Exercise 2: Alloy 1

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.

```
1 /* TODO: a) */
2
3 fact headIsMember { /* TODO: b) */ }
4
5 pred successful[f:TennisClub] { /* TODO: c) */ }
6
7 pred rivals[f1: TennisClub, f2: TennisClub] { /* TODO: d) */ }
```

```
// a)
sig Player {}

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

// b)
fact headIsMember { all tc: TennisClub | tc.head in tc.members }

// c)
pred successful[f:TennisClub ] { #f.members ≥ 3 }

// d)
pred rivals [ f1 : TennisClub , f2 : TennisClub ] {
    successful(f1) &
    sucessful(f2) &
    (all p: Player | !{p in f1.members ∧ p in f2.members})
}
```

Exercise 3: Alloy 2

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 `everybodyBringsGifts` 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.

```
sig Party {
  guests: set Guest,
}

pred partyIsSuccess[p: Party] { /* TODO: b) */ }
```

```
// a)
// The cardinality #a counts how many members are in a set
// The predicate is inconsistent, because if we have 4 guests
// #(Guest)=4, so in order to satisfy the fact the cardinality
// of #(Guest.gift), which is Int also needs to be 4.
// But because it can only take values beween 1 and 3, the set
// has at most 3 members, which means the cardinality
// can maximally be 3.

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

fact giftValue { all g: Guest | 1 ≤ g.gift }

pred everybodyBringsGifts { #(Guest) = #(Guest.gift) }
//run everybodyBringsGifts for 4 but exactly 4 Guest

// b)
sig Party {
    guests: set Guest
}

pred partyIsSuccess[p: Party] { #{g: p.guests | g.gift ≥ 2} ≥ 3 }
run partyIsSuccess
```

Exercise 4: Design by Contract 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)?

```
/* Preconditions:
* arr is not empty (has at least one element)
*
* Postconditions:
* Every element of arr is >= arr[0]
*/
public int compute (int[] arr) {
    // Refactoring to throw IllegalArgumentException if
        ↪ preconditions are not met
    if (arr.length < 1) {
        throw new IllegalArgumentException("Preconditions not
            ↪ met");
    }
    int min = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < min) {
            arr[i] = min ;
        }
    }
    return min;
}
```

Exercise 5: Design by Contract 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.

`Allowed`

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

`Not allowed, the precondition is stricter than the contract precondition`

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

`Not allowed, the non-null purchased item postcondition is violated`

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

`Not allowed, the non-null purchased item postcondition is violated`

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

`Not allowed, the non-null purchased item postcondition is violated`

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

`Allowed`

Exercise 6: Systematic Testing

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:

- Bug #100: Deadlock between backend services after new milestone release (Milestone v3.2)

  `System Testing`

- Bug #101: Email validation function accepts an empty string as valid email

  `Unit Testing`

- Bug #102: Uncaught `NullPointerException` somewhere in the `WebSiteBuilder` class

  `Unit Testing`

- Bug #103: In-app purchases trigger a race condition in the backend services

  `System Testing`

- Bug #104: Incompatible formatted strings are passed between `DataProcessor` and `DatabaseHandler`

  `Integration Testing`

- Bug #105: The `HTTPProtocolParser` output triggers an `IncompatibleFormatException` in `HTTPProtocolDataExtractor`

  `Integration Testing`

- Bug #106: The *save*-button on a buyer's profile information is barely visible for a human

  `Manual Testing`

- Bug #107: Some calls to the `calculate` method inside the `CostCalculator` class return undesired negative results

  `Unit Testing`

Exercise 7: Verification vs Validation

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.

`Validation`

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

`Verification`

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

`Verification`

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

`Validation`

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

`Verification`

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

`Validation`

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

`Verification`

h) Reviewing of user stories and use cases.

`Validation`

i) Reviewing code segments and their test cases.

`Verification`

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

`Verification`

Exercise 8: Dependency Injection

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.

```
class DiscountService {
    private DiscountCalendar _cal;
    private ClientDao _dao;
    private DiscountSender _sender;

    public DiscountService(DiscountCalendar cal, ClientDao dao,
         DiscountSender mailSender) {
        this._cal = cal;
        this._dao = dao;
        this._sender = mailSender;
    }

    public void announceTodaysDonutDiscount() {
        for (Client client : this._dao.load()) {
            if (this._cal.isFreeDonutDay()) {
                this._sender.sendDiscount (client, 100);
            } else {
                this._sender.sendDiscount(client, 0);
            }
        }
    }
}
```

Exercise 9: Testability

a) Label the following methods as *domain* or *infrastructure* code:

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

☒ domain ☐ infrastructure

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

☐ domain ☒ infrastructure

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

☐ domain ☒ infrastructure

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

- Optimizing an algorithm improves testability because of the faster runtime.

  ☐ true ☒ false
- Separating infrastructure and domain code improves testability.

  ☒ true ☐ false
- If code size increases due to separating software into testable units the testability decreases.

  ☐ true ☒ false
- Proper testing strategies reveal all bugs in a system.

  ☐ true ☒ false
- Being able to fake dependencies increases testability.

  ☒ true ☐ false
- Instantiating dependencies inside the constructor instead of class methods increases testability.

  ☐ true ☒ false

Task 10: Test Driven Development

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

- When using TDD, software is tested exhaustively.
- Many software defects are detected early.
- When using TDD, you think about the requirements earlier and more carefully.
- The tests provide feedback about the design of the code they test.
- You can tell the management that you follow best practices.
- Code developed with TDD does not require additional documentation.
- Code developed with TDD is designed to be testable.
- Developers can choose the pace at which they develop code.