Software Engineering Design 2022

These are raw solutions straight out of the exam and have not been peer reviewed or checked in any way.

Q1

a)

First Test:

```
private PaymentSystem payment = context.mock(PaymentSystem.class);

  AuctionManager auction = new AuctionManager(payment);

  @Test
  public void bidsCanBePlacedInTheAuction() {
    auction.startAuction("Item", "Seller");

    context.checking(new Expectations() {{
      oneOf(payment).charge(10, "Alice");
    }});

    BidOutcome outcome = auction.bid(10, "Alice");

    assertEquals(BID_ACCEPTED, outcome);
  }
```

First Implementation:

PaymentSystem.java

```
package ic.doc;

public interface PaymentSystem {
  void charge(int price, String customer);

  void pay(int price, String recipient);
}
```

BidOutcome.java

```
package ic.doc;

public enum BidOutcome {
  BID_ACCEPTED,
```

```
    BID_TOO_LOW
  }
```

AuctionManager.java

```
public class AuctionManager {

  private final PaymentSystem payment;

  public AuctionManager(PaymentSystem payment) {
    this.payment = payment;
  }

  public void startAuction(String item, String seller) {
  }

  public BidOutcome bid(int price, String customer) {
    payment.charge(price, customer);
    return BidOutcome.BID_ACCEPTED;
  }
}
```

b)

Second Test:

```
@Test
  public void bidsLowerThanCurrentMaximumAreRejected() {
    placeInitialBid();

    context.checking(new Expectations() {{
      never(payment);
    }});

    BidOutcome outcome = auction.bid(5, "Carole");

    assertEquals(BID_TOO_LOW, outcome);
  }

  private BidOutcome placeInitialBid() {
    auction.startAuction("Item", "Seller");

    context.checking(new Expectations() {{
      oneOf(payment).charge(10, "Alice");
    }});

    BidOutcome outcome = auction.bid(10, "Alice");
    return outcome;
  }
```

Second Implementation:

```
public class AuctionManager {

  private final PaymentSystem payment;
  private int currentMaximum = 0;
  private String currentHighestBidder;

  public AuctionManager(PaymentSystem payment) {
    this.payment = payment;
  }

  public void startAuction(String item, String seller) {
  }

  public BidOutcome bid(int price, String customer) {
    if (price > currentMaximum) {
      currentMaximum = price;
      currentHighestBidder = customer;

      payment.charge(price, customer);

      return BidOutcome.BID_ACCEPTED;
    } else {
      return BidOutcome.BID_TOO_LOW;
    }
  }
}
```

c)

Third Test:

```
@Test
  public void newBidsHigherThanCurrentMaximumAreAccepted() {
    placeInitialBid();

    context.checking(new Expectations() {{
      oneOf(payment).charge(20, "David");
    }});

    BidOutcome outcome = auction.bid(20, "David");

    assertEquals(BID_ACCEPTED, outcome);
  }
```

Third Implementation:

```
public class AuctionManager {

  private final PaymentSystem payment;
  private int currentMaximum = 0;
  private String currentHighestBidder;

  public AuctionManager(PaymentSystem payment) {
    this.payment = payment;
  }

  public void startAuction(String item, String seller) {
  }

  public BidOutcome bid(int price, String customer) {
    if (price > currentMaximum) {
      currentMaximum = price;
      currentHighestBidder = customer;

      payment.charge(price, customer);

      return BidOutcome.BID_ACCEPTED;
    } else {
      return BidOutcome.BID_TOO_LOW;
    }
  }
}
```

d)

Fourth Test:

```
private final Dispatcher dispatcher = context.mock(Dispatcher.class);

 @Before
  public void startAuction() {
    auction.startAuction("Item", "Seller");
  }

@Test
  public void moneyIsTransferredAndItemIsDispatchedWhenAuctionEnds() {
    placeInitialBid();

    context.checking(new Expectations() {{
      ignoring(payment).charge(20, "David");
    }});

    BidOutcome outcome = auction.bid(20, "David");

    context.checking(new Expectations() {{
      oneOf(payment).pay(20, "Seller");
      oneOf(payment).pay(10, "Alice");
```

```
      oneOf(dispatcher).dispatch("Item", "David");
    }});

    auction.endAuction();
  }

  private BidOutcome placeInitialBid() {
    context.checking(new Expectations() {{
      ignoring(payment).charge(10, "Alice");
    }});

    return auction.bid(10, "Alice");
  }
```

Fourth Implementation:

