stock);  
 product.setCurrency(Currency.*getInstance*("USD"));  
 productDao.save(product);  
 }  
  
 @Override  
 public void delete(long productId) {  
 productDao.delete(productId);  
 }  
  
 @Override  
 public void updateImageUrl(long productId, String imageUrl) {  
 productDao.updateProductImageUrl(productId, imageUrl);  
 }  
  
 @Override  
 public void updateTag(long productId, String tag) {  
 productDao.updateProductTag(productId, tag);  
 }  
  
 @Override  
 public void updateDescription(long productId, String description) {  
 productDao.updateProductDescription(productId, description);  
 }  
  
 @Override  
 public void updatePrice(long productId, double price) {  
 productDao.updateProductPrice(productId, price);  
 }  
  
 @Override  
 public void updateStock(long productId, int stock) {  
 productDao.updateProductStock(productId, stock);  
 }  
}

DAO

public class ArrayListProductDao extends ArrayListGenericDao<Product> implements ProductDao {  
 private static final String *BLANK* = "\\p{Blank}";  
  
 private ArrayListProductDao() {  
 }  
  
 private static class SingletonHolder {  
 public static final ArrayListProductDao *ARRAY\_LIST\_PRODUCT\_DAO\_INSTANCE* = new ArrayListProductDao();  
  
 private SingletonHolder() {  
 }  
 }  
  
 public static ArrayListProductDao getInstance() {  
 return SingletonHolder.*ARRAY\_LIST\_PRODUCT\_DAO\_INSTANCE*;  
 }  
  
 @Override  
 public synchronized void updateProductImageUrl(long productId, String imageUrl) {  
 items.stream().filter(p -> p.getId().equals(productId)).findAny().ifPresent(p -> p.setImageUrl(imageUrl));  
 }  
  
 @Override  
 public synchronized void updateProductDescription(long productId, String description) {  
 items.stream().filter(p -> p.getId().equals(productId)).findAny().ifPresent(p -> p.setDescription(description));  
 }  
  
 @Override  
 public synchronized void updateProductTag(long productId, String tag) {  
 items.stream().filter(p -> p.getId().equals(productId)).findAny().ifPresent(p -> p.setTag(tag));  
 }  
  
 @Override  
 public synchronized void updateProductPrice(long productId, double price) {  
 items.stream().filter(i -> i.getId().equals(productId)).findAny()  
 .ifPresent(p -> {  
 p.setPrice(BigDecimal.*valueOf*(price));  
 p.setPriceHistory(LocalDate.*now*(), BigDecimal.*valueOf*(price));  
 });  
 }  
  
 @Override  
 public synchronized void updateProductStock(long productId, int stockValue) {  
 items.stream().filter(i -> i.getId().equals(productId)).findAny().ifPresent(p -> p.setStock(stockValue));  
 }  
  
 @Override  
 public synchronized List<String> findTags() {  
 return items.stream().map(Product::getTag).distinct().collect(Collectors.*toList*());  
 }  
  
 @Override  
 public synchronized List<Product> findProducts(String query, SortField sortField, SortOrder sortOrder) {  
 if (sortField != SortField.*DEFAULT* && sortOrder != SortOrder.*DEFAULT*) {  
 Comparator<Product> comparator = Comparator.*comparing*(p -> defineSortField(p, sortField));  
 if (sortOrder == SortOrder.*DESC*) {  
 comparator = comparator.reversed();  
 }  
 return search(query).sorted(comparator).collect(Collectors.*toList*());  
 }  
 return search(query).collect(Collectors.*toList*());  
 }  
  
 private <U> U defineSortField(Product p, SortField sortField) {  
 return (U) (SortField.*DESCRIPTION* == sortField ? p.getDescription() : p.getPrice());  
 }  
  
 private Stream<Product> search(String query) {  
 return items.stream()  
 .filter(getPriceAndStockPredicate(StringUtils.*isBlank*(query)))  
 .filter(p -> Arrays.*stream*(query.split(*BLANK*)).anyMatch(p.getDescription()::contains))  
 .sorted(Comparator.*comparing*(p -> Arrays.*stream*(query.split(*BLANK*))  
 .filter(p.getDescription()::contains).count(), Comparator.*reverseOrder*()));  
 }  
  
 private Predicate<Product> getPriceAndStockPredicate(boolean queryPresent) {  
 if (queryPresent) {  
 return product -> product.getPrice() != null && product.getStock() > 0;  
 } else {  
 return product -> true;  
 }  
 }  
}