Assignment 1 (solution)

Exercise 1 - Design and Documentation

1. The design decisions would make sense if we would rarely modify the `source` and the `target` - otherwise it would make more sense to have them as arguments to `shortestPath`.

2. No, the shortest path does not exist if the `source` and the `target` are not connected through edges. In that case, the method should return `null`. If the `source` and the `target` are equal, then the `shortestPath` should return an empty list. Otherwise, the method should return a list of nodes (without including `source` and `target`), such that the sum of the `distances` of the corresponding edges is minimum.

3. In order to improve the efficiency we could cache the shortest path by recomputing it lazily or eagerly.

   (a) Lazily:

```
class STGraph{
  List<Edge> edges;
  Node source;
  Node target;

  private List<Node> sp;

  public STGraph(Node source, Node target){
    this.source = source;
    this.target = target;
    this.edges = new ArrayList<Edge>();
    sp = null;
  }

  public void setST(Node source, Node target){
    this.source = source;
    this.target = target;
    sp = null;
  }

  void addEdge(Edge e){
    edges.add(e);
    sp = null;
  }

  List<Node> shortestPath(){
    if(sp == null && areConnectedST()){
      sp = computeShortestPath();
    }
    return sp;
  }

  private boolean areConnectedST(){...}

  private List<Node> computeShortestPath(){...}
}
```

(b) This should not influence the client-visible documentation, except perhaps for the memory consumption.

(c) One could add the following documentation:

   i. postcondition for `addEdge`: `sp == null`

   ii. postcondition for `shortestPath`: `old(sp == null && areConnectedST())|| result == sp`

(d) Eagerly:

```
class STGraph{
  List<Edge> edges;
  Node source;
  Node target;

  private List<Node> sp;

  public STGraph(Node source, Node target){
    this.source = source;
    this.target = target;
    this.edges = new ArrayList<Edge>();
    sp = computeShortestPath();
  }

  public void setST(Node source, Node target){
    this.source = source;
    this.target = target;
    sp = computeShortestPath();
  }

  public void addEdge(Edge e){
    edges.add(e);
    sp = computeShortestPath();
  }

  public List<Node> shortestPath(){
    return sp;
  }

  private boolean areConnectedST(){...}

  private List<Node> computeShortestPath(){
    if(!areConnectedST()){
      return null;
    }
    // do the actual computation
    // ...
  }
}
```

- This should not influence the client-visible documentation, except for the complexity of the constructor, `setST` and `addEdge`.

- One could add the following documentation:
  - (a) class invariant for class `STGraph`: `areConnectedST()=> sp != null`
  - (b) postcondition for `shortestPath`: `result == sp`

We could also think of a more involved design where we keep the intermediate data of the shortest path algorithm and update it incrementally when adding edges.

Exercise 2 - Design

1. This is one possible scenario:
   (a) A new list 'a' is created (without keeping the reference to the array passed to the constructor).
   (b) This list 'b' is obtained by calling the method `take` on 'a'.
   (c) The list 'b' is modified by calling the method `set`.
   (d) The list 'a' is modified by calling the method `set` and the `elems` are cloned even though the array is technically not shared anymore

```
List<Integer> a = new List<Integer>(new Integer[]{10, 20, 30}, 3);
List<Integer> b = a.take();
b.set(0, -5);
a.set(1, 40);
```

2. One could use actual reference counting instead of using the boolean field `shared`. This field has to be shared between the `List` objects, so we cannot just replace the `boolean` with an `int`. For this reason, we decided to create the wrapper class `NumberOfReferences`, as showed below:

```
class NumberOfReferences{
    int counter;

    NumberOfReferences(int counter){
      this.counter = counter;
    }

    void increase(){
      counter ++;
    }

    void decrease(){
      counter --;
    }

    int getValue(){
      return counter;
    }
}


class List<E> {
  E[] elems;
  int len;
  NumberOfReferences nr;

  List(E[] e, int l, NumberOfReferences nr){
    elems = e;
    len = l;
    this.nr = nr;
    this.nr.increase();
  }

  void set(int index, E e){
    if(nr.getValue() > 1){
     elems = elems.clone();
     nr.decrease();
     nr = new NumberOfReferences(1);
    }
    elems[index] = e;
  }

  List<E> take(){
    return new List<E>(elems, len - 1, nr);
  }
}
```

3. No, the above solution cannot handle the following scenario:
   (a) A new list 'a' is created (`List<Integer> a = new List<Integer>(new Integer[]{10, 20, 30}, 3, new NumberOfReferences(0));`).
   (b) The list 'b' is obtained by calling the method `take` on 'a'.
   (c) The list 'b' is not used anymore and is removed from the heap by the garbage collector.
   (d) The list 'a' is modified by calling the method `set` and the `elems` are still cloned, because the `nr.getValue()` is still 2.

   To fix this inefficiency, one could implement a `finalize` method, where the `numberOfReferences` in decreased before the object is eventually removed from the heap:

```
class List<E> {
  // ...
  // same as before
  // ...

  @Override
  protected void finalize(){
    nr.decrease();
  }
}
```

Exercise 3 - Requirements Elicitation

There is no authoritative solution to this exercise since it depends on the discussion in the exercise session. The following should be mainly seen as hints:

- Actors:
    - Customer
    - Flower Shop Manager
    - Messenger
- Some open issues:
    - How does the messenger communicate with the web browser?
    - How are undelivered flower orders handled?
    - Can he use the system to sell faster the flowers that will expire soon?
    - Who is going to host the system?
- Scenarios:
    - Scenario 1 (normal)
        1. Jill wishes to purchase some flowers.
        2. She logs into the internet browser with her user name and password.
        3. She selects the flowers that she likes and presses check-out.
        4. For the address, she selects her home address.
        5. Jill pays with her credit card.
        6. The system offers her a receipt for the delivery and her credit card is charged.
    - Scenario 2 (exceptional)
        1. Bob wants to become a frequent customer for the web shop.
        2. He enters the URL of the shop and selects new customer.
        3. Bob gives his preferred username.
        4. The system finds out that the selected username already exists and notifies Bob that he has to choose a different one.
    - Scenario 3 (unspecified)
        * Sarah has already checked out and printed her receipt.
        * She realizes that the address she selected was incorrect.
        * She immediately logs back in and selects her last order.
        * The system tells her that it hasn't been prepared yet and that she is able to modify the order.
        * She changes the address to the correct one and prints out the new receipt.
- Non-functional requirements:
    - The clients should be able to use standard web browsers.
    - The response time of the system should be within 3 seconds.
    - The system should support at least 400 clients.
    - The system should use the existing point of sales system.