Bid.java

```
package ic.doc;

import java.util.Objects;

public class Bid {
  private final String bidder;
  private final int amount;

  public Bid(String bidder, int amount) {
    this.bidder = bidder;
    this.amount = amount;
  }

  public String getBidder() {
    return bidder;
  }

  public int getAmount() {
    return amount;
  }

  // Hashcode and Equals as Bids are being stored in a data structure
  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null || getClass() != o.getClass()) {
      return false;
    }
    Bid bid = (Bid) o;
    return amount == bid.amount && Objects.equals(bidder, bid.bidder);
  }
```

```
  @Override
  public int hashCode() {
    return Objects.hash(bidder, amount);
  }
}
```

Dispatcher.java

```
package ic.doc;

public interface Dispatcher {
  void dispatch(String item, String recipient);
}
```

AuctionManager.java

```
public class AuctionManager {

  private final PaymentSystem payment;
  private final Dispatcher dispatcher;

  private Bid highestBid;

  private final List<Bid> previousHighest = new ArrayList<>();

  private String item;
  private String seller;

  public AuctionManager(PaymentSystem payment, Dispatcher dispatcher) {
    this.payment = payment;
    this.dispatcher = dispatcher;
  }

  public void startAuction(String item, String seller) {
    this.item = item;
    this.seller = seller;
  }

  public BidOutcome bid(int price, String customer) {
    // Initial bidder
    if (highestBid == null) {
      replaceBid(price, customer);

      return BidOutcome.BID_ACCEPTED;
    } else if (price > highestBid.getAmount()) {
      previousHighest.add(highestBid);
      replaceBid(price, customer);

      return BidOutcome.BID_ACCEPTED;
```

```
    }

    return BidOutcome.BID_TOO_LOW;
  }

  private void replaceBid(int price, String customer) {
    highestBid = new Bid(customer, price);
    payment.charge(price, customer);
  }

  public void endAuction() {
    payment.pay(highestBid.getAmount(), seller);
    dispatcher.dispatch(item, highestBid.getBidder());

    // Refund other successful bids
    for (Bid failed : previousHighest) {
      payment.pay(failed.getAmount(), failed.getBidder());
    }
  }
}
```

Q2

a)

Pattern:

```
Singleton Pattern
```

Create field for the instance which is constructed at class instantiation:

```
private static final MediaLibrary instance = new MediaLibrary();
```

Change constructor of MediaLibrary to be private:

```
private MediaLibrary() {
    ...
}
```

Provide a getInstance method to retrieve the instance:

```
public static MediaLibrary getInstance()
{
    return instance;
}
```

Update usage in VideoStreamer.java:

```
List<Movie> recommendations =
MediaLibrary.getInstance().recommendedMoviesFor(user);
```

b)

New ContentLibrary.java Interface:

```
package ic.doc;

import ic.doc.movies.Movie;
import java.util.List;

public interface ContentLibrary {
  List<Movie> recommendedMoviesFor(User user);
}
```

MediaLibrary implements ContentLibrary:

```
public class MediaLibrary implements ContentLibrary {
  ...
}
```

VideoStreamer takes a content library as a parameter and queries this for recommended movies:

```
private final ContentLibrary library;

public VideoStreamer(ContentLibrary library) {
    this.library = library;
  }

 public List<Movie> getSuggestedMovies(User user) {
    List<Movie> recommendations = library.recommendedMoviesFor(user);

    // sort the list of suggestions in descending order of number of views
    List<Movie> suggestions = new ArrayList<>(recommendations);
    suggestions.sort(Comparator.comparing(Movie::numberOfViews).reversed());
    return suggestions;
  }
```

New test added without using MovieLibrary:

```
  private final ContentLibrary library = context.mock(ContentLibrary.class);

  @Test
    public void getSuggestedMoviesProvidesListOfMovies() {

      VideoStreamer streamer = new VideoStreamer(library);
      User user = new User("Bob", 3);

      context.checking(new Expectations() {{
        oneOf(library).recommendedMoviesFor(user);
        will(returnValue(EXAMPLE_MOVIES));
      }});

      List<Movie> movies = streamer.getSuggestedMovies(user);

      assertEquals(1, movies.size());
    }
```

c)

New Interface EventLog:

```
package ic.doc.streaming;

import ic.doc.User;
import ic.doc.movies.Movie;

public interface EventLog {
  void logWatched(User user, Movie movie);

  void logRejection(User user, Movie movie);
}
```

PlaybackEventLog implements EventLog:

```
public class PlaybackEventLog implements EventLog {
  ...
}
```

New Interface TimeQuery:

```
package ic.doc;

import java.time.LocalTime;

public interface TimeQuery {
```

```
  LocalTime now();
}
```

StreamTracker takes a TimeQuery as a parameter and sets start time based on it:

```
public class StreamTracker {
  private final User user;
  private final LocalTime timestamp;

  public StreamTracker(User user, TimeQuery clock) {
    this.user = user;
    this.timestamp = clock.now();
  }

  public LocalTime startTime() {
    return timestamp;
  }

  public User user() {
    return user;
  }
}
```

VideoStreamer now receives and stores an EventLog instead of a PlaybackEventLog, and a TimeQuery:

```
  private final EventLog playbackEvents;

  private final ContentLibrary library;
  private final TimeQuery clock;

  public VideoStreamer(ContentLibrary library, TimeQuery clock, EventLog log) {
    this.library = library;
    this.clock = clock;
    this.playbackEvents = log;
  }
```

StreamTrackers are constructed using this TimeQuery:

```
currentStreams.put(stream, new StreamTracker(user, clock));
```

The end time of streams is queried using this TimeQuery:

```
public void stopStreaming(VideoStream stream) {
    StreamTracker streamTracker = currentStreams.remove(stream);
    LocalTime endTime = clock.now();
```

```
    long minutesWatched = ChronoUnit.MINUTES.between(streamTracker.startTime(),
endTime);
    if (minutesWatched > 15) {
      playbackEvents.logWatched(streamTracker.user(), stream.movie());
    }
  }
```

By mocking the log and time query, we can add tests which verify this behaviour:

```
public class VideoStreamerTest {

  @Rule
  public JUnitRuleMockery context = new JUnitRuleMockery();

  private final ContentLibrary library = context.mock(ContentLibrary.class);
  private final TimeQuery mockClock = context.mock(TimeQuery.class);
  private final EventLog mockLog = context.mock(EventLog.class);

  VideoStreamer streamer = new VideoStreamer(library, mockClock, mockLog);
  User user = new User("Bob", 3);

  private static final Movie EXAMPLE_MOVIE = new Movie("No Time To Die",
      "Another installment of the James Bond franchise",
      1342365,
      List.of(new Actor("Daniel Craig")),
      Set.of(Genre.ACTION, Genre.ADVENTURE),
      EMPTY_LIST,
      TWELVE_A
  );
  private static final List<Movie> EXAMPLE_MOVIES = List.of(EXAMPLE_MOVIE);

  @Test
  public void allowsUserToStreamSuggestedMovies() {
    context.checking(new Expectations() {{
      allowing(mockClock).now();
      will(returnValue(LocalTime.now()));
    }});

    VideoStreamer streamer =
        new VideoStreamer(MediaLibrary.getInstance(), mockClock, new
PlaybackEventLog());
    User user = new User("Adam", 9);

    List<Movie> movies = streamer.getSuggestedMovies(user);
    VideoStream stream = streamer.startStreaming(movies.get(0), user);

    // adam watches the movie

    streamer.stopStreaming(stream);
  }

  @Test
```

```
  public void getSuggestedMoviesProvidesListOfMovies() {

    context.checking(new Expectations() {{
      oneOf(library).recommendedMoviesFor(user);
      will(returnValue(EXAMPLE_MOVIES));
    }});

    List<Movie> movies = streamer.getSuggestedMovies(user);

    assertEquals(1, movies.size());
  }

  @Test
  public void moviesWatchedForUnder15MinutesAreNotLogged() {
    context.checking(new Expectations() {{
      never(mockLog);
    }});

    watchMovieBetween(LocalTime.of(15, 00), LocalTime.of(15, 14));
  }

  @Test
  public void moviesWatchedForOver15MinutesAreLogged() {
    context.checking(new Expectations() {{
      oneOf(mockLog).logWatched(user, EXAMPLE_MOVIE);
    }});

    watchMovieBetween(LocalTime.of(15, 00), LocalTime.of(16, 59));
  }

  @Test
  public void moviesWatchedForExactly15MinutesAreNotLogged() {
    context.checking(new Expectations() {{
      never(mockLog);
    }});

    watchMovieBetween(LocalTime.of(15, 00), LocalTime.of(15, 15));
  }

  private void watchMovieBetween(LocalTime start, LocalTime end) {
    context.checking(new Expectations() {{
      oneOf(mockClock).now();
      will(returnValue(start));

      oneOf(mockClock).now();
      will(returnValue(end));
    }});

    VideoStream stream = streamer.startStreaming(EXAMPLE_MOVIE, user);
    streamer.stopStreaming(stream);
  }
```