1 Requirements /2

A local farmer has approached you and asks you for advice for creating a web shop for his farmed products.

The farmer has been selling his fresh produce to local customers for many years, and his loyal clientele greatly appreciate the seasonal variety of fruits and vegetables he offers. To bring that same seasonal charm to his web shop, the farmer plans to showcase a selection of seasonal products and current special offers to visitors. Additionally, logged-in customers will be able to view the most recent purchases made by others. To ensure a smooth purchasing process, customers are required to create an account and provide their address and credit card details. The information provided is checked for plausibility. When placing orders, customers can choose from their previously saved addresses or enter a new one if necessary. Because the products have limited availability, the farmer wants to set the available quantity and expiry date for each product via an administrative page of the web shop. Products that expire soon should be offered at a discounted rate.

Elicit the requirements for the web shop and identify:

- the actors
- two normal scenarios and one exceptional scenario
- and three non-functional requirements.

1.1 Solution

Actors:

- The Farmer (as the role of Administrator)
- A Visitor
- A Customer (logged in)

Normal Scenarios:

- A logged in customer sees a product that he/she wants to purchase and uses their pre-set adress
- The farmer adds a new product to the web shop and sets another item to a discount as it expires soon

Exceptional Scenario: A visitor wants to buy products and creates an account. The visitor sets wrong payment information and cannot proceed (a error message may be displayed).
Non-Functional Requirements:

- Security: While being just a small online shop the users data should be protected at all times. Credentials and payment information should be encrypted, proper authentication mechanism should be implemented, etc.
- Availability: The site should have an uptime of 99%, maintenance updates should be communicated ahead of time, if the site should be unavailable a corresponding message should be displayed, etc.
- Compliance: The shop should follow all relevant laws and regulations for e-commerce shops. All transactions should be protocolled accordingly, user data is only requested and shared as needed (e.g. no tracking, no selling data, . . .)

2 Alloy 1 /2

Once upon a time, in a small town known for its love of tennis, two rival tennis clubs, the Forehand Flyers and the Backhand Blazers, found themselves facing a unique challenge. Both clubs wanted to model their structure, improve operations, and provide better services to their members. To achieve their goals, they hired you, a well known Alloy expert, to handle following tasks for them.

a) Provide a signature for the following entities:

- player (Player);
- tennis club (TennisClub), containing a set of members and a head, all of whom are players.

b) Complete fact headIsMember to ensure that the head of each tennis club is a member.

c) A tennis club is considered to be successful if it has at least 3 members. Define a predicate successful reflecting this condition.

d) Two tennis clubs are rivals if they are both successful and have no members in common. Define a predicate rivals reflecting this condition.

2.1 Solution

```
//a)

sig Player {}

sig TennisClub {
  head: one Player,
  members: set Player
}




//b)

fact headIsMember {
  all c: TennisClub | one p: c.members | p = c.head
}

//c)
```

```
pred successful[f: TennisClub] {
  #f.members >= 3
}

//run successful

//d)

pred rivals [f1 : TennisClub , f2 : TennisClub] {
  successful[f1] && successful[f2] && (f1.members & f2.members = none)
}

run rivals for 3 TennisClub, 8 Player
```

3 Alloy 2 / 1

The following Alloy model describes birthday parties where guests bring gifts. A gift has a value related to its price, ranging from 1 to 3.

```
sig Guest {
  gift : Int // value of gift
}

fact giftValue {
  all g : Guest | 1 <= g.gift and g.gift <= 3
}

pred everybodyBringsGifts {
  #(Guest) = #(Guest.gift)
}

run everybodyBringsGifts for 4 but exactly 4 Guest
```

a) This Alloy specification contains an error. Specifically, the predicate everybodyBrings-Gifts is inconsistent for 4 guests.

- Explain why the predicate is inconsistent.
- Fix the Alloy model, without altering the predicate or the run statement, such that every guest brings a gift of their own. You can add and change signatures as well as change fact giftValue.

b) In the following task, you are given a Party signature. A host considers the party to be a success, iff at least 3 invited guests bring gifts each valued at least 2. Define a predicate partyIsSuccess capturing the mentioned property.

3.1 Solution

a) The problem is that each gift is only identified by its value. There can only be 3 distinct gifts, but since there are 4 guests and each guest needs to have a distinct gift the predicate is inconsistent. See Listing 3.1

b) See Listing 3.1

Listing 1: Solution for Exercise 3a)

```
sig Gift {
  value: Int // value of gift
}
```

```
sig Guest {
  gift : one Gift
}

fact giftValue {
  all g : Guest | 1 <= g.gift.value and g.gift.value <= 3
}

pred everybodyBringsGifts {
  #(Guest) = #(Guest.gift)
}

run everybodyBringsGifts for 4 but exactly 4 Guest
```

Listing 2: Solution for 3b)

```
sig Party {
  guests : set Guest
}

pred partyIsSuccess [ p : Party ] {
#({g: p.guests | g.gift.value >= 2}) >= 3
}

run partyIsSuccess
```

4 Design by Contract 1 / 1

Consider the following Java function:

```
public int compute(int[] arr) {
    int min = arr[0];
    for (int i = 1; i < arr.length; i ++) {
        if (arr[i] < min) {
            arr[i] = min;
        }
    }
    return min;
}
```

a) Function compute makes assumptions about its input. List preconditions for input array arr such that a call to compute is guaranteed to run without throwing an exception.

b) Refactor the function to throw an IllegalArgumentException if the preconditions are not met.

c) What are its postconditions (assuming the preconditions are met)?

4.1 Solution

```
// Pre: arr is not empty
// Post: computes minimum element in arr ~~(arr remains unchanged)~~
public int compute(int[] arr) {
    if (arr.length == 0) {
        throw new IllegalArgumentException("Array is empty!");
    }

    int min = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < min) {
            arr[i] = min;
        }
    }
    return min;
}
```

5 Design by Contract 2 / 2

The interface VendingMachine models vending machines.

```
interface VendingMachine {
    Object purchase(int payment);
}
```

The method purchase has the following contract:

- Precondition: payment of at least 10.
- Postcondition: non-null purchased item.

Which of these interface implementations are allowed by the contract? If not, which part of the contract is violated?

a) Pay-as-you-like machine: You get an item regardless of the inserted payment amount.

b) Luxury vending machine: It only accepts payments larger than 20.

c) Donation machine: You can donate money, you don't get an item in return. purchase always returns null.

d) Gift dispenser: You only get an item if the payment amount is exactly 0.

e) Slot machine: You can gamble as much money as you want and have the chance of winning an item in return.

f) Soda machine: You get a can of soda for each unit of payment.

5.1 Solution

a) Yes: Pre is weaker, Post is the same ✓

b) No: Pre is stronger ✓

c) No: Pre is the weaker, Post is weaker ✓ values don't match

d) ~~Yes: Pre is weaker, Post is the same~~ No ✓

e) No: Pre is weaker, Post is weaker ✓

f) Yes: Pre is weaker, Post is stronger ✓

6 Systematic Testing /1

You recently started working in one of those notorious dynamic and young tech-startups. Right on your first day, you notice that the current team does NOT carry out ANY tests! After processing your first shock, you go straight to work. Many of the encountered errors could have been avoided or detected earlier. Your plan is to map recently reported bugs (see the list below) to appropriate test strategies, i.e., unit, integration, system, and manual testing.

For each bug listed, find the best strategy from the testing pyramid:

0. Bug #100: Deadlock between backend services after new milestone release (Milestone v3.2)
1. Bug #101: Email validation function accepts an empty string as valid email
2. Bug #102: Uncaught NullPointerException somewhere in the WebSiteBuilder class
3. Bug #103: In-app purchases trigger a race condition in the backend services
4. Bug #104: Incompatible formatted strings are passed between DataProcessor and DatabaseHandler
5. Bug #105: The HTTPProtocolParser output triggers an IncompatibleFormatException in HTTPProtocolDataExtractor
6. Bug #106: The save-button on a buyer's profile information is barely visible for a human
7. Bug #107: Some calls to the calculate method inside the CostCalculator class return undesired negative results

6.1 Solution

0. System testing ✓
1. Unit testing ✓
2. Unit testing ✓
3. System testing Integration Testing
4. Integration testing ✓
5. Integration testing ✓
6. Manual testing ✓
7. Unit testing ✓

7 Verification vs Validation /1

Lisa had recently joined a renowned software development company, eager to contribute her skills and learn from experienced professionals. One day, during a team meeting, Lisa noticed that the terms software validation and verification were being used interchangeably. Curiosity sparked within her. Knowing that you have been learning about this in your software engineering course, she asks you to help her classify the following tasks to be either software validation or verification.

a) Conducting user surveys or interviews to gather feedback on the software's usability, functionality, and overall satisfaction.

b) Testing the software on different platforms, browsers, or devices to ensure it works correctly in various environments.

c) Running unit tests to check the functionality of individual components or modules.

d) Running beta testing with real users in real-world environments to gather feedback.

e) Performing security testing to identify vulnerabilities and ensure the software is secure.

f) Monitoring and analyzing user feedback to identify any issues or areas for improvement.

g) Using formal verification techniques to mathematically prove the correctness of the software.

h) Reviewing of user stories and use cases.

i) Reviewing code segments and their test cases.

j) Develop automated test scripts to streamline the testing process and improve efficiency

7.1 Solution

a) Validation ✓

b) Verification ✓

c) Verification ✓

d) Validation ✓

e) Verification ✓

f) Validation ✓

g) Verification ✓

h) Validation ✓

i) Verification ✓

j) Verification ✓

8 Dependency Injection / 7

Consider function announceTodaysDonutDiscount of class DiscountService.

```
class DiscountService {
    public void announceTodaysDonutDiscount() {
        DiscountCalendar cal = new DiscountCalendar();
        ClientDao dao = new ClientDao();
        DiscountSender sender = new MailSender();
        for (Client client : dao.load()) {
            if (cal.isFreeDonutDay()) {
                sender.sendDiscount(client, 100);
            } else {
                sender.sendDiscount(client, 0);
            }
        }
    }
}
```

Improve the testability of this function by applying dependency injection.

8.1 Solution

```
interface DiscountCalendar {
    boolean isFreeDonutDay();
}

interface ClientDao {
    List<Client> load();
}

interface DiscountSender {
    void sendDiscount(Client c, int d);
}

class DiscountService {
    private DiscountCalendar cal;
    private ClientDao dao;
    private DiscountSender sender;

    public DiscountService
    (DiscountCalendar cal, ClientDao dao, DiscountSender sender) {
        this.cal = cal;
        this.dao = dao;
        this.sender = sender;
    }

    public void announceTodaysDonutDiscount() {
        for (Client client : dao.load()) {
            if (cal.isFreeDonutDay()) {
                sender.sendDiscount(client, 100);
            } else {
                sender.sendDiscount(client, 0);
            }
        }
    }
}
```

9 Testability / 2

a)
```
private int getDiscount(Customer customer) {
    int discount = 0;
    if (custome.hasDiscount()) {
        discount += customer.getDiscount();
    }
    if (isSale()) {
        discount += 20;
    }
    return discount;
}
```

Is domain code. ✓

```
private void sendNewsletter(Customer customer, String msg) {
    EmailClient client = new EmailClient("mail.company.com");
    client.send(customer.getEmail(), msg);
}
```

Is infrastructure code. ✓

```
private ArrayList<Product> getSimilarProducts
(Product product, DatabaseConnection connection) {
    String query = "SELECT * FROM Products WHERE Label = $label;";
    return client.query(query, product.label);
}
```

Is infrastructure code. ✓

b) For the following statements, check the appropriate boxes:

- Optimizing an algorithm improves testability because of the faster runntime. **false** ✓
- Separating infrastructure and domain code improves testability. **true** ✓
- If code size increases due to separating software into testable units the testability decreases. **false** ✓
- Proper testing strategies reveal all bugs in a system. **false** ✓
- Being able to fake dependencies increases testability. **true** ✓
- Instantiating dependencies inside the constructor instead of class methods increases testability. **false** ✓

10 Test Driven Development /1

alle Inputs

Which of the following are good reasons for applying test-driven development (TDD)?

- When using TDD, software is tested exhaustively. **it depends. Tests are just as exhaustive as the requirements are** } bad
- Many software defects are detected early. **good** ✓
- When using TDD, you think about the requirements earlier and more carefully. **good** ✓
- The tests provide feedback about the design of the code they test. **good** ✓
- You can tell the management that you follow best practices. **not necessarily, you ideally always follow best practices regardless** ✓
- Code developed with TDD does not require additional documentation. **false** ✓
- Code developed with TDD is designed to be testable. **not necessarily, all code should be testable** } good
- Developers can choose the pace at which they develop code. **good** ✓