**Narrative**

The artifact is a Contact class. Contact ID, first and last names, phone number, and address are among the variables it contains. Constructors for initializing these variables and methods for validating, obtaining, and setting their values are also included in the class. In order to ensure safe and effective processing of contact information, this artifact was made as part of a software project. This was created to use validation checks and particular exceptions to guarantee data integrity and appropriate error handling.

Because the Contact class captures many essential elements of software development, such as object-oriented programming, data validation, error handling, and acceptable coding techniques, I chose it as an asset for my ePortfolio. Managing contact information is a typical requirement in many software projects, and it is a real implementation of these notions.

Yes, I was able to meet the intended course outcomes from Module One thanks to the improvements made to the Contact class. These improvements specifically showed my capacity to create and execute software solutions that are safe, reliable, and maintainable. I demonstrated my abilities to write excellent code that complies with best practices by reworking the validation logic, enhancing error handling, and included thorough documentation. I do not have any updates to my outcome-coverage plans at this moment. The enhancements have effectively addressed the planned outcomes, and I will continue to apply these principles and best practices to other aspects of the project to further meet and reinforce these course outcomes.

**Target Areas for Improvement**

Here are some target areas for improvement for the Contact class to enhance its structure, validation, error handling, and documentation:

**Structure**

**Refactor Validation Logic**: Consider refactoring the validation logic into separate methods for better readability and reusability. This will make the code cleaner and more maintainable.

java

// Validation methods

private void validateContactId(String contactId) {

if (contactId == null || contactId.length() > 10) {

throw new IllegalArgumentException("Invalid ID: contactId must not be null and must be 10 characters or fewer.");

}

}

private void validateName(String name, String fieldName) {

if (name == null || name.length() > 10) {

throw new IllegalArgumentException(fieldName + " should be no more than 10 characters.");

}

}

private void validatePhone(String phone) {

if (phone == null || phone.length() != 10) {

throw new IllegalArgumentException("Phone Number should be exactly 10 characters.");

}

}

private void validateAddress(String address) {

if (address == null || address.length() > 30) {

throw new IllegalArgumentException("Address should be no more than 30 characters.");

}

}

**Validation**

**Enhance Validation Messages**: Enhance the validation messages to be more descriptive and user-friendly. This will help users understand what went wrong and how to fix it.

**Error Handling**

**Provide More Specific Exceptions**: Improve error handling by providing more specific exceptions or error codes. This helps in identifying and troubleshooting issues more effectively.

public void setFirstName(String fName) {

validateName(fName, "First Name");

this.firstName = fName;

}

**4. Commenting and Documentation**

**Add Comments and Documentation**: Add comments and documentation to explain the purpose of each method and the validation rules. This makes the code easier to understand and maintain.

/\*\*

\* Sets the first name of the contact.

\*

\* @param fName the new first name

\* @throws IllegalArgumentException if the first name is null or longer than 10 characters

\*/

public void setFirstName(String fName) {

validateName(fName, "First Name");

this.firstName =

**Areas for Improvement in the Unit Tests**

Your unit tests for the Contact class can be improved in several areas to enhance readability, effectiveness, and correctness.

**Structure and Readability**

**Refactor Test Cases**: Consider refactoring the test cases to ensure consistency and readability. Group similar tests together and use meaningful method names.

**Assertions**

**Use** assertThrows **for Exception Tests**: For tests that expect exceptions, use assertThrows to ensure that the expected exception is thrown.

**Commenting and Documentation**

Adding comments to each test case to explain what is being tested and why. This makes the tests easier to understand and maintain.

**Improved Unit Tests**

Below is the improved version of your unit tests:

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

import org.junit.jupiter.api.Test;

class ContactTest {

@Test

void testContact() {

// Test for valid input

var newContact = new Contact("1000", "Farrik", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("Farrik", newContact.getFirstName());

assertEquals("Barnard", newContact.getLastName());

assertEquals("1000", newContact.getContactId());

assertEquals("6096657878", newContact.getPhone());

assertEquals("4011 Great Ln. Party, NJ 08754", newContact.getAddress());

}

// Test for too long ID

@Test

void testContactIdMoreThanTen() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("123456789010", "Farrik", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

});

}

@Test

void testContactIdLessThanTen() {

var newContact = new Contact("12345678", "Farrik", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("12345678", newContact.getContactId());

}

@Test

void testContactIdEqualToTen() {

var newContact = new Contact("1234567890", "Farrik", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("1234567890", newContact.getContactId());

}

// Test for null ID

@Test

void testContactIdNull() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact(null, "Farrik", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

});

}

// Test for first name

@Test

void testContactFirstNameMoreThanTen() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik23423423432", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

});

}

@Test

void testContactFirstNameLessThanTen() {

var newContact = new Contact("10001", "Farrik", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("Farrik", newContact.getFirstName());

}

@Test

void testContactFirstNameEqualToTen() {

var newContact = new Contact("10001", "Farrik1234", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("Farrik1234", newContact.getFirstName());

}

// Test for null first name

@Test

void testContactFirstNameNull() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("1001", null, "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

});

}

// Test for last name

@Test

void testContactLastNameMoreThanTen() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik1234", "Barnard23423423432", "6096657878", "4011 Great Ln. Party, NJ 08754");

});

}

@Test

void testContactLastNameLessThanTen() {

var newContact = new Contact("10001", "Farrik1234", "Barnard", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("Barnard", newContact.getLastName());

}

@Test

void testContactLastNameEqualToTen() {

var newContact = new Contact("10001", "Farrik1234", "Barnard123", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("Barnard123", newContact.getLastName());

}

// Test for null last name

@Test

void testContactLastNameNull() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik1234", null, "6096657878", "4011 Great Ln. Party, NJ 08754");

});

}

// Test for phone number

@Test

void testContactPhoneEqualToTen() {

var newContact = new Contact("10001", "Farrik1234", "Barnard123", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("6096657878", newContact.getPhone());

}

@Test

void testContactPhoneMoreThanTen() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik1234", "Barnard123", "6096657878343", "4011 Great Ln. Party, NJ 08754");

});

}

@Test

void testContactPhoneLessThanTen() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik1234", "Barnard123", "609878", "4011 Great Ln. Party, NJ 08754");

});

}

// Test for null phone

@Test

void testContactPhoneNull() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik1234", "Barnard123", null, "4011 Great Ln. Party, NJ 08754");

});

}

// Test for address

@Test

void testContactAddressMoreThanThirty() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik1234", "Barnard123", "6096657878", "4011 Great Ln. Party, NJ 08754 23423423432");

});

}

@Test

void testContactAddressLessThanThirty() {

var newContact = new Contact("10001", "Farrik1234", "Barnard123", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("4011 Great Ln. Party, NJ 08754", newContact.getAddress());

}

@Test

void testContactAddressEqualToThirty() {

var newContact = new Contact("10001", "Farrik1234", "Barnard123", "6096657878", "4011 Great Ln. Party, NJ 08754");

assertEquals("4011 Great Ln. Party, NJ 08754", newContact.getAddress());

}

// Test for null address

@Test

void testContactAddressNull() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("10001", "Farrik1234", "Barnard123", "6096657878", null);

});

}

}

**Summary of Enhancements**

The experience of improving and changing the Contact class was worthwhile and highlighted the significance of safe coding techniques, efficient error management, and unambiguous documentation. I discovered that mitigating security flaws like injection attacks requires strong input validation. The code became easier to read and maintain when the validation logic was reorganized into distinct methods. The class's ease of use and comprehension was guaranteed by the provision of detailed documentation and precise error messages.

One of the difficulties I encountered was striking a balance between comprehensive validation and understandable error messages. Extensive testing was necessary to make sure that refactoring did not introduce new bugs. Furthermore, developing thorough unit tests to cover a range of scenarios was difficult but essential to verifying that all edge cases were handled appropriately